Chapter 13
Basics of Side-Channel Analysis

Marc Joye

13.1 Introduction

Classical cryptography considers attack scenarios of adversaries getting black box access to a cryptosystem, namely to its inputs and outputs. For example, in a chosen ciphertext attack, an adversary can submit ciphertexts of her choice to a decryption oracle and receives in return the corresponding plaintexts. In real life, however, an adversary may be more powerful. For example, an adversary may in addition monitor the execution of the cryptosystem under attack and collect some side-channel information, such as the execution time or the power consumption. The idea behind side-channel analysis is to infer some secret data from this extra information.

This chapter presents several applications of side-channel analysis using different types of side-channel leakage. The primary goal is to explain the basic principles of side-channel analysis through concrete examples. Simple countermeasures to prevent side-channel leakage are also discussed. More sophisticated methods and advanced techniques are presented in the next chapters.

13.2 Timing Analysis

The concept of using *side-channel information* as a means to attack cryptographic schemes first appeared in a seminal paper by Kocher [8]. In this paper, Kocher exploits differences in computation times to break certain implementations of RSA and of discrete logarithm-based cryptosystems.

In this section, we describe two timing attacks. We show how an attacker able to make timing measurements with some accuracy can recover secret data. The first attack applies to a password verification routine. Following [5], the second attack is against an implementation of an RSA signature scheme [2, 13].

Thomson R&D France
e-mail: marc.joye@thomson.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_13,

13.2.1 Attack on a Password Verification

In order to restrict access to data to legitimate users, certain applications make use of passwords. Think for example of a file that requires a valid password to be opened. If the password is an 8-byte value, a brute-force attack would require $256^8 = 2^{64}$ trials!

Now assume that the password verification routine is implemented as described in Algorithm 4.[1] $\widetilde{P}$ denotes the 8-byte password proposed by the user and P denotes the correct password. The routine returns "`true`" if the entered password is valid and "`false`" if it is not.

Algorithm 4 Password verification.

Input: $\widetilde{P} = (\widetilde{P}[0], \ldots, \widetilde{P}[7])$ (and $P = (P[0], \ldots, P[7])$)
Output: '`true`' or '`false`'
1: **for** $j = 0$ to 7 **do**
2: **if** $(\widetilde{P}[j] \neq P[j])$ **then return** '`false`'
3: **end for**
4: **return** '`true`'

This implementation is insecure against an attacker measuring the time taken by the routine to return the status, "`true`" or "`false`". We can see that the verification of a valid password is longer since the routine returns "`false`" as soon as two bytes differ. Based on this, an attacker can mount the following attack.

1. For $0 \leq n \leq 255$, the attacker proposes the 256 passwords $\widetilde{P}^{(n)} = (n, 0, 0, 0, 0, 0, 0, 0)$ and measures the corresponding running time, $\tau[n]$.
2. Next, the attacker computes the maximum running time

$$\tau[n_0] := \max_{0 \leq n \leq 255} \tau[n] \ .$$

   The correct value for the first byte of P, $P[0]$, is given by n_0.
3. Once $P[0]$ is known, the attacker reiterates the attack with

$$\widetilde{P}^{(n)} = (P[0], n, 0, 0, 0, 0, 0, 0),$$

   and so on until the whole value of P is recovered.

This timing attack is very efficient. It requires at most $256 \cdot 8 = 2048$ calls to the verification routine to completely recover an 8-byte password.

To prevent the attack, one may think of adding a random delay before returning the status. This is however not sufficient since the attacker would still be able to mount a similar attack. In Step 1, the attacker would propose t times the password

[1] Note that this is basically the way the C function `memcmp` compares two memory regions.

$\widetilde{P}^{(n)} = (n,0,0,0,0,0,0,0)$ and measure the corresponding average running time over the t executions, $\bar{\tau}[n]$, for $0 \leq n \leq 255$. The attack then proceeds as before. Its complexity increases by a factor of t.

The correct way to prevent the attack is to make a *constant time* implementation irrespective of the input data.

13.2.2 Attack on an RSA Signature Scheme

Like a handwritten signature, the purpose of a digital signature is to guarantee the authenticity and the integrity of a message. There are two keys: a signing key which is private and a verification key which is public.

A common practice to produce a digital signature of a message m with RSA relies on the "hash-and-sign" paradigm. Message m is first hashed into $\mu(m)$ and the result is then raised to the dth power modulo N, $S = \mu(m)^d \bmod N$, where d denotes the private RSA key. The public verification key is $\{e,N\}$ where $ed \equiv 1 \pmod{\phi(N)}$ and ϕ is Euler's totient function. The validity of a putative signature S of message m is verified by checking whether $S^e \equiv \mu(m) \pmod N$.

There are various ways to evaluate a modular exponentiation. We consider below the square-and-multiply algorithm. At iteration j of the main loop, there is a modular squaring and when bit d_j of d is equal to 1 there is also a modular multiplication.

