Attacking an Obfuscated Cipher by Injecting Faults

Matthias Jacob[1], Dan Boneh[2], and Edward Felten[1]

[1] Princeton University
{mjacob,felten}@cs.princeton.edu
[2] Stanford University
dabo@cs.stanford.edu

Abstract. We study the strength of certain obfuscation techniques used to protect software from reverse engineering and tampering. We show that some common obfuscation methods can be defeated using a fault injection attack, namely an attack where during program execution an attacker injects errors into the program environment. By observing how the program fails under certain errors the attacker can deduce the obfuscated information in the program code without having to unravel the obfuscation mechanism. We apply this technique to extract a secret key from a block cipher obfuscated using a commercial obfuscation tool and draw conclusions on preventing this weakness.

1 Introduction

In recent years the advent of mass distribution of digital content fueled the demand for tools to prevent software and digital media from illegal copying. The goal is to make it harder for a malicious person to reverse engineer or modify a given piece of software. One well known technique for preventing illegal use of digital media is watermarking for audio and video content [1] which had only limited success. Another common approach is to only distribute encrypted content (see, e.g., CSS [2], Intertrust [3], MS Windows Media Technologies [4], Adobe EBooks [5]). Users run content players on their machines and these players enforce access permissions associated with the content. In most of these systems the software player contains some secret information that enables it to decrypt the content internally. Clearly the whole point is that the user should not be able to emulate the player and decrypt the content by herself. As a result, the secret information that enables the player to decrypt the content must be hidden somehow in the player's binary code. We note that hardware solutions, where the decryption key is embedded in tamper-resistant hardware [6,7,8], have had some success [9,10], but clearly a software only solution, assuming it is secure, is superior because it is more cost efficient and easier to deploy.

This brings us to one of the main challenges facing content protection vendors: is it possible to hide a decryption key in the implementation of a block cipher (e.g. AES) in such a way that given the binary code it is hard to extract the decryption key. In other words, suppose $D^k(c)$ is an algorithm for decrypting the ciphertext c using the key k. Is it possible to modify the implementation of $D^k(c)$ so that extracting k by reverse engineering is sufficiently hard? If hiding the key in a binary is possible, it has a crucial advantage over alternative key hiding techniques: in order to decrypt content the binary needs to be executed, and efficient access control mechanisms exist in the operating system in order to prevent unauthorized execution, whereas hiding a stored key in memory is difficult [11]. Key obfuscation is a very old question already mentioned in the classic paper of Diffie and Hellman [12].

Code obfuscation is a common technique for protecting software against reverse engineering and is commonly used for hiding proprietary software systems and sensitive system components such as a cipher. Commercial obfuscation tools often work by taking as input arbitrary program source code, and they output obfuscated binary or source code that is harder to reverse engineer and thus to manipulate than the original software [13,14,15,16,17]. However, it is unclear whether obfuscation techniques can be strong enough to protect sensitive software systems such as a cipher implementation.

In this paper we investigate a commercial state-of-the-art obfuscated cryptosystem [18] that hides a secret key. An *ideal obfuscation tool* turns program code into a black-box, and therefore it is impossible to find out any properties of the program. In practice however, obfuscation tools often only *approximate the ideal case.* When obfuscating a cryptosystem the obfuscator embeds a secret key into the program code and obfuscates the code. It should be hard to figure out any properties about the key by just investigating the code. However, we show how to extract the secret key from the system in only a few cryptographic operations and come to the conclusion that current obfuscation techniques for hiding a secret key are not strong enough to resist certain attacks.

Our attack is based on differential fault analysis [19] in which an attacker injects errors into the code in order to get information about the secret key. The impact of this attack is comparable to an attack on an RSA implementation based on the Chinese Remainder Theorem that requires only one faulty RSA signature in order to extract the private key [20].

Fault attacks are a threat on tamper-resistant hardware [9], and in this paper we show that an adversary can also inject faults to extract a key from obfuscated software. Based on our experience in attacking an obfuscated cryptosystem we propose techniques for strengthening code obfuscation to make fault attacks more difficult and make a first step in understanding the limits of practical software obfuscation.

2 Attacking an Obfuscated Cipher Implementation

In this section we describe our attack on a state-of-the-art obfuscator [18] illustrated in Figure 1. We were given the obfuscated source code for both DES encryption and decryption of the iterated block cipher. Our goal was to reverse engineer the system only based on knowledge of this obfuscated source code. For the given obfuscated code the attacker does not learn more properties about the

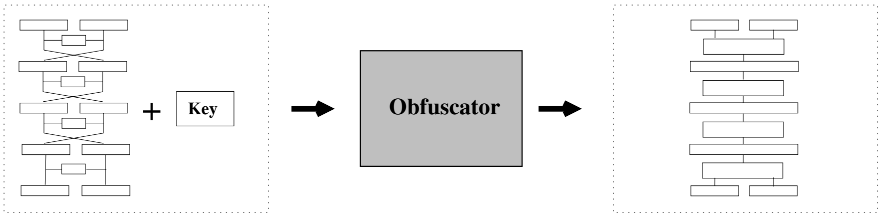

Fig. 1. Operation of the obfuscator on the round-based cipher: It transforms the key and the original source code into code that implements every round as a lookup table of precomputed values. The intermediate results after each round are encoded

program by investigating the obfuscated source code than by just disassembling the binary because most of the program is composed of lookup tables.

