PV181: Training 6 - Smartcard security aspects

a. Logical attacks – On-card type control

The attack is directed against type control of the JavaCard Virtual Machine, that allows execution of the untrusted code after off-card type verification. Idea of attack is pretty simple: to obtain two pointers pointing on the same memory place, but with the different data type. If second type of pointer is of larger primitive type than original one, larger memory than originally allocated can be accessed.

Preparation of the attack:

1. Compile project *TypeAttackServer* with methods *typeAttack()* and *TypeAttackInterface::typeAttack()* with data type of input buffer set to **byte[].**
2. Compile project *TypeAttackClient*. This project use interface *TypeAttackInterface* with expected data type **byte[].** Then convert to JavaCard byte code (*.jar, *.cap).
3. In *TypeAttackServer* project, change data type of input parameter of methods *typeAttack()* and *TypeAttackInterface::typeAttack()* onto **short[].** Compile project and convert to JavaCard byte code (*.jar, *.cap).
4. Load applets TypeAttackServer and TypeAttackClient onto the same smart card.

Attack execution:

5. Select *TypeAttackServer* and set parameter *m_readPos* (index of value to be read with type short) on requested value. Parameter readPos is set using APDU command with CLA=0xB0, INS=0x30, DataIn=value_of_m_readPos. Values up to 130 (can differ slightly with particular smart card type) will point into correctly allocated memoery of APDU command data array. Higher values up to ~260 will point after the end of allocated array and should not be readable.
6. Select applet TypeAttackClient and send APDU command with CLA=0xB0, INS=0x40, DataIn=01 02 03 04 05 06.The method *TypeAttackClient::startAttackMethod()* is called and input buffer of APDU command with data type **byte[]** is passed to method *TypeAttackServer:: typeAttack(),* which iterprets input parameter as the array with **short[]** data type. We now should have the pointer to array of short[260] (around 520 bytes) pointing onto array with real length only around 260 bytes! We can then read the value of **short** type from position given by *m_readPos* index parameter (set before attack by overself) and return back on output. In case, when memory controls are implemented correctly, then reading ends up with an exception. Otherwise, value from memory that should not be accessible is returned.

Moral:

- If smartcard performs on-card type control before applet instalation, then type mismatch should be detected even before the attack applet is installed.
- If smartcard perform runtime type control, than type mismatch should be detected during array exchange in *startAttackMethod()* even when subsequent read is performed in correctly allocated area (m_readPos < 130).
- If smartcard do not perform type controls mentioned above, but use some other methods to guard array boundaries, than exception should be thrown when attempt to read/write outside allocated memory is detected (m_readPos < 130).

Possible modifications:

- Use your own buffer allocated in EEPROM/RAM instead of APDU incoming data buffer.
- Do not read from array using high-level index access (e.g. short tmp = buffer[m_readPos];) in step 6, but use low level copy functions like Util.arrayCopyNonAtomic() or Util.arrayFillNonAtomic(). Check routines for reading/writing outside allocated memory may not apply for low-level operations.

In this basic attack, *Shareable* interface is used to allow uploading two codes with different data types. No direct modification of the compiled byte code is thus required. Attack can be extended to "after compilation" modification of byte code to mismatch data types even in the same applet. However, more obscure way of passing data pointers, better change to bypass JCVM controls.

b. SCSAT02 measurement board

The laboratory measure device SCSAT02 (further referred as "the board") has been developed in the cooperation of FI MUNI and VUT. The board is deployed between the smartcard and reader (see Figure 1). It is equipped with an oscilloscope to measure the power consumption of the smart card during the execution of selected commands with the sampling frequency up to 5MHz. It only has a limited internal memory (512kB) for local storage of measured data that are later transported to a PC using the LPT port. Inverse smartcard reader is used to physically connect smart card to the reader.
The board can be used for power analysis, time analysis, fault induction attacks (power glitches) and for monitoring the PC-smartcard communication (useful for the reverse engineering of proprietary protocols).