Algorithm 5 Computation of an RSA signature.

Input: $m, N, d = (d_{k-1}, \ldots, d_0)_2$, and $\mu : \{0,1\}^* \to \mathbb{Z}/N\mathbb{Z}$
Output: $S = \mu(m)^d \pmod N$
1: $R_0 \leftarrow 1; R_1 \leftarrow \mu(m)$
2: **for** $j = k-1$ downto 0 **do**
3: $\quad R_0 \leftarrow {R_0}^2 \pmod N$
4: $\quad$ **if** $(d_j = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_1 \pmod N$
5: **end for**
6: **return** R_0

For better performance, the modular multiplications can be evaluated using Montgomery method [10]. Montgomery modular multiplication produces a result correct modulo N but lying in the interval $[0,2N[$ so that a subtraction by N may be needed to have the result in $[0,N[$ as expected.[2]

As we will see, the time required to perform this possible subtraction can be discriminatory and allows an attacker to recover the value of secret exponent d. The attack iteratively recovers d, bit-by-bit, starting from the most significant position.

[2] We note that there are implementations of Montgomery multiplication that directly produce a result in $[0,N[$. See [6, 14].

We assume that the attacker already knows $d_{k-1},\ldots,d_{k-n+1}$. Her goal is to recover the value of the next bit, d_{k-n}.

1. The attacker *guesses* that $d_{k-n}=1$.
2. Next, the attacker randomly chooses t messages, $m_1,\ldots,m_t$, and prepares two sets of messages, $\mathscr{S}_0$ and $\mathscr{S}_1$, given by

$$\mathscr{S}_0=\big\{m_i \mid \text{Montgomery multiplication } R_0\leftarrow R_0\cdot R_1 \text{ in Line 4 of Algorithm 5 does not induce a subtraction for } j=k-n\big\}$$

   and

$$\mathscr{S}_1=\big\{m_i \mid \text{Montgomery multiplication } R_0\leftarrow R_0\cdot R_1 \text{ in Line 4 of Algorithm 5 induces a subtraction for } j=k-n\big\}\ .$$

3. For each message in set $\mathscr{S}_0$, the attacker requests the signature and measures the computation time to get it. She does the same for messages in set $\mathscr{S}_1$. Let $\bar{\tau}_0$ and $\bar{\tau}_1$ denote the average time (per signature request) for messages in $\mathscr{S}_0$ and $\mathscr{S}_1$, respectively.
4. If $\bar{\tau}_1\approx\bar{\tau}_0$ then the guess of the attacker was wrong and $d_{k-n}=0$. If $\bar{\tau}_1\gg\bar{\tau}_0$ (more precisely, if the time difference between $\bar{\tau}_1$ and $\bar{\tau}_0$ is roughly the time of a subtraction) then the attacker correctly guessed that $d_{k-n}=1$.
5. Now that the attacker knows $d_{k-1},\ldots,d_{k-n+1},d_{k-n}$, she iterates the attack to recover the value of d_{k-n-1} and so on.

It is worth noting that if $d_{n-k}=0$ then two sets, $\mathscr{S}_0$ and $\mathscr{S}_1$, behave as two random sets and so the average computation time for messages in $\mathscr{S}_0$ and $\mathscr{S}_1$ will be roughly the same.

The previous attack can be improved in a number of ways. In particular, the knowledge of public exponent $e=d^{-1} \bmod \phi(N)$ and lattice basis reduction techniques may help to speed up the recovery of d [4].

13.3 Simple Power Analysis

The power consumption of cryptographic devices, such as smart cards, depends on the manipulated data and of the executed instructions. It can be monitored on an oscilloscope by inserting a resistor in series with the ground or power supply pin. The so-obtained measurement is called a *power trace*.

Simple power analysis (SPA) [7] is a technique that involves direct interpretation of a power trace. This section presents some applications of simple power analysis. We show how to reverse-engineer the code of a program, namely how to recover an unknown instruction. The two other applications are actually attacks. We mount key recovery attacks against implementations of a private RSA exponentiation and a DES key schedule [11], as alluded in [8, Section 11].

13.3.1 Reverse-Engineering of an Algorithm

Several models are used to characterize the information leakage through the power consumption. The simpler model is the *Hamming-weight model*. It assumes that the power consumption varies according to the Hamming weight (i.e., the number of non-zero bits in a given bit string) of the manipulated data or the executed instruction.

Another model that works well in practice for many cryptographic devices is the *Hamming-distance model*. This model considers the number of flipping bits in the current state compared with the previous state. If Hw denotes the Hamming-weight function and if state_t and state_{t-1}, respectively, denote the state at clock cycles t and $t-1$ then the Hamming distance is given by

$$\mathrm{Hw}(\mathsf{state}_t \oplus \mathsf{state}_{t-1}),$$

where $\oplus$ denotes the XOR (exclusive OR) operator. It is easy to see that the above relation yields the number of flipping bits.