In this particular approach the obfuscation method hides the secret key of a round-based cipher in the code. Because a round-based cipher exposes the secret key every time it combines the key with the input data of a round, the obfuscator injects randomness and redundancies and refines the resulting boolean operations into lookup tables. Instead of executing algorithmic code, the program steps through a chain of precomputed values in lookup tables and retrieves the correct result. Therefore it is difficult to obtain any information about the single rounds by just looking at the source code or binary code, but in our attack we obtain information by observing and changing data during the encryption process.

2.1 Obfuscating an Iterated Block Cipher

The obfuscation process of the cipher implementation is shown in Figure 1. The obfuscator transforms the original source code and the key into a cipher in which the key is embedded and hidden in the rounds. The single rounds of the cipher are unrolled, but the boundaries of each round are clearly recognizable. The cipher contains n rounds π_i^k for each $i = 1, .., n$ with the key k. Including the initial permutation λ the cipher computes the function

$$E^k(M) := \left[\lambda^{-1} \cdot \pi_n^k \cdot \pi_{n-1}^k \cdot ... \cdot \pi_1^k \cdot \lambda\right](M).$$

However, interpretation of any intercepted intermediate results is difficult since the obfuscator maps the original intermediate results after each round to a new representation. This transformation is described in detail in [18].

In the following paragraphs we give an algebraic definition for the transformation into the 96-bit intermediate representation of the obfuscator in [18]. In the first step we define some basic operations. $x|_i^m$ extracts bits i through $i+m$ from a bit string. $EP(x)$ computes the DES expansion permutation.

$$x_1x_2...x_n|_i^m = x_ix_{i+1}...x_{i+m}$$
$$x_1x_2...x_n|_i = x_i$$

$$EP_i(x) = EP(x)|_{6i}^6$$

$$R'^{k}_{r} = EP(R^{k}_{r})$$
$$R'^{k}_{r,i} = EP_i(R^{k}_{r})$$

The t-box $T^{k}_{r,i}(L_r, R'^{k}_{r})$ computes the i-th DES s-box in round r for $i = 0..7$ and appends $R(L_r, R'^{k}_{r})$ which takes the first and sixth bit from $R'^{k}_{r,i}$ and appends two random bits from L_r. The bits from L_r are used to forward the left hand side information in the t-boxes, and the first and sixth bit from $R'^{k}_{r,i}$ to reconstruct R^{k}_{r} from the s-box result in order to forward it to round $r+1$ as the left hand side input.

$$T^{k}_{r,i}(L_r,\ R'^{k}_{r}) = S^{k}_{r,i}(R'^{k}_{r,i}) \mid\mid R(L_r,\ R'^{k}_{r,i})$$

$$T^{k}_{r}(L_r,\ R'^{k}_{r}) = T^{k}_{r,\gamma_r(0)}(L_r,\ R'^{k}_{r}) \mid\mid T^{k}_{r,\gamma_r(1)}(L_r,\ R'^{k}_{r}) \mid\mid ... \mid\mid T^{k}_{r,\gamma_r(11)}(L_r,\ R'^{k}_{r})$$

For $i = 8...11$ $T^{k}_{r,i}(L_r,\ R'^{k}_{r})$ outputs either random dummy values or bits from L_r.

In order to obfuscate the result γ_r permutes the order of the t-boxes on $T_r = \{T^{k}_{r,0}....T^{k}_{r,11}\}$. Additionally, ϕ_r applies a bijective non-linear encoding on 4-bit blocks x_j for $j = 1...24$ where
$\phi_r(x) = (\phi_{r,1}(x_1), \phi_{r,2}(x_2), ..., \phi_{r,24}(x_{24}))$ and $x = x_1x_2...x_{24}$. Since a single t-box consists of 8 bit outputs, two different bijective non-linear encodings belong to one t-box.

In order to do the second step the obfuscated DES implementation needs to be able to recover the original right hand side input to round r, and this gets implemented using function $\alpha^{k}_{r,i}(y)$ which takes the forwarded bits x_1 and x_2 that describe the row of the s-box.

$$\alpha^{k}_{r,i}(y, x_1, x_2) = EP_i^{-1}((S^{k}_{r,i})^{-1}(y, x_1, x_2))$$

$$L_r = L^0_r \mid\mid L^1_r \mid\mid L^2_r \mid\mid ... \mid\mid L^7_r$$
$$R'_r = R'^0_r \mid\mid R'^1_r \mid\mid R'^2_r \mid\mid ...\mid\mid R'^7_r$$

The second step then implements the function $\tau^{k}_{r,i}$ in which $\mu_r(n)$ describes the corresponding position of the bit in the output of the t-boxes, and PB is the DES p-box operation:

$$\begin{aligned}\tau^{k}_{r,i}(x)(L^i_r, R'^i_r) = {} & \underbrace{\alpha^{k}_{r,i}(x|^4_{8\gamma_r(i)},\ x|_{8\gamma_r(i)+4},\ x|_{8\gamma_r(i)+5})}_{depends\ on\ R_{r-1}\ only} \mid\mid \\ & EP_i\big[PB\ \underbrace{(x|^4_{\gamma_r(0)} \mid\mid x|^4_{\gamma_r(1)} \mid\mid ... \mid\mid x|^4_{\gamma_r(11)})}_{depends\ on\ R_{r-1}\ only} \oplus \\ & \underbrace{(x|_{\mu_r(0)} \mid\mid ... \mid\mid x|_{\mu_r(32)})}_{depends\ on\ L_{r-1}\ only}\ \big]\end{aligned}$$

$$\tau^{k}_{r}(x) = \tau^{k}_{r,0}(x) \mid\mid \tau^{k}_{r,1}(x) \mid\mid ... \mid\mid \tau^{k}_{r,11}(x)$$

