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] Carleton University, Ottawa, Canada
(This research was carried out at Cloakware Corp.)
{stanley.chow, phil.eisen, harold.johnson}@cloakware.com,
vanoorschot@scs.carleton.ca

Abstract. For applications such as digital rights management (DRM) solutions employing cryptographic implementations in software, *white-box* cryptography (or more formally: a 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, and our objective is to prevent the extraction of secret keys from the program. We present methods to make key extraction difficult in the white-box context, 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,

* Version: Oct 15, 2002; Pre-proceedings for ACM DRM-2002 workshop.

performing arbitrary static analyses on the software, and altering results of subcomputation (e.g. via breakpoints) for perturbation analysis.

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 and permutations 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 [4]. The main concept is the encoding of a transformation, we use this extensively in this paper — we consider a substitution-box lookup to be a transformation; we also consider the whole DES to be a transformation. We use input/output encodings to protect all these transforms.

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.

In many places, the transformations have wide inputs and/or outputs (in the DES construction, some are 96 bits input and output), which means the implementation can get very large. To avoid huge tables, we can construct an encoding as the *concatenation* of smaller bijections. Consider bijections F_i of size n_i, where $n_1 + n_2 + \ldots + n_k = n$. Let $\|$ denote vector concatenation.

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 $F^{-1} = F_1^{-1} \| F_2^{-1} \| \ldots \| F_k^{-1}$.*

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.

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*

$$Y' \circ X' \quad = \quad (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) \quad = \quad H \circ (Y \circ X) \circ F^{-1} \ .$$

In the subsequent sections, it is often useful to keep track of the sizes of the inputs and outputs. We use the following notations.

P' denotes an encoded implementation derived from the function P. ${}^{n}_{m}P$ denotes P, emphasizing that it maps m-vectors to n-vectors. For a matrix M, ${}^{n}_{m}M$ denotes explicitly that M has m columns and n rows. (Interpreting application of M to a vector as function application, this notation is the same as above, though there is not necessarily a matrix representation M for every arbitrary function P.)

${}^{k}_{k}\mathbf{I}$ is the identity function on k-vectors. ${}^{n}_{m}\mathbf{E}$ (a mnemonic for an *entropy*-transference function) is any function from m-vectors to n-vectors such that, if $m \leq n$, the mapping loses no bits of information, and if $m > n$, it loses at most $n - m$ bits.

$\langle e_1, e_2, e_3, \ldots, e_k \rangle$ is a k-vector with elements e_i; it will be evident from context whether the elements are bits. $x \| y$ is the concatenation of vectors x and y. v_i is the ith element. $v_{i \cdot\cdot j}$ is the sub-vector containing elements i through j. ${}_{k}v$ denotes explicitly that v has k elements. ${}_{k}\mathbf{e}$ is any vector with k elements (mnemonically: an *entropy*-vector). $\oplus$ denotes bitwise *xor*.

AT (mnemonic: affine transformation) denotes a vector-to-vector transformation function P which can be defined for all ${}_{m}\mathbf{e}$ by ${}^{n}_{m}P({}_{m}\mathbf{e}) = {}^{n}_{m}M_{m}\mathbf{e} + {}_{n}d$, or $P(\mathbf{e}) = M\mathbf{e} + d$, where M is a constant matrix and d is a constant *displacement* vector. We consider ATs over GF(2). Note that if A and B are ATs, then so are $A \| B$ and $A \circ B$ (where defined).

3 Producing Encoded Implementations

As is well known, DES consists of "permutations", S-Box lookups and *xor* operations. We will follow our approach of applying encodings to each of these steps. For S-Box lookups and *xor* operations, encoding each operation (along with its input and output) seem to produce good security. For the various permutations, the problem is more difficult.

These permutations are, by nature, very simple; and it is difficult to hide the information being manipulated. We address this by changing the domain from permutations to linear algebra so that we have more tools at our disposal. We start by expressing each of the DES permutations and the bitwise *xor* operations as ATs. Clearly, the resultant ATs are still very simple and do not hide information well, but subsequent use of non-linear encoding (see 4) will make such hiding much more effective.

3.1 Some techniques

In general, we are working towards an implementation consisting entirely of substitution boxes, none of which implement ATs. Several intermediate methods are needed to transform a normal cipher implementation to such a form. We describe such intermediate methods below. We will use a subset of these in our DES example.

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

Mixing Bijection A *mixing bijection* is a bijective AT which attempts to maximize the dependency of each output bit on all input bits. (Clearly, it is invertible and the inverse is also an AT.)