Of course, other models can be imagined for different technologies or chips.

Figure 13.1 shows four power traces from a given smart card as it evaluates $f(x,0)$ for byte values $x = 0, 1, 7$ and 255. We see that the traces are almost identical everywhere except at a few locations. These differences are called *trace signatures*. Different values for x give rise to different power consumption levels.

Trace signatures can be exploited to reverse-engineer a program code. Imagine that function f in the above example is performed on a smart card and is unknown. More precisely, imagine that we just know that at some point in the computation the smart card loads input byte x into the accumulator and applies some binary operator, say ∂, to x and 0:

$$f(x,0) = x \,\partial\, 0.$$

If the leakage model is known, it is easy to recover the unknown instruction ∂. Assume that the smart card follows the Hamming-distance model. It suffices to choose a location in the power trace that presents a signature and, for each input value $x \in [0,\ldots,255]$, to measure the corresponding power consumption. The measured

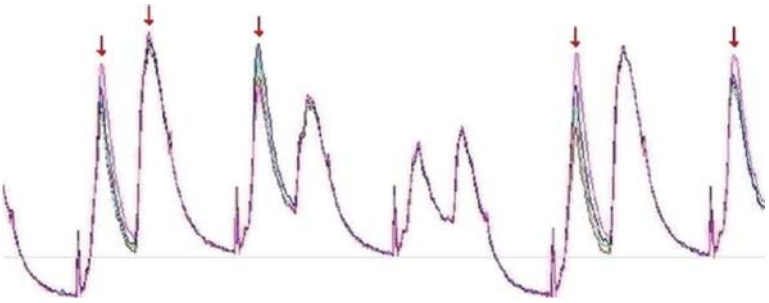

Fig. 13.1 Power traces of $f(x,0)$ for different values of x.

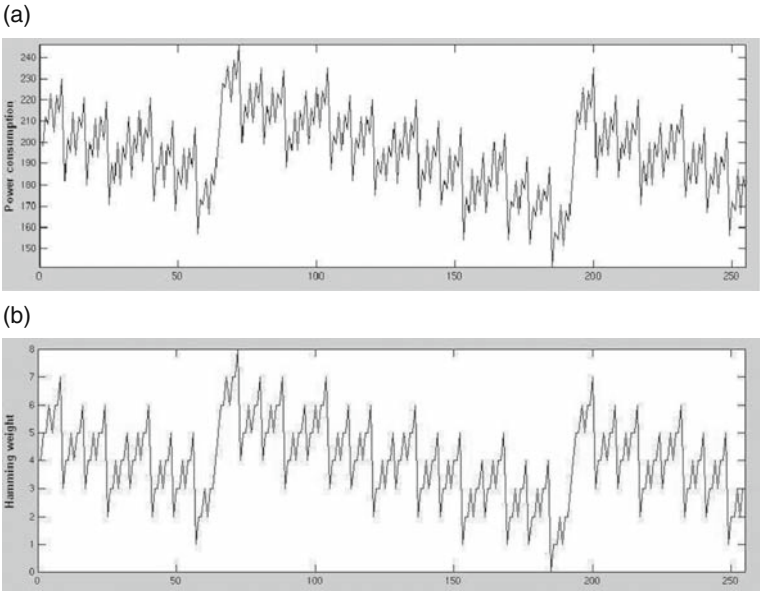

Fig. 13.2 Recovering an unknown instruction. (a) Instant power consumption, for $x \in [0,\ldots,255]$. (b) Hamming weight of $184 \oplus x$, for $x \in [0,\ldots,255]$.

power consumptions are then reported on a graph as depicted on Figure 13.2a. The next step is, for each byte value $\partial \in [0,\ldots,255]$, to plot the graph $\big(x, \mathrm{Hw}(\partial \oplus x)\big)$ for $x \in [0,\ldots,255]$. The graph that most resembles Figure 13.2a yields the value of unknown instruction. In our example, this unknown instruction is "184" which, for the considered smart card, corresponds to an XOR. Observe that Figure 13.2a and Figure 13.2b have globally the same shape.

13.3.2 Attack on a Private RSA Exponentiation

Figure 13.3 below represents (the beginning of) a power trace corresponding to the computation of an RSA signature with the square-and-multiply algorithm as per Algorithm 5.