ψ_r and ϕ_r are different non-linear bijective encodings on 4-bit blocks, and δ_r

$$\delta_r(L, R') = \gamma_r(\mu_r((L|0^{24}), R'))$$

$$\begin{aligned}\mu_r(x_0x_1...x_{47}, y_0...y_{47}) &= y_0...y_5x_{\mu_r^{-1}(0)}x_{\mu_r^{-1}(1)}y_6...y_{11}x_{\mu_r^{-1}(2)}x_{\mu_r^{-1}(3)}...y_{42}...y_{47}\\ &\quad x_{\mu_r^{-1}(22)}x_{\mu_r^{-1}(23)}...x_{\mu_r^{-1}(47)}\\ \gamma_r(z_0z_1...z_{95}) &= z_{\gamma_r^{-1}(0)}...z_{(\gamma_r^{-1}(0)+5)}z_6z_7...z_{\gamma_r^{-1}(11)}...z_{(\gamma_r^{-1}(11)+5)}z_{94}z_{95}\end{aligned}$$

The obfuscated t-box is

$$T_r'^k(x) = (\phi_r\, T_r^k\, \psi_{r-1}^{-1})(x).$$

Hence the transformed function is:

$$\begin{aligned}E^k(x) = \big[&(\lambda^{-1}\delta_n^{-1}\psi_n^{-1}) \cdot \big((\psi_n\delta_n\tau_n^k\phi_n^{-1}) \cdot (\phi_nT_n^k\psi_{n-1}^{-1})\big) \cdot ... \cdot\\ &\big((\psi_1\delta_1\tau_1^k\phi_1^{-1}) \cdot (\phi_1T_1^k\psi_0^{-1}) \cdot (\psi_0\delta_0\beta\lambda)\big)\big](x)\end{aligned}$$

with

$$\beta(L, R) = L\,||\,EP(R)$$

By setting

$$\tau_r'^k = \begin{cases}\psi_0\;\delta_0\;\beta\;\lambda & r = 0\\ \psi_r\;\delta_r\;\tau_r^k\;\phi_r^{-1} & r = 1, .., n\\ \lambda^{-1}\;\delta_n^{-1}\;\psi_n^{-1} & r = n+1\end{cases}$$

the resulting encryption operation is

$$E^k(x) = \left[\tau_{n+1}'^k \cdot \left(\tau_n'^k \cdot T_n'^k\right) \cdot ... \cdot \left(\tau_1'^k \cdot T_1'^k\right) \cdot \tau_0'^k\right](x)$$

Every component $\tau_i'^k$ and $T_i'^k$ is implemented within a separate lookup table. For convenience set

$$\tau_r''^k = \begin{cases}\tau_r'^k & r = 0,\ r = n+1\\ \tau_r'^k \cdot T_r'^k & r = 1, .., n\end{cases}$$

and obtain

$$E^k(x) = \left[\tau_{n+1}''^k \cdot \tau_n''^k \cdot ... \cdot \tau_0''^k\right](x)$$

Figure 2 shows the deobfuscation problem. Given one DES round and the obfuscated intermediate representations an attacker wants to find out the intermediate representation which is encoded by the unknown function σ_r. This σ_r is the inverse of the encoded input to the t-box (by ψ), the permutation of the t-boxes γ_r, and the random distribution of the left hand side μ_r:

$$\sigma_r(L_r, R_r) = \psi_r(\delta_r(L_r, EP(R_r)))$$

$E^k(x)$ contains the key k implicitly in $\tau_r''^k$ (in [18] $\tau_0'^k$ corresponds to M_1, $\tau_{n+1}'^k$ to M_3 and all other $\tau_r'^k$ to M_2). In other words, the implementation of $\tau_r''^k$ hides the decomposition into its components σ_{r-1}^{-1}, π_r^k, and σ_r. Hence, recovering the key boils down to the problem of extracting π_r^k out of τ_r''. In any further explanations we remove λ from any computation since it does not play any role in the attack and can be easily inverted. Therefore $\tau_0''^k = \psi_0$ and $\tau_{n+1}''^k = \psi_n$.

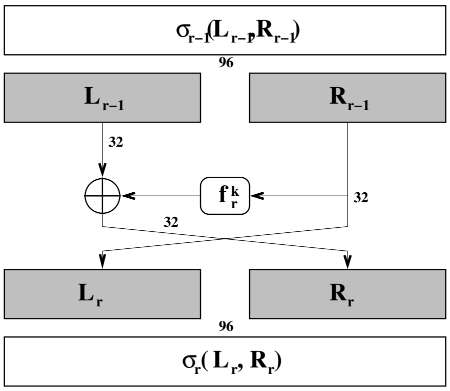

Fig. 2. Round r with the function f_r^k hiding the key k. σ_r is the intermediate representation and L_r and R_r are the left hand and the right hand side of the intermediate result respectively. The rounds π_r^k correspond to $\pi_r^k = f_r^k(R_{r-1} \oplus L_{r-1}, R_{r-1})$ for $r = 1..n$

2.2 Attacking an Obfuscated Iterated Block Cipher

In an example for a naive approach for attacking the obfuscated cipher an adversary encrypts some arbitrary plaintext and intercepts intermediate results to obtain $\sigma_r(L_r, R_r)$. The adversary starts the attack by encrypting plaintexts p that have one single bit set, and afterward examines the obfuscated intermediate results after the first round π_1^k during encryption. By heuristically computing the differences between $(\tau_1''\tau_0'')(p)$ and $(\tau_1''\tau_0'')(0)$ for $p \neq 0$ we find that $(\tau_1''\tau_0'')(p)$ changes deterministically for all p that have one bit set in the left hand side of the plaintext L_0 due to the construction of the t-boxes. However, since the adversary is not able to compute σ_1^{-1} in order to retrieve R_1 any knowledge of R_0 and L_0 is meaningless if she wants to extract the key. An attack that works on the first round by recovering σ_1^{-1} of the cipher is the statistical bucketing attack [18]. This attack exploits some properties of the DES s-boxes and requires about 2^{13} encryptions. In contrast our attack works for any round-based block cipher and requires only dozens of encryptions.