In DES, for example, the permutations, represented as ATs, have very sparse matrices (one or two 1-bits per row or column). In order to diffuse information over more bits, we can represent such a permutation P by $J \circ K$, where K is a mixing bijection and $J = PK^{-1}$, thereby replacing a sparse matrix with two non-sparse ones (which is advantageous in subsequent de-linearizing encoding steps).

I/O-Blocked Encoding An arbitrary ${}^{n}_{m}P$, where m and n are large, cannot simply be encoded using two arbitrary bijective encodings as $P' = G \circ P \circ F^{-1}$ using a substitution-box representation, since the size of a substitution-box varies exponentially with the number of input bits.

We can produce practically implementable encodings for such a P by dividing its input into j blocks of a bits each, and its output into k blocks of b bits each, so that $m = ja$ and $n = kb$. Let ${}^{m}_{m}J$ and ${}^{n}_{n}K$ be two *mixing bijections* (see above).

We choose (arbitrary) coding bijections for each block of the input and output: ${}^{a}_{a}F_1, \ldots, {}^{a}_{a}F_j$ and ${}^{b}_{b}G_1, \ldots, {}^{b}_{b}G_k$. 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.

(We still have the problem of how wide-input ATs are to be represented by networks of substitution boxes. Methods for this are described in §4.)

This permits us to use *networked encoding* (def. 3) 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 AT X, and the output from P is used by an AT 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 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, making it harder for an attacker to separate and 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 In general, we want the implementation of each transform to have extra entropy at both the input and output, so that it is more difficult to

identify the transform. For example, for a transform ${}^{n}_{m}P$, we carry a extra bits of entropy at the input, and b extra bits of entropy at the output, where $a \geq b$, we can encode ${}^{n+b}_{m+a}P'$ as $G \circ (P \| {}^{b}_{a}\mathbf{E}) \circ F^{-1}$. We call ${}^{b}_{a}\mathbf{E}$ the *by-pass* component of P'. (If ${}^{b}_{a}\mathbf{E}$ has the specific form ${}^{a}_{a}\mathbf{I}$, so that $a = b$, we call it *identity by-pass*.)

Split-Path Encoding For a function ${}^{n}_{m}P$, we may use an encoding that is really a concatanation of two separate encodings. That is, we define ${}^{n+k}_{m}Q({}_{m}\mathbf{e}) = P({}_{m}\mathbf{e}) \| {}^{k}_{m}R({}_{m}\mathbf{e})$ for all ${}_{m}\mathbf{e}$, for some fixed function R. The effect is that, if P is lossy, Q may lose less information (or no information). We sometimes use this method to achieve *local security* (see §3.2.)

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, we may encode a function P as $k \geq 2$ functions, $P_1, P_2, \ldots, P_k$, where the encoded implementation for each part can mix in additional entropy as described 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 ${}^{n}_{m}P$, we can choose $k = 2$, define ${}^{n}_{m}P_1$ to be a randomly selected fixed ${}^{n}_{m}\mathbf{E}$, and define ${}^{n}_{m}P_2({}_{m}\mathbf{e}) = P({}_{m}\mathbf{e}) \oplus P_1({}_{m}\mathbf{e})$ for all ${}_{m}\mathbf{e}$.

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 AT into information about P's output.

3.2 Substitution Boxes and Local Security

We can represent any function ${}^{n}_{m}P$ by a *substitution box* (S-Box): an array of 2^m entries, each of n bits. 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* in the sense that it is not possible to extract any useful information by examining the encoded S-Box. The reason is that given a S-Box for P', every possible bijective P fits (just as in an One-Time Pad, every possible plaintext can result in every possible ciphertext). Of course, this just means that an attack must be *non*-local.

The lossy case is not *locally secure*. When a *slightly* lossy encoded function is represented as an S-Box, *some* information about the function beyond its input and output widths can be found by examining its S-Box. While it still requires non-local attacks to *completely* unravel such a lossy box, it often leaks enough information to allow attacks such as the statistical bucketing attack (see §5.4).

4 Wide-Input Encoded ATs: Building Encoded Networks

Clearly, if we want to construct a S-Box with wide-input, say 32 bits or even 96 bits, we will quickly use immense amounts of storage. This means we cannot represent a wide-input encoded AT by an S-Box. We can, however, construct *networks* of S-Boxes to implement a wide-input encoded AT. 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 ${}^n_m A'$. The form of the network can remain invariant except for variations in the bit patterns within its S-Boxes.