We can distinguish two patterns in the power trace: a high-level pattern and a low-level pattern. We know that, at each step, the square-and-multiply algorithm performs a square and if the exponent bit is a 1 it also performs a multiply. Hence, it is not difficult to deduce that high levels correspond to multiplies and low levels to squares. The very beginning of the power trace presents a very low level of power consumption, this corresponds to the squares $R_0 \leftarrow 1^2 \pmod N$ (cf. Line 3 in Algorithm 5) until the first non-zero bit of d is encountered, in which case the accumulator, R_0, contains $\mu(m)$: $R_0 \leftarrow 1 \cdot R_1 \pmod N = \mu(m)$. The next bits of d are recovered from the left to the right with the following rule: A low level followed

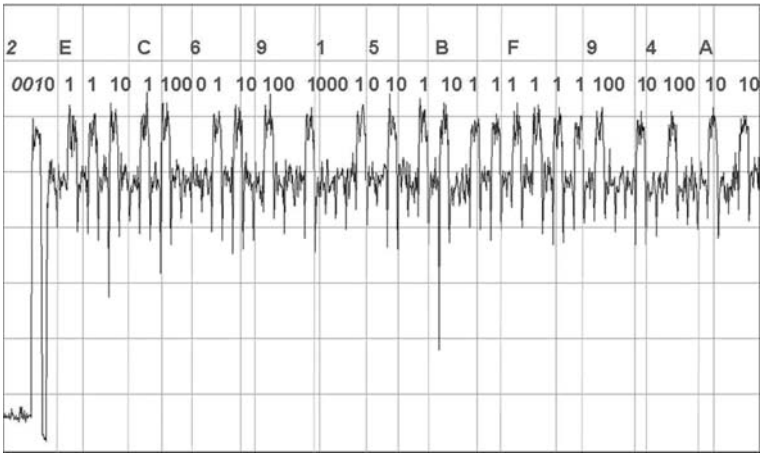

Fig. 13.3 Power trace of an RSA exponentiation.

by a high level corresponds to a 1-bit and a low level not followed by a high level corresponds to a 0-bit. So, we get (in hexadecimal) $d = \texttt{2EC6915BF94A}\ldots$, that is, the private RSA key!

Contrary to the timing attack of Section 13.2.2, this SPA attack equally applies to signature schemes using probabilistic paddings like RSA-PSS [3].

13.3.3 Attack on a DES Key Schedule

SPA-type attacks are not restricted to public-key algorithms but can potentially be applied to other types of cryptographic algorithms. Actually, a power trace can be viewed as a two-dimensional leakage function since it gives the power consumption level at a given time. It also reveals local timing information. We present in this section an SPA attack against an implementation of the DES key schedule.

DES is a block cipher that operates on 64-bit blocks of plaintext; the key length is 56 bits. DES comprises 16 identical rounds and makes use of a 48-bit sub-key for each round. The round sub-keys are obtained through what is called the key schedule. At round t, $1 \leq t \leq 16$, a 56-bit input buffer K_{t-1} is split into two 28-bit halves, C_{t-1} and D_{t-1}, and each half is circularly shifted by b bits to the left[3]—the value of b depending on the round number (see Table 13.1)—to produce a 56-bit output buffer K_t:

$$K_t = C_t \| D_t \quad \text{with } C_t = C_{t-1} \circlearrowleft b \text{ and } D_t = D_{t-1} \circlearrowleft b\,,$$

and where $\circlearrowleft$ denotes the circular shifting to the left.

[3] When DES is used in decryption, the circular shifting is done to the right.

Table 13.1 Number of key bits (b) shifted per round.

Round number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DES	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1
DES^{-1}	0	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

The round sub-key is then obtained by extracting 48 bits from K_t. The initial value K_0 is derived from the cipher key.

Algorithm 6 describes an implementation of a DES key shifting by one bit to the left. The algorithm is called b times to shift by b bits to the left. As the 56-bit input buffer K_{t-1} is updated with the 56-bit output buffer K_t, we drop the subscript and consider a 56-bit buffer $K = C|D$. In this implementation, buffer K is coded on 7 bytes $K[0], K[1], \ldots, K[6]$ with $0 \leq K[i] \leq 255$. We let $(K[i]_7, K[i]_6, \ldots, K[i]_0)_2$ denote the binary representation of byte $K[i]$ $(0 \leq i \leq 6)$. So, we have $C = (K[0]_7, \ldots, K[0]_0, K[1]_7, \ldots, K[1]_0, K[2]_7, \ldots, K[2]_0, K[3]_7, \ldots, K[3]_4)_2$ and $D = (K[3]_3, \ldots, K[3]_0, K[4]_7, \ldots, K[4]_0, K[5]_7, \ldots, K[5]_0, K[6]_7, \ldots, K[6]_0)_2$,

$$
\begin{array}{cccccccc}
K[0]_7 & K[0]_6 & K[0]_5 & K[0]_4 & K[0]_3 & K[0]_2 & K[0]_1 & K[0]_0 \\
K[1]_7 & K[1]_6 & K[1]_5 & K[1]_4 & K[1]_3 & K[1]_2 & K[1]_1 & K[1]_0 \\
K[2]_7 & K[2]_6 & K[2]_5 & K[2]_4 & K[2]_3 & K[2]_2 & K[2]_1 & K[2]_0 \\
K[3]_7 & K[3]_6 & K[3]_5 & K[3]_4 & K[3]_3 & K[3]_2 & K[3]_1 & K[3]_0 \\
K[4]_7 & K[4]_6 & K[4]_5 & K[4]_4 & K[4]_3 & K[4]_2 & K[4]_1 & K[4]_0 \\
K[5]_7 & K[5]_6 & K[5]_5 & K[5]_4 & K[5]_3 & K[5]_2 & K[5]_1 & K[5]_0 \\
K[6]_7 & K[6]_6 & K[6]_5 & K[6]_4 & K[6]_3 & K[6]_2 & K[6]_1 & K[6]_0
\end{array}.
$$

Algorithm 6 DES key shifting.