We now describe how we use a simplified differential cryptanalysis called differential fault analysis [19] to recover the key in a few operations. In this attack an adversary flips bits in the input to the last round function f_n^k and computes the different outputs to find out the round function f_n^k of the last round n. When injecting single bit faults into the last round using chosen ciphertexts only dozens of cryptographic operations are necessary in order to find f_n^k. The implementation of this attack requires less information about the intermediate representation than the naive attack since an attacker only needs to flip a single bit in the obfuscated intermediate representation, and it is not necessary to figure out any inverse mappings σ_r^{-1}. Also, this attack is independent from the DES structure and can be applied to any round-based block cipher. We try to apply deterministic changes to $\sigma_{n-1}(L_{n-1}, R_{n-1})$, the state going into the last round, and then run the last round operation.

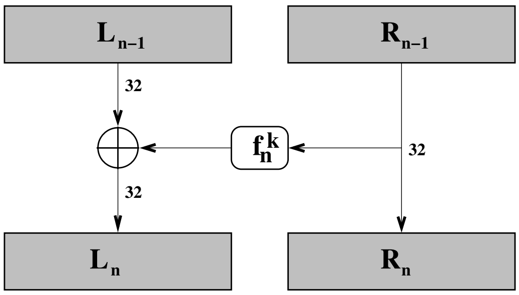

Fig. 3. Last round with the round function f_n^k. In the last round the right hand side and the left hand side of the output are usually not crossed over

Figure 3 shows the last round of the cipher. An attacker knows $R_n = R_{n-1}$ from the ciphertext which is also the input to the round function of the last round. In addition an attacker can modify R_{n-1} even if the mapping of σ_{n-1} is unknown by changing R_n in the ciphertext, decrypting the ciphertext, and encrypting the resulting plaintext afterward. Therefore we have two preconditions for the attack: First, both encryption and decryption operations need to be available, and second, the attacker needs to be able to modify the ciphertext arbitrarily. Using this technique we can find out the positions of $\mu_r(i)$ for $i = 0...32$ which describe the bits for the left-hand side. From the definition of $T_{r,i}^k$ it is clear, that if the attacker keeps the right-hand side input constant, the observed changes in the input to the t-boxes uniquely refer to changes in the left-hand side of the input. The attacker is not able to set L_{n-1} to 0 since she would need to know the round function and hence the key. Therefore, $R_n = 0$ and $L_{n-1} = f_n^k(0) \oplus L_n$.

Now the attacker builds a table of

$$\Delta(c) := \sigma_{n-1}(c, 0) \oplus \sigma_{n-1}(0, 0)$$

for $c = 1...2^{32}$.

Since σ_r contains the unknown non-linear bijection δ_{r-1} it is not possible to build a linear operator in Δ. However, using the table the attacker can always reconstruct the left-hand side of the input in the scenario where the right-hand side is 0. Furthermore, different bits of the left-hand side L_{n-1} can correspond to the same t-box, and in this case the encoding depends on two bits. Therefore, in the first part the attacker tests which bits correspond to the same t-box and then tries all possible bit combinations into this t-box. In this way the attacker gets all possible values for σ_r induced by the left-hand side L_{n-1}. Determining the original value $L_{n-1} \oplus f_n^k(0)$ given the intermediate representation is just a table lookup.

The idea now is to inject faults into the input to the s-box and observe the output. Unfortunately, the attacker does not know how the right-hand side gets encoded in σ_r. In order to get around this problem the attacker feeds a value x into R_{n-1} that is different from 0 and then resets L_{n-1} to 0. Finally, L_n contains $f_n^k(x) \oplus f_n^k(0)$, and the attacker can extract the key for the last round using differential cryptanalysis. Getting the DES key from the round key requires a 2^8 brute-force search.

The problem is that if the right hand side R_{n-1} changes to some value $\neq 0$ the t-box inputs collide with the 16 bits of the left-hand side L_{n-1}. Therefore it is not possible to decode the left-hand side L_{n-1} uniquely since complete new values might show up in the t-boxes that are taking as input bits from the left-hand side.

However, if the attacker sets only one bit in R_{n-1} at most two different t-box outputs are affected, and hence the attacker can simply count the occurrences of the encoded 4-bit values at a certain position in σ_r.

We describe the algorithm for the attack when the specification of the round function is known. We will explain at the end of the algorithm how the algorithm needs to be changed to attack an unknown round function. For convenience we use $D^k(c)$ to describe the decryption of ciphertext c using key k, and $E_i^k(p) = (L_i, R_i)$ to describe iteration of plaintext p for i rounds in the encryption operation using key k. $s^n(k) = s_n^1(k)|...|s_n^8(k)$ is the key schedule for key k in round n, m is the size of the input word, and the sboxes $sb_n(x) = sb_n^1(x_1)|...|sb_n^8(x_8)$:

$$f_n^k(x_1|...|x_8) := sb_n^1(x_1 \oplus s_n^1(k))|...|sb_n^8(x_8 \oplus s_n^8(k))$$

In our simplified model the in- and outputs of the s-box have the same size, and the system computes the xor of the key and the input to the s-box. The algorithm consists of 3 basic operations: A *Set* operation changes any arbitrary variable. When we do a *Compute* we execute an operation in the iterated block cipher. This can be encryption, decryption, or just a single round of the cipher. *Derive* computes values on known variables without executing the cipher. Figure 4 illustrates the single steps of the algorithm.