For an AT A, we simply partition the matrix and vectors into blocks, giving us 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 the methods in §3.1 above, 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 ${}_m\mathbf{e}$. 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 $\sum_1^{m_\#} m_i = m$ and $\sum_1^{n_\#} n_i = n$. 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, jth block in partitioned M contains m_i columns and n_j rows, the ith partition of the input contains m_i elements, and the jth partition 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 representing it as a network of S-Boxes (see §3.2.) In such a network,*none* of the subcomputations is linear: each is encoded and represented as a non-linear S-Box.

A naive version of this consists of a forest of $n_\#$ trees of binary 'vector add' S-Boxes, 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 S-Boxes.

A potential weakness of this entire approach is that the blocking of A may produce blocks, such as 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, so far as we have yet determined, such blocked implementations remain combinatorially quite difficult to crack, especially if we apply the following proposal.

To address the above potential weakness, instead of encoding ${}^{n}_{m}A$, find transforms ${}^{n}_{m}A_1$ and ${}^{m}_{m}A_2$, such that A_2 is a mixing bijection (see §3.1), and $A_1 = A \circ A_2^{-1}$. Encode the two functions separately into networks of S-Boxes, and connect the outputs of the A_2' representation to the inputs of the A_1' representation, thus creating a representation of $A_1' \circ A_2' = A'$.

While the above proposal helps, 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 ${}^{kn}_{kn}M$ 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 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 will start with a simple construction with weaknesses, both in security and efficiency. We discuss how to address these in §5.3.

5.1 Replacing the DES SBs

DES is performed in 16 rounds, each employing the same eight 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]).

In Figure 1(a), we see an unrolling of two typical DES rounds. The round structure implements a Feistel network with a by-pass left-side data-path ($\mathbf{L}_{r-1}$, $\mathbf{L}_r$, $\mathbf{L}_{r+1}$ in the figure) and an active right-side data-path (everything else in the figure). $\mathbf{K}_r$ is the round subkey of round r.

???Check change fig 1b to be unrolled to match fig 1a??? The encoded implementation will look like Figure 1(b), where each round is represented by 12 **T** boxes. Between rounds, the left and right sides are combined into a single 96 bit representation and we uses a single ${}^{r}\mathbf{M}_2$ transform to to subsume the P-Box, *xor*, side flip, and E-box. (See The Transfer Functions in §5.2 for a more detailed description.)

We also need $\mathbf{M}_1$ to do an initial expansion of the input from 64 bits to the internal 96 bits form and $\mathbf{M}_3$ for the final shrink of the output.

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, determined by the algorithm and the key. We can merge the 'predictable' information and the DSBs into new S-Boxes that are dependent on the key and round.