1: $\texttt{carry} \leftarrow K[3]_3$
2: **for** $j = 6$ downto 0 **do**
3: $\quad K[j] \leftarrow K[j] \circlearrowleft 1$ ▷ This operation also affects the carry flag!
4: **end for**
5: $K[3]_4 \leftarrow 0$
6: **if** (carry) **then** $K[3]_4 \leftarrow 1$

At the end of the **for** loop, the carry contains the value of bit $K[0]_7$ before shifting. This bit should replace bit $K[3]_4$ after shifting. This is done in two steps: $K[3]_4$ is first forced to "0" and if the carry is set then it is written as "1". Because the corresponding power trace will present a different pattern depending on the input bit $K[0]_7$, the value of this bit can be deduced. Moreover, since after 16 rounds 28 bits are going into the carry in a DES encryption (see Table 13.1), the value of 28 cipher-key bits can be recovered. The value of the 28 remaining cipher-key bits can be found by exhaustive search or by applying the same attack on DES in decryption mode (which yields the value of 27 bits, see Table 13.1).

13.4 Differential Power Analysis

Differential power analysis (DPA) [7] is a sophisticated power analysis technique that makes use of statistical methods on a collection of power traces. This section introduces the important notions: the DPA selection function and the DPA trace. We show how this enables an attacker to locate an algorithm within a power trace. We also show how this can be used to recover secret keys of cryptographic algorithms. To illustrate this, we present a DPA attack against an implementation of the AES block cipher [12] and generalize the timing attack presented in Section 13.2.2.

13.4.1 Bit Tracing

It is not always easy to precisely locate things in a power trace. Imagine for example that somewhere in a long cryptographic process data are encrypted to form a ciphertext which is next transferred to another location of RAM memory. The goal is to locate the encryption algorithm.

We need to introduce some notation. Let σ denote a *Boolean selection function* which returns $\sigma(y) = 0$ or $\sigma(y) = 1$, depending on the value of y. Let also $\langle\cdot\rangle$ represent the average operator and $\mathscr{C}_P(t)$ denote the power consumption of process P at time period t.

Differential power analysis runs in two phases. In the first phase, several power traces of a same process P are collected. In the second phase, depending on a *known* intermediate value y, a partition of two sets, $\mathscr{S}_0$ and $\mathscr{S}_1$, is made:

$$\mathscr{S}_0 = \{y \mid \sigma(y) = 0\} \quad \text{and} \quad \mathscr{S}_1 = \{y \mid \sigma(y) = 1\} \ .$$

The *DPA trace* is then given by

$$\Delta_P(t) := \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_1} - \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_0} ,$$

namely, the difference of the average power consumption curve corresponding to sets $\mathscr{S}_1$ and $\mathscr{S}_0$, for each time period t.

Basically, the DPA trace magnifies the effect of selection function σ. Back to our example, suppose that the cryptographic process is performed on an 8-bit device that obeys the Hamming-weight model. Suppose further that the ciphertexts are known and are represented on 16 bytes. If selection function σ is defined as returning the value of a given bit of the first byte of the ciphertext then sets $\mathscr{S}_0$ and $\mathscr{S}_1$ will contain ciphertexts for which a given bit is always a "0" and a "1", respectively. As a consequence, the average Hamming-weight value for the first byte of ciphertexts in set $\mathscr{S}_0$ will be of 3.5 while for ciphertexts in set $\mathscr{S}_1$ its average value will be of 4.5. Furthermore, since in the Hamming-weight model the power consumption is a function of the Hamming weight of the manipulated data, this difference between the average Hamming weights will translate into a difference between the average

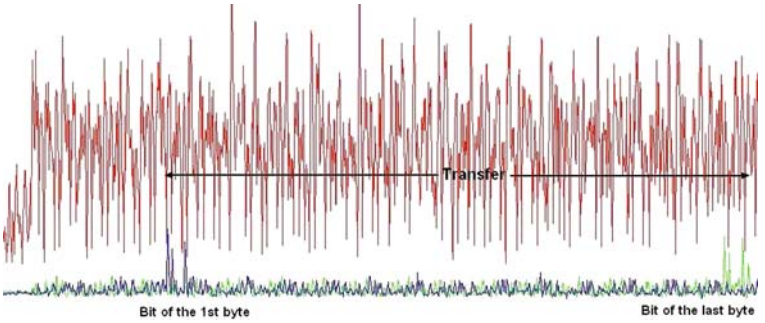

Fig. 13.4 Locating an algorithm.

power consumptions for set $\mathscr{S}_0$ and set $\mathscr{S}_1$ when the first byte of the ciphertext is being manipulated. This appears as a peak in the DPA trace. DPA peaks allow one to trace the bit used for the selection: each time this bit is manipulated a DPA peak should appear.