Our attack algorithm works as follows:

1. **Initialization:** (Figure 4 top left)

```
    Set $L_n := 0$, $R_n := 0$
    Compute $\sigma_{n-1}(L_{n-1}, R_{n-1}) = E_{n-1}^k(D^k(L_n, R_n))$
    Result: $L_{n-1} = f_n^k(0)$, $R_{n-1} = 0$
    Derive $\Omega = \sigma_{n-1}(L_{n-1}, R_{n-1}) = \sigma_{n-1}(f_n^k(0), 0)$
```

2. **Reconstruct $\Delta(x)$:** (Figure 4 top right)

```
    For $j = 0$ to $23$:
        Set $m(j) := 0$
    For $i = 0$ to $31$:
        Set $L_n := 2^i$, $R_n := 0$
        Compute $\sigma_{n-1}(L_{n-1}, R_{n-1}) = E_{n-1}^k(D^k(L_n, R_n))$
        Set $\Delta(L_n) := \sigma_{n-1}(L_{n-1}, R_{n-1}) \oplus \Omega$
        For $j = 0$ to $23$:
            If $\left(\Delta(L_n)|_{4j}^{4} \neq 0\right)$
              Set $b[j][m(j)] := i$
              Set $m(j) := m(j) + 1$
```

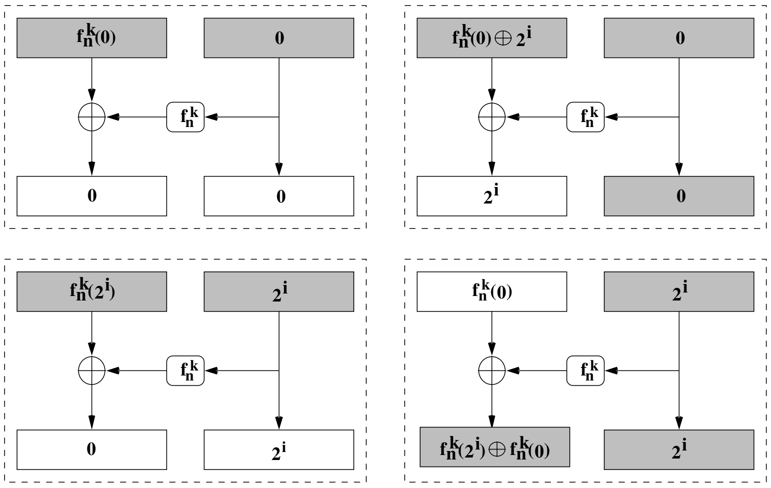

Fig. 4. Attacking the last round of the iterated block cipher. Boxes having a white background indicate that the attacker changed values. The picture on the top left shows the initialization of the algorithm (step 1). Afterward, on the top right we change L_n to 2^i in order to reconstruct $\psi_{n-1}(x)$ (step 2). In the bottom left we set 2^i to be input to the round function. The fault injection takes place on the bottom right (step 3): We reset L_{n-1} to $f_n^k(0)$ and obtain the difference $f_n^k(2^i) \oplus f_n^k(0)$ in L_n

```
For $j = 0$ to $23$:
    For $l = 0$ to $2^{m(j)} - 1$:
        Set $e := 0$
        For $k = 0$ to $m(j)$:
            If $(((l >> k) \& 1) = 1)$
                Set $e := e + 2^{b[j][k]}$
        Set $L_n := e$, $R_n := 0$
        Compute $\sigma_{n-1}(L_{n-1}, R_{n-1}) = E_{n-1}^k(D^k(L_n, R_n))$
        Set $\Delta(L_n) := \sigma_{n-1}(L_{n-1}, R_{n-1}) \oplus \Omega$
```

3. **Reset L_{n-1} to $f_n^k(0)$:** (Figure 4 bottom left)

```
For $i = 0$ to $31$:
    Set $L_n := 0$, $R_n := 2^i$
    Compute $\sigma_{n-1}(L_{n-1}, R_{n-1}) = E_{n-1}^k(D^k(L_n, R_n))$,
    Result: $L_{n-1} = f_n^k(2^i)$, $R_{n-1} = 2^i$
    Derive $w := \sigma_{n-1}(L_{n-1}, R_{n-1}) \oplus \Omega = \sigma_{n-1}(f_n^k(2^i), 2^i) \oplus \sigma_{n-1}(0, 0)$
    For $x$ in $\Delta^{-1}$
        For $j = 0$ to $23$
            If $\left(\Delta(x)|_{4j}^{4} = w|_{4j}^{4}\right)$
                $w|_{4j}^{4} := 0$
    Compute $(L'_n, R'_n) = (\tau''_n \tau''_{n+1})(w) = (\sigma_{n-1}^{-1} \pi_n^k)(w)$
    Result: $L'_n \approx f_n^k(2^i) \oplus f_n^k(0)$, $R'_n \approx 2^i$
```

4. **Do differential cryptanalysis to extract the key for the round function f_n^k:**

```
$l^s = L'_n|^4_{4(s-1)}$, $r^s = EP(R'_n)|^6_{6(s-1)}$
For $s = 1$ to 8:
    $d^s = 0$
For $s = 1$ to 8:
    For $i = 0$ to 31:
        Compute $c^s[i]$: $sb_n^s(r^s[i] \oplus c^s[i]) = l^s[i]$
        Compute $d^s[i] = d^s[i] + 1$
    Set $\tilde{c}^s := c^s[\max_{i=1}^m d^s[i]]$
```

5. **Reconstruct the original key:**

```
$k := \tilde{c}^1|\tilde{c}^2|...|\tilde{c}^8$
Compute $s_n(k)^{-1}$ to retrieve original key
brute-force search on the remaining bits of the key.
```

Step 2 of the algorithm reconstructs $\Delta(x)$, in step 3 we inject the fault by resetting L_{n-1} to $f_n^k(0)$ and computing $L_n = f_n^k(R_n) \oplus f_n^k(0)$. In steps 4 and 5 we compute the key given a round function f_n^k by concatenating the components going into the s-boxes, inverting the key schedule, and running a brute-force search on the remaining key bits.