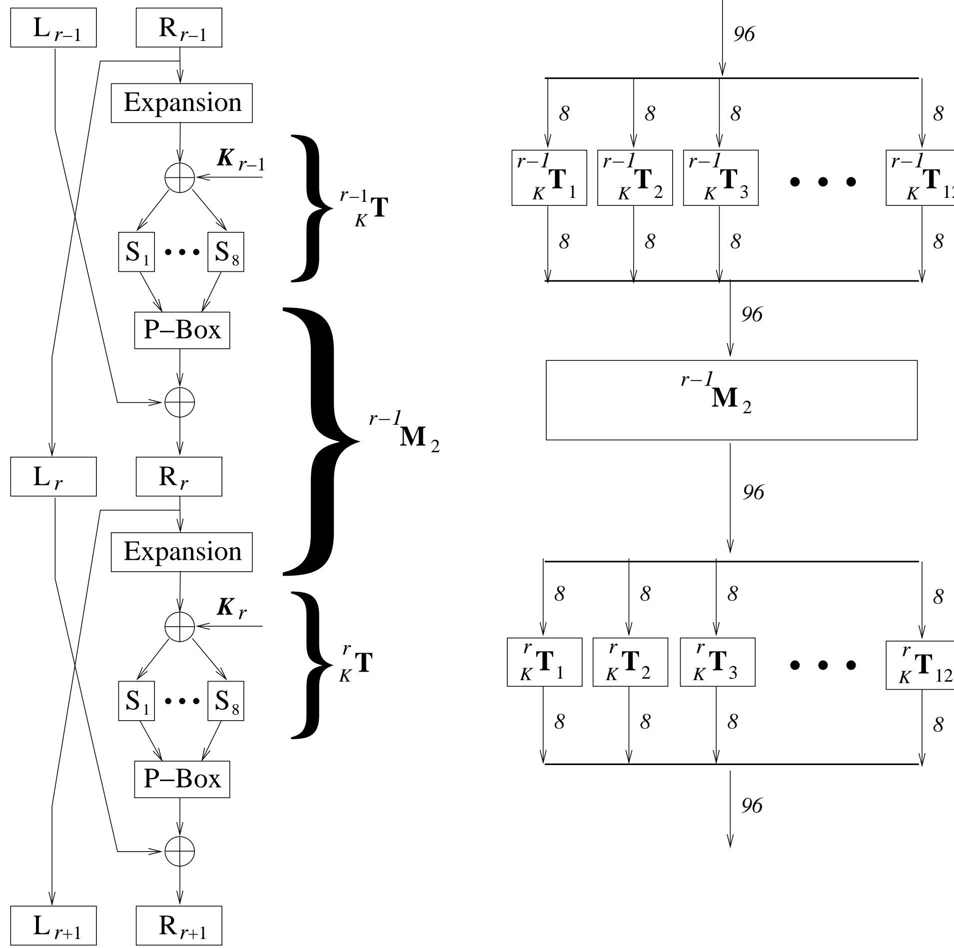

Fig. 1. Original and Modified DES

Let us therefore produce new S-Boxes identified as ${}^{r}_{K}\mathbf{S}_i$, where K is the encryption key, r is the round number, and i is the corresponding DSB number, such that, for any given input, ${}^{r}_{K}\mathbf{S}_i$ 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 of 6 input bits and 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 Figure 1(a).

Preparing the Modified DSBs for Local Security In *grey-box* (smart card) implementations of DES the DSBs have proven to be effective sites for statistical attacks. To make such attacks more difficult for our *white-box* implementation, we would like to end up with locally secure (see §3.2) S-Boxes. This means we need to replace the lossy S-Boxes with something that is bijective. We convert the lossy ${}^{r}_{K}\mathbf{S}_i$'s into ${}^{8}_{8}\mathbf{E}$ form by means of split-path encoding (see §3.1):

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

for all ${}_8\mathbf{e}$, for the fixed key K, for round $r = 1, \ldots, 16$, for S-Box number $i = 1, \ldots, 8$, where we 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 ${}_8\mathbf{e}$. This is an application of split-path encoding

The plan is that 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.

To see that each ${}^{r}_{K}\mathbf{T}_i$ is a bijection, we note that 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 the key K effectively re-orders this permutation into a new permutation. 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., it is a bijection.)

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

As converted above, each ${}^{r}_{K}\mathbf{T}_i$ carries eight bits to the next ${}^{r}\mathbf{M}_2$: 4 bits of S-Box output, 2 bits from the right side and 2 bits that we can choose to be from the left. This means eight T boxes will only carry 16 bits from the left and

16 bits from the right. This means the by-pass capacity of the ${}^{r}_{K}\mathbf{T}_i$'s is too small by 32 bits.

So we add four more S-Boxes for each round, designated as ${}^{r}_{K}\mathbf{T}_9, \ldots, {}^{r}_{K}\mathbf{T}_{12}$. 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 new S-Boxes provide the remaining 32 bits: 16 bits of right-side by-pass capacity, and 16 bits of left-side by-pass capacity.

5.2 Connecting and Encoding the New SBs to Implement DES

The overall data-flow structure of our DES implementation immediately prior to de-linearization of ATs and encoding of S-Boxes (see §3.1, §3.2), is shown in Figure 1(b). It would look just the same after de-linearization and encoding, except that each $\mathbf{M}_i$ would be replaced by a corresponding $\mathbf{M}'_i$ and each ${}^{r}_{K}\mathbf{T}_i$ would be replaced by a corresponding ${}^{r}_{K}\mathbf{T}'_i$. Except for the addition of these "$'$" characters, the figure would be identical.

Data-Flow and Algorithm Before de-linearization and encoding, each $\mathbf{M}_i$ is representable as a matrix, with forms ${}^{96}_{64}\mathbf{M}_1$, ${}^{96}_{96}\mathbf{M}_2$, and ${}^{64}_{96}\mathbf{M}_3$, respectively. (We briefly discuss the role of each one in §5.1 and in more detail below in THE TRANSFER FUNCTIONS.)

In Figure 1(b), 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 ${}^{r}_{K}\mathbf{T}_i$'s in order by i in Figure 1(b) does not indicate any ordering of their appearance in the implementation: the intervening $\mathbf{M}_2$ transformations can handle any such re-ordering.

The Transfer Functions In constructing $\mathbf{M}_1$, $\mathbf{M}_2$'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 appearing in Figure 1(a), are all 0-bits except for one or two 1-bits in each row and column. The *xor* operations ("$\oplus$" in Figure 1(a)) are similarly sparse.

Therefore, we use the second method proposed for handling sparseness in §4: 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 all involve only various combinations, compositions, simple reorganizations, and concatenations of ATs, and are therefore straightforward.