Applied to our example, we get a first DPA trace (bottom curve in Figure 13.4 presenting two peaks on the left-hand side). The second DPA trace (bottom curve in Figure 13.4 presenting two peaks on the right-hand side) is obtained similarly by the selection bit as a given bit in the last byte of the ciphertext. Hence, the ciphertext is manipulated between the two points determined by the DPA peaks. We can assume that this is the ciphertext being transferred for emission on the I/O or used in a subsequent function. In order for the ciphertext to exist, the encryption algorithm will have been executed. We therefore know that encryption algorithm is somewhere before in the power trace (top curve in Figure 13.4).

13.4.2 Attack on an AES Implementation

DPA can also be used to recover secret information. In the previous section, we have seen that DPA allows one to trace the activity of a selected bit. The idea was to use this bit to make a partition of two sets. An encryption algorithm takes on input a message and an encryption key to form a ciphertext. If the encryption key is unknown, it is not possible to make a partition on intermediate values during the course of the encryption. Nevertheless, DPA can be used by an attacker to validate the value of a key candidate as follows.

1. The attacker makes a guess on the value of the key.
2. Next, she applies the DPA methodology to some intermediate value depending on the key.
3. If the DPA trace does not present DPA peaks then it means that the guess was wrong and the attacker goes back to step 1. If it does present peaks then the attacker found the key, see, e.g., Figure 13.5.

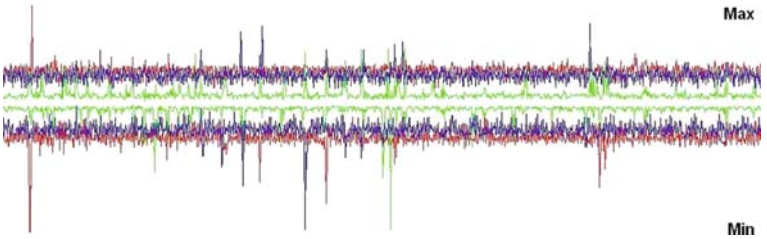

Fig. 13.5 DPA traces for different selection bits.

At first glance, it is not apparent to see in what this differs from a regular exhaustive key search. The difference is that the technique may be applicable to *part* of the key. We illustrate this on an implementation of the AES cipher.

The Advanced Encryption Standard (AES) is the successor of the older DES. The AES block cipher encrypts 128-bit blocks of plaintext and supports key lengths of 128, 192 and 256 bits. We consider the 128-bit version. The 128-bit plaintext, m_i, and 128-bit cipher key, K, are viewed as (4×4) matrices of bytes

$$m_i = (s_{u,v}^{(i)})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}} \quad \text{and} \quad K = (k_{u,v})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}}.$$

Plaintext m_i is first XORed with the cipher key and then gradually updated by applying round functions `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` in a series of 10 rounds. The `SubBytes` function substitutes each byte $s_{u,v}^{(i)}$, $0 \le u, v \le 3$, by another byte through a non-linear permutation $\mathrm{S_{RD}}$. As `SubBytes` has an effect on the value of a whole byte, selection function σ can be defined as the value of a given output bit of $\mathrm{S_{RD}}(s_{u,v}^{(i)})$, for example the first bit.

We recap in more detail how to recover a key byte $k_{u,v}$.

1. The attacker makes a guess on the key byte $k_{u,v}$ (there are 256 possible values).
2. The attacker partitions the t random 128-bit messages,

$$m_1 = (s_{u,v}^{(1)})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}}, \ldots, m_t = (s_{u,v}^{(t)})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}},$$

   into two sets, $\mathscr{S}_0$ and $\mathscr{S}_1$, given by

$$\mathscr{S}_0 = \left\{ m_i \mid \mathrm{bit}_1\left(\mathrm{S_{RD}}(s_{u,v}^{(i)})\right) = 0 \right\} \text{ and } \mathscr{S}_1 = \left\{ m_i \mid \mathrm{bit}_1\left(\mathrm{S_{RD}}(s_{u,v}^{(i)})\right) = 1 \right\}$$

   where $\mathrm{bit}_1\left(\mathrm{S_{RD}}(s_{u,v}^{(i)})\right)$ denotes the first bit of byte $\mathrm{S_{RD}}(s_{u,v}^{(i)})$.
3. Next, she evaluates the corresponding DPA trace

$$\Delta_P(t) := \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_1} - \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_0},$$

   and observes whether there are (significant) DPA peaks.
4. If not, the attacker goes back to step 1 with another guess for $k_{u,v}$. Otherwise, the attacker iterates the same attack to find the other key bytes.