If the key schedule $s_n(k)$ for round n is unknown, we cannot do step 5 to get the key out. In this case we have to compute the key for round n and then use this key to attack round $n-1$ until we extract all round keys. If the round function f_i^k is unknown, we can first try out different known round functions (e.g. Skipjack, Blowfish, DES etc) for f_i^k. If none of them works, we have to do cryptanalysis to recover the s-boxes from scratch. We make the basic assumption that the round function is based on an s-box with fixed inputs.

This attack is fully automated and can be run without any knowledge of the system. Given the plaintext length as $2n$ and the length of the intermediate representation as $4m$ the attack in steps 1-5 extracts the key in $O(\max(m, n))$ cryptographic operations, and therefore undermines the security of the obfuscation system.

2.3 Summarizing the Attack

We exploit two weaknesses in this attack: First, the boundaries of the rounds are identifiable and protection of intermediate results against tampering is not strong enough. This means that a) hiding the rounds can strengthen the implementation and b) data needs to be safe against leaking of information during execution.

In this attack we show that faults in ciphers are a cheap and efficient technique to extract a secret key from an obfuscated cipher implementation in software. Our attack on obfuscated cipher implementations in software requires only a few cryptographic operations, and therefore an adversary can run the attack on any inexpensive hardware.

We had to modify the original algorithm for differential fault analysis [19] in several steps. The main difference is that it is not possible to inject random faults since the intermediate representation is obfuscated and has multiple points of failure. However, it is still possible to find out a sufficient amount of information about the obfuscated intermediate representation that make it possible for an attacker to inject faults.

In the underlying attack model it is the goal to decrypt some media stream on different machines at the same time. To do this we assume that copy protection of the decryption system is sufficiently strong, and therefore an attacker has to extract the secret key. In the current implementation our attack requires that a decryption system colludes with an encryption system, but actually an attacker only needs to obtain plaintexts for $2m$ chosen plaintexts and the decryption system. Or, since the system is a symmetric block cipher, we run the attack on the encryption system and need $2m$ chosen ciphertexts from the decryption operation. Furthermore, it is an open question how difficult it is to turn an obfuscated decryption system into an encryption system. In this case having the decryption system is sufficient for the attack.

In the recommended variant the system executes the encryption operation $E'(x) = (f^{-1}Eg)(x)$ and the decryption operation $D'(x) = (g^{-1}Df)(x)$ where f and g are non-linear bijective encodings. The current attack is now impossible, but the disadvantage is that given a ciphertext it is only possible to decrypt when f, g, and the key k are known, or the obfuscated decryption program is being used. It is not implementing DES anymore.

It is crucial to fix the weaknesses in the system or implement other techniques to prevent any common attacks that recover the secret key. In the following sections we explore what we can do about the weaknesses and investigate how to strengthen obfuscation techniques against common attacks.

3 Theoretical Considerations

The weaknesses in this attack are specific to the implementation of the obfuscated cipher. We were able to use specific properties of the DES cipher and the obfuscation method in order to extract the secret key. However, theoretical considerations do not necessarily limit any stronger obfuscation techniques. Here we give a simple argument why the general problem of retrieving embedded data from a circuit is NP-hard, and therefore no efficient general deobfuscator exists for this problem.

In MATCH-FIXED-INPUT we are given two circuits, one of which has additional input k. It is the goal to find a k such that the two circuits are equivalent.

Definition: MATCH-FIXED-INPUT: Given circuits two $C(x,k)$ and $C'(x)$ where $x \in \{0,1\}^n$ and $k \in \{0,1\}^c$ where $c \in N$ is constant, find $k' \in \{0,1\}^c$ such that $\forall x : C(x,k') = C(x)$.

Theorem: MATCH-FIXED-INPUT is NP-hard.

Proof: We reduce SAT to MATCH-FIXED-INPUT which is almost trivial. In order to test satisfiability of circuit $D(x)$, set $C(x,k) = D(k)$ and $C'(x) = true$, and run MATCH-FIXED-INPUT. If MATCH-FIXED-INPUT returns a k' such that $C(x,k') = C'(x)$, then according to the definition there exists an x such that $D(x) = true$. If MATCH-FIXED-INPUT does not return a k', then for all x $D(x) = false$. Hence, we reduce SAT to MATCH-FIXED-INPUT. □

For practical purposes, however, this theoretical observation is not much of a relevance since the problem is hard in the worst case but can still be easy for practical purposes. On the average the problem MATCH-FIXED-INPUT is NP-hard, but in several cases heuristic methods can extract the fixed input as in the example of this obfuscated DES cipher.