$\mathbf{M}_1$ combines the following: (1) the initial permutation of DES, (2) the **Expansion** (see Figure 1(a)), modified to deliver its output bits to the first six inputs of each ${}^{r}_{K}\mathbf{T}_i$, 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 ${}^{r}_{K}\mathbf{T}_1, \ldots, {}^{r}_{K}\mathbf{T}_8$ and 16 bits of by-pass provided at randomly chosen positions in the four 'dummies', ${}^{r}_{K}\mathbf{T}_9, \ldots, {}^{r}_{K}\mathbf{T}_{12}$, all in randomly chosen order.

$\mathbf{M}_2$ for each round combines the following: (1) the **P-box** transform (see Figure 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 **Expansion**, as in $\mathbf{M}_1$, (5) the left-side by-pass, as in $\mathbf{M}_1$.

$\mathbf{M}_3$ combines the following: (1) ignoring the inputs provided for simultaneous by-pass, (2) the left-side by-pass, as in $\mathbf{M}_1$, (3) inversion of the **Expansion**, ignoring half of each redundant bit pair, (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.

Blocking and Encoding Details We recommend using 4×4 blocking for the $\mathbf{M}_i$'s. As a result of the optimization noted in §4, this means that the entire 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 ${}^{64}_{64}\mathbf{I}$ (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 Variants

The above section completes a *naked* variant of white-box DES. The *recommended* variant applies input and output encodings to the whole DES operations. That is, we modify the scheme shown in Figure 1(b), 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 mixing bijections. Each of $\mathbf{M}_1 \circ \mathbf{M}_0$ and $\mathbf{M}_4 \circ \mathbf{M}_3$ is, of course, a single AT. When it is encoded in 4-bit blocks, it becomes 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 even 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 quite an in-obvious one. In effect, it can serve as a 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 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. Every such engine is also effectively water-marked 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. There are many variations on this theme.

5.4 Attacks on Example Implementation

The attacker cannot extract information from the ${}^{r}_{K}\mathbf{T}'_i$'s themselves: they are *locally secure* (see §3.2). 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 attacks on the recommended variants.

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 the first round (round 1), the initial input is known (the $\mathbf{M}_1$ input is not coded), and in the last round (round 16), the final output is known (the $\mathbf{M}_3$ output is not coded). Both known attacks exploit this weak point.

The Jacob Attack on the Naked Variant The recent attack in [7] is a clever DFA-like attack [3]. This particular attack induces a controlled fault by taking advantage of the unchanged data in the Feistel structure, thus bypassing much of the protection afforded by the encodings; but it requires that the input (or output) be naked (i.e., unencoded), and requires 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, we guess keys and use difference of power profiles to confirm or deny our guesses. In our Statistical Bucketing

attack, we also guess keys, but we confirm or deny our guesses by checking if buckets are disjoint.

Attacks should be focussed on the first (1st) and final (16th) 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 will only talk about 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 is affected by 6 bits of round 1 sub-key; its output bits go to different DSB s 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 DSB which is 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 2 different S-Boxes in round 2, either one will do.

We make a guess on the 6 bits of sub-key affecting ${}^{1}\mathbf{S}_i$ and run through the 64 inputs to it and construct 64 corresponding plaintexts. The plaintexts need to feed the correct bits into ${}^{1}\mathbf{S}_i$ as well as the *xor* operation involving b. For convenience, we will hold 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, we divide these 64 plaintexts into two buckets, I_0, I_1, which have the property that if our 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 our guess is correct, b will have a value of 1.

Now we take these two buckets of plaintexts and run them through the encoded implementation. Since the implementation is naked, we can easily track the data-flow to discover ${}^{2}\mathbf{T}_{z_j}$ that encodes ${}^{2}\mathbf{S}_j$. We will examine the input to ${}^{2}\mathbf{T}_{z_j}$ to confirm or deny our 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 our key guess is correct, I'_0 and I'_1 must necessarily be disjoint sets. Any overlap indicates that our guess is wrong. If no overlap occurs, our 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 our 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 DSB s). 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-sub-keys) + 256 (remaining 8 bits of key) $\approx 2^{13}$ encryptions. This attack has been implemented, and it successfully finds the key in under 10 seconds.

5.5 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. §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, affects only the sense of the output vector elements, and not how the AT redistributes 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 ${}^{n}_{n}P$ (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 below, e.g. for pre- and post-encodings.

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. However, 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: 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 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 if the answer is negative for this particular variant, we believe the general approach remains promising, due to the many variations possible from the various approaches discussed.

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 and 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 and 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 and 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 and 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 and 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 and 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/`