The attack requires to evaluate at most 256 DPA traces to recover one key byte and so at most $256 \cdot 16 = 4096$ DPA traces to recover the cipher key.

13.4.3 Attack on an RSA Signature Scheme (2)

The attack we described in Section 13.2.2 readily applies by considering the power consumption as a side channel. We assume the Hamming-weight model. The two sets of messages can for example be defined as

$$\mathscr{S}_0 = \big\{m_i \mid \mathrm{lsb}(X_i) = 0\big\} \quad \text{and} \quad \mathscr{S}_1 = \big\{m_i \mid \mathrm{lsb}(X_i) = 1\big\},$$

where $X_i = \mu(m_i)^{(d_{k-1},\ldots,d_{k-n+1},1)_2} \bmod N$ and $\mathrm{lsb}(X_i)$ denote the least significant bit of X_i.

13.5 Countermeasures

Since the publication of side-channel attacks against cryptographic devices, numerous countermeasures have been devised. Countermeasures are available at both the hardware and the software levels but the sought-after goal is the same: to suppress the correlation between the side-channel information and the actual secret value being manipulated.

A first class of countermeasures consists in reducing the available side-channel signal. This implies that the implementations should behave regularly. In particular, branchings conditioned by secret values should be avoided.

A second class of countermeasures involves introducing noise. This includes power smoothing whose purpose is to render power traces smoother. The insertion of wait states (hardware level) or dummy cycles (software level) is another example. This technique is useful to make statistical attacks harder to implement as the signals should first be "re-aligned" (i.e., the introduced delays should be removed). One can also make use of an unstable clock to desynchronize the signals. At the software level, data masking is another popular countermeasure. For example, an RSA signature (cf. Section 13.2.2) can equivalently be evaluated as

$$S = \big[(\mu(m) + r_1 N)^{d + r_2 \phi(N)} \bmod (r_3 N)\big] \bmod N,$$

for three random values r_1, r_2 and r_3.

Efficiency is not the only criterion to deal with when developing cryptographic products. The implementations must also be resistant against side-channel attacks. To protect against the vast majority of known attacks, they must combine countermeasures of both classes described above. Moreover, experience shows that the best results are attained by mixing hardware and software protections.

Acknowledgments I would like to express my gratitude to my former colleagues of Gemplus and in particular to David Naccache, Francis Olivier and Michael Tunstall. I am also grateful to my colleagues of the Security Labs of Thomson for comments. The figures of this chapter are courtesy of Gemalto.

13.6 Exercises

1. Suppose that in the password verification routine of Algorithm 4, the comparison is done in a random order. More specifically, let $\mathfrak{S}$ be the set of permutations on the set $\{0,1,\ldots,7\}$. At each execution a permutation s is randomly chosen in $\mathfrak{S}$ and the comparison (cf. Line 2 of Algorithm 4) is replaced with
   2: **if** $(\widetilde{P}[s(j)] \neq P[s(j)])$ **then return** "`false`" .

   a. Do you think that this implementation is secure against timing attacks? Can you break it?
   b. Can you extend your attack if a random delay is added before returning the status?

2. Consider the timing attack given in Section 13.2.2 against an implementation of the "hash-and-sign" RSA signature scheme (Algorithm 5).

   a. The attack recovers one bit of secret exponent d at a time. Can you modify it to recover more than one bit at a time?
   b. Let $N = pq$ be a k-bit RSA modulus ($k \geq 1024$) where p and q are two (different) balanced secret primes satisfying $\gcd(e,(p-1)(q-1)) = 1$.
      i. Prove that for public exponent $e = 3$, the corresponding private exponent is given by $d := 3^{-1} \bmod \phi(N) = \frac{1+2(p-1)(q-1)}{3}$. Next, letting $\hat{d} = \frac{1+2N}{3}$, prove that $\hat{d} - d < 2^{\lfloor k/2 \rfloor + 1}$.
      ii. Explain how the previous relation can be used to speed up the attack when public exponent $e = 3$.
   c. The so-called square-and-multiply-*always* algorithm is a balanced version of Algorithm 5. It requires an additional register (for convenience, we write it as R_{-1}). It is obtained by replacing Lines 1 and 4 of Algorithm 5 with
      1: $R_0 \leftarrow 1; R_1 \leftarrow \mu(m); R_{-1} \leftarrow \mu(m)$
      4: $b \leftarrow d_j - 1; R_b \leftarrow R_b \cdot R_1 \pmod N$
      respectively. Can you mount a timing attack against this modified implementation?