4 Strengthening Obfuscation

In this section we briefly discuss various mechanisms for defending against our attack using software faults. We first describe some common attacker goals when attacking obfuscated code:

- **Hide data in the program:** The attacker wants to find out certain data values. This case subdivides into the possibility of tracing values during runtime and discovering static values in the code.
- **Protect the program from controlled manipulation:** In this case the attacker wants to force the program to behave in a certain way, e.g. to remove copy protection mechanisms or to cause damage on a system.
- **Hide algorithms of the program:** According to Kerckhoff's principle cryptographic algorithms are usually public, but in some cases it is useful to hide certain properties by which an attacker can recognize the algorithm, i.e. distinguish for example between AES, IDEA or Blowfish [21,22,23].

Often when obfuscating a cipher, commercial tools first encode the plaintext using some hidden encoding function, then run the cipher, and finally decode the ciphertext using some other hidden decoding function. More precisely, the encryption process looks like $E'_k(x) = (F \cdot E_k \cdot G^{-1})(x)$ where E_k is the original DES encryption [18]. Note that F and G must be one-to-one functions so that decryption is possible. The decryption process is similar: $D'_k(x) = (G \cdot D_k \cdot F^{-1})(x)$. This pre- and post-encoding makes chosen ciphertext attacks more difficult since an adversary first needs to recover G. As a result, these encoding makes our fault attack harder to mount. One can still potentially attack the system by using a fault attack against inners levels of the Feistel cipher.

4.1 Defending against a Fault-Based Attack

We mention a few mechanisms for protecting obfuscated systems from a fault attack. One approach is to protect all intermediate results using checksums. These checksums are frequently checked by the obfuscated code. We refer to this approach as *local checking.* Clearly the code for checking these checksums must be hidden in the total program code so that an attacker cannot disable these checkers. One approach for using checksums to ensure code integrity is explained in [24]. In this approach we compute checksums for parts of the program and verify them during program execution. In the extreme we verify a checksum for every single instruction and every data element.

Another approach for checking the computation of obfuscated code is to use *global checking.* The idea is to execute the obfuscated program k times (e.g. $k = 3$) by interleaving the k executions. At the end of the computation the code verifies that all k executions resulted in the same value. As before, the checker must be obfuscated in the code so that it cannot be targeted by the attacker. This global checking approach makes our attack harder since the attacker now has to modify internal data consistently in all k executions of the code.

The problem with the checking approaches is the vulnerability of the checker since it is unprotected against any tampering attack. One approach to make the checker more robust is to obfuscate it and have it verify its own integrity repeatedly while it is checking the program. This variant reduces the maximum time interval an attacker has to run the modified program. In any case the attacker needs to modify to system at more than one place. We note that if the integrity check fails the program should not stop execution immediately since this will tell an attacker where the checker is.

Another approach for making the fault attack more difficult is to diversify the obfuscation mechanism. In other words, each user gets a version of the code that is obfuscated differently (e.g. by using different encoding functions). In diversification we add randomness to the obfuscation methods, and therefore two obfuscated programs are always different after obfuscation. Especially vulnerable places in a program such as the intermediate results of the iterated round-based cipher need to be diversified.

5 Related Work

Informally tamper-resistance of a software implementation measures to what extent the implementation resists arbitrary or deliberate modifications. For example, an implementation can be protected from removing a copy protection mechanism. Thus, obfuscation is a common technique for improving tamper-resistance. Barak et al. [25] give a formal definition of obfuscation using a black-box approach which is the ideal case. They show that in their model, that obfuscation is not possible.

Encrypting the executable binary [26] is the most common approach for hiding code. In binary encryption the program is encrypted and decrypts itself during runtime. The problem is that the program is available in the clear at some point before it gets executed on the processor, and it can be intercepted. Furthermore, the system needs to hide the decryption key, and that reduces recursively to the key obfuscation problem itself.

A common approach for obfuscation is to obstruct common static program analysis [27,28,29]. The main technique for doing this is to insert of additional code that creates pointer aliasing situations. Applying static program analysis to analyze a program containing possible pointer aliasing turns out to be NP-hard [30]. This obfuscation technique only protects against attacks by static program analysis. It is still possible to do dynamic attacks with a debugger or any type of tampering.

The goal of obfuscation is to hide as many program properties as possible. The principle of improving tamper-resistance by obfuscation is that if an attacker cannot find the location for manipulating a value, it is impossible to change this value. In addition an obfuscator can eliminate single points of failure. On the other hand obfuscation never protects against existential modification.

Collberg et al define some metrics for obfuscation in [28]. They classify obfuscation schemes by the confusion of a human reader ("potency"), the successfulness of automatic deobfuscation ("resilience"), the time/space overhead ("cost"), and the blending of obfuscated code with original code ("stealth"). But obfuscation of a secret key requires stronger properties of obfuscation, since any definition of tamper-resistance is missing. A program that is a good obfuscator in these metrics can still have a single point of failure, and therefore it does not protect the program against fault attacks.

Tamper-resistance can also be improved by techniques other than obfuscation. We already mentioned self-checking of code as one possibility [24,31,16]. Protection by software guards is another technique to prevent tampering [32]. Software guards are security modules that implement different tasks of a program and thus eliminate single points of failure. In addition a program can implement anti-debugging techniques in order to prevent tampering with a debugger [33]. Anti-debugging inserts instructions into a program or changes properties in order to confuse a debugger. For example a program can arbitrarily set break points or misalign code. Furthermore, virtual software processors are are a technique for making tampering difficult [13]. Virtual software processors run the original program on a software processor, and in order to reverse engineer the original program, an attacker needs to compromise any protection mechanism of the virtual software processor as well.

Goldreich and Ostrovsky show in [34] that software protection against eavesdropping can be reduced to *oblivious simulation of RAMs.* In their definition a RAM is oblivious if two different inputs with the same running time create equivalent sequences of memory accesses. Oblivious RAM protects against any passive attack and therefore strengthens an obfuscator because it is impossible to find out the memory locations a program accesses. However, it does not protect against the fault injection attack.

Current hardware dongles are based on the idea of oblivious RAM, since the code implementing the license check sits on the dongle.

6 Open Problems

In other areas of information hiding techniques, such as watermarking, benchmark programs are available to measure the strength of a technique to hide information. For example, StirMarks [35] uses a variety of different generic attacks on a watermarked image to make the watermark illegible. It is an open problem to build such a benchmark for code obfuscation and tamper resistance tools. Such a benchmark would take as input some tamper resistant code and attempt to break the tamper resistance. Currently no such benchmark exists and there is no clear model for building such a benchmark.

One of the main open problems in code obfuscation is to come up with a model for obfuscation that can be realized in practice. [25] defines obfuscation using a black-box model that hides all properties of a program. They show that it is not possible to achieve obfuscation in that model. For practical purposes a black box model might not always be necessary. In the example of the obfuscated DES cipher in this paper we only need to make sure that it is impossible to get information about the secret key. The open research problem is to find the most general definition for obfuscation that can be realized in practice.

7 Conclusion

Code obfuscation provides some protection against attackers who want to find out secret data or properties of a program, but it is not sufficient as a standalone system. In this study we evaluate the usability of obfuscation when hiding a secret key in an iterated round-based software cipher. We find weaknesses in a commercial state-of-the-art obfuscator. Our attack enables automated extraction of the secret key from the obfuscated program code. We discuss a few methods for defending against these attacks.

References

1. Craver, S.A., Wu, M., Liu, B., Stubblefield, A., Swartzlander, B., Wallach, D.S., Dean, D., Felten, E.W.: Reading between the lines: Lessons from the SDMI challenge. In: Proceedings of the 10th USENIX Security Symposium. (2001)
2. CSS: `http://www.dvdcca.org/css` (2002)
3. Intertrust: `http://www.intertrust.com` (2002)
4. Microsoft Windows Media Technologies: `http://www.microsoft.com/windows/windowsmedia` (2002)
5. Adobe EBooks: `http://www.adobe.com/epaper/ebooks` (2002)
6. Abraham, D.G., Dolan, G.M., Double, G.P., Stevens, J.V.: Transaction Security System. IBM Systems Journal **30** (1991) 206–229
7. Dallas Semiconductor: Soft Microcontroller Data Book. (1993)
8. Trusted Computing Platform Alliance: `http://www.trustedpc.org` (2002)
9. Anderson, R., Kuhn, M.: Low cost attacks on tamper resistant devices. In: Proceedings of the 5th International Security Protocols Conference. (1997) 125–136
10. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. Lecture Notes in Computer Science **1666** (1999) 388–397
11. Shamir, A., van Someren, N.: Playing "hide and seek" with stored keys. Lecture Notes in Computer Science **1648** (1999) 118–124
12. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **IT-22** (1976) 644–654

13. Microsoft Corporation: World Intellectual Property Organization, WO 02/01327 A2 (2002)
14. Cloakware Corporation: World Intellectual Property Organization, WO 00/77596 A1 (2000)
15. Intertrust Corporation: US Patent Office, US 6,157,721 (2000)
16. Intel Corporation: US Patent Office, US 6,205,550 (2000)
17. RetroGuard Java Obfuscator: `http://www.retrologic.com` (2002)
18. Chow, S., Johnson, H., van Oorschot, P.C., Eisen, P.: A White-Box DES Implementation for DRM Applications. In: Proceedings of Workshop on Digital Rights Management 2002. (2002)
19. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. Lecture Notes in Computer Science **1294** (1997) 513–525
20. Boneh, D., DeMillo, R.A., J.Lipton, R.: On the importance of checking cryptographic protocols for faults. Lecture Notes in Computer Science **1233** (1997) 37–51
21. Schneier, B.: Applied Cryptography. Wiley (1994)
22. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC Press (1997)
23. Daemen, J., Rijmen, V.: Rijndael for AES. In NIST, ed.: The Third Advanced Encryption Standard Candidate Conference, National Institute for Standards and Technology (2000) 343–347
24. Aucsmith, D.: Tamper-resistant software: An implementation. Lecture Notes in Computer Science **1174** (1996) 317–333
25. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. Lecture Notes in Computer Science **2139** (2001) 1–18
26. grugq, scut: Armouring the ELF: Binary encryption on the UNIX platform. Phrack Inc. **58** (2001)
27. Wang, C., Davidson, J., Hill, J., Knight, J.: Protection of software-based survivability mechanisms. Proceedings of the 2001 Dependable Systems and Networks (DSN'01) (2001)
28. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: The 25th Symposium on Principles of Programming Languages (POPL '98), Association for Computing Machinery (1998) 184–196
29. Steensgaard, B.: Points-to analysis in almost linear time. In: The 23th Symposium on Principles of Programming Languages (POPL '96), Association for Computing Machinery (1996) 32–41
30. Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems **1** (1992) 323–337
31. Horne, B., Matheson, L., Sheehan, C., Tarjan, R.E.: Dynamic self-checking techniques for improved tamper-resistance. Lecture Notes in Computer Science **2320** (2001) 141–159
32. Chang, H., Atallah, M.J.: Protecting software code by guards. Lecture Notes in Computer Science **2320** (2001) 160–175
33. Cesare, S.: Linux anti-debugging techniques (fooling the debugger). Security Focus (2000)
34. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the Association for Computing Machinery **43** (1996) 431–473
35. Petitcolas, F.A.P., Anderson, R.J., Kuhn, M.G.: Attacks on copyright marking systems. Lecture Notes in Computer Science **1525** (1998) 219–239