3. Figure 13.1 represents four power traces of $f(x,0)$ for $x = 0$, 1, 7 and 255. You can observe that the highest consumption level is not always obtained for the same value of x (look at the different trace signatures). Does this give you indications on the leakage model?
4. The power trace of Figure 13.3 corresponds to an RSA exponentiation with private exponent $d = \texttt{2EC6915BF94A}\ldots$

a. How can you figure out the number of leading 0-bits? Namely, how can you determine that d starts with $\underline{00}101110\ldots = \texttt{2E}\ldots$ and not for example with $\underline{0}1011101\ldots = \texttt{5D}\ldots$?
b. Let ϕ denote Euler's totient function. Adding to d a random multiple of $\phi(N)$—or of $(ed-1)$ if $\phi(N)$ is not available—does not modify the result of an RSA exponentiation. Do you think it helps if, at each execution, private exponent d is randomized as $d \leftarrow d + r\phi(N)$ for a random integer r?

5. a. Explain why the circular shifting is done to the right when DES is used in decryption whereas it is done to the left for encryption. Why is there no circular shifting for the first round of DES^{-1} (see Table 13.1)?
   b. The DES key shifting presented in Algorithm 6 is susceptible to SPA. Propose an SPA-resistant implementation of it. Does your implementation resist DPA?
6. The bit-tracing method described in Section 13.4.1 is used to locate where in the power trace the ciphertext is transferred to RAM memory.
   a. The analysis requires the knowledge of the ciphertext. It is worth remarking that the ciphertext is known but not chosen. What is the impact on the analysis if the encryption algorithm is weak?
   b. As presented, the analysis considers a given bit of the first byte of the ciphertext to locate the beginning of the memory transfer. Propose another selection function so as to obtain higher peak values.
7. a. The DPA attack against AES in Section 13.4.2 assumes that the attacker has access to the input plaintexts. Can you adapt the attack in the case where the attacker only knows the output ciphertexts?
   b. It might be the case that different key byte candidates give rise to DPA peaks. In such a case, how could you identify the right candidate?
   c. Semantic security implies that encryption should be probabilistic. This can be achieved with AES by encrypting a 128-bit plaintext m under key K as $C = (c_1, c_2)$ with $c_1 = \text{AES}_K(r)$ and $c_2 = m \oplus r$ for a 128-bit random r. Plaintext m can then be recovered from ciphertext $C = (c_1, c_2)$ using key K as $m = c_2 \oplus \text{AES}_K^{-1}(c_1)$.
      i. Do you think that this implementation is secure against DPA attacks?
      ii. Can you mount a DPA attack against this randomized version of AES?
8. Give a detailed presentation of the DPA attack against the RSA signature scheme as described in Algorithm 5.

13.7 Projects

Power analysis requires specialized equipment which is not necessarily easily accessible. So, we will only consider timing information as a side channel. These projects can be done in C or using a higher-level language like Pari/GP [1].

1. a. Implement the password verification routine as presented in Algorithm 4.
   b. Mount the timing attack described in Section 13.2.1 against your implementation.
   c. Design a password verification routine that resists timing attacks.
2. a. Implement the Montgomery modular multiplication (see [10] or [9, Section 14.3.2] for details).
   b. Implement the RSA signature scheme as presented in Algorithm 5 using your implementation of the Montgomery modular multiplication. To simplify the implementation, you may assume that input messages are in $[0, N-1]$ and replace hash function μ with the identity map.
   c. Mount a timing attack against your implementation.
   d. Optimize your attack for the case $e = 3$ (see Problem 2(b)).
   e. Consider other exponentiation algorithms (e.g., [9, Section 14.6]) and mount timing attacks against the resulting implementations.

References

1. C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. Pari/GP. Freely available at URL http://pari.math.u-bordeaux.fr
2. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security*, pp. 62–73. ACM Press, 1993.
3. M. Bellare and P. Rogaway. The exact security of digital signatures. In *Advances in Cryptology – EUROCRYPT '96*, LNCS vol. 1070, pp. 399–416. Springer, 1996.
4. D. Boneh, G. Durfee, and Y. Frankel. Exposing an RSA private key given a small fraction of its bits. In K. Ohta and D. Pei, editors, *Advances in Cryptology – ASIACRYPT '98*, LNCS vol. 1514, pp. 25–34. Springer-Verlag, 1998.
5. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. In J. J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications (CARDIS '98)*, LNCS, vol. 1820, pp. 167–182. Springer-Verlag, 2000.
6. G. Hachez and J.-J. Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, LNCS, vol. 1965, pp. 293–301. Springer-Verlag, 2000.
7. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, LNCS, vol. 1666, pp. 388–397. Springer-Verlag, 1999.
8. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, LNCS, vol. 1109, pp. 104–113. Springer-Verlag, 1996.

9. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. Online version available at URL http://www.cacr.math.uwaterloo.ca/hac/
10. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
11. National Bureau of Standards. Data encryption standard. Federal Information Processing Standards Publication 46, U.S. Department of Commerce, January 1977.
12. National Institute of Standards and Technology. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce, November 2001.
13. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
14. C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.