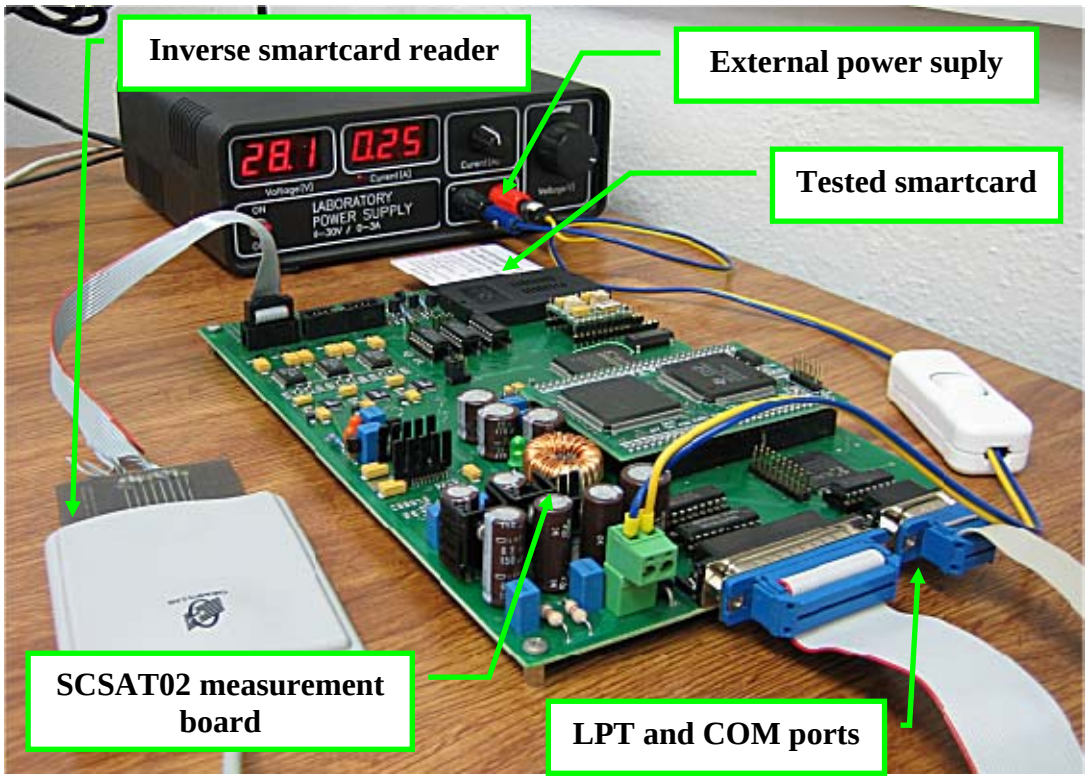

Figure 1 - SCSAT02 measuring board

c. Time analysis

The basic idea is to obtain some additional information about the smartcard internal state and the sensitive data processed in the smartcard by monitoring the duration of the execution [Ko96]. Relies on the assumption, that secret data are processed on the card and the processing time:

- depends on the value of secret data
- leaks information about secret data
- leakage can be measured (at least as differences)

Defense against the timing attack is simple in theory: algorithm execution lengths must be independent on the processed data. However this requirement typically prohibits speed optimizations and requires usage of specially designed algorithms rather that simple and clear ones.
Note: The execution length of algorithms is typically NOT measured as the time between the incoming and outgoing APDU, because too much of time noise can be introduced by intermediate components (smart card reader and PC smart card subsystem). More often, length of the operation is observed from simple power trace (see Figure 2 for example).

Examples:

1. **PIN verification** – Assume, that PIN verification is implemented as per-digit comparison, that stop directly after the first non-matching digit in the PIN to be verified. Rejection of the PIN with first two correct digits will take longer time than rejection of a PIN with the incorrect digit as the first one.
2. **Exponentiation** – Assume that m^d is implemented using Montgomery algorithm, where d is the secret value maintained by the card. The algorithm computes square (shift) of some value *S* every time and multiplies *S* by *m* if the i-th bit of *d* is equal to one. Multiplication typically takes much longer time than simple shift and bits of *d* can be thus revealed. This idea can be used to attack naive implementation of the RSA algorithm.
3. **Writing into memory –** Assume that an applet on the smartcard wants to hide the number of bytes that are being written into the memory (e.g. secret logging, where alert message is much longer than the neutral one). However, writing 65 bytes requires two invocations of the low level native function for memory management, as this functions write by 64-byte blocks (see Figure 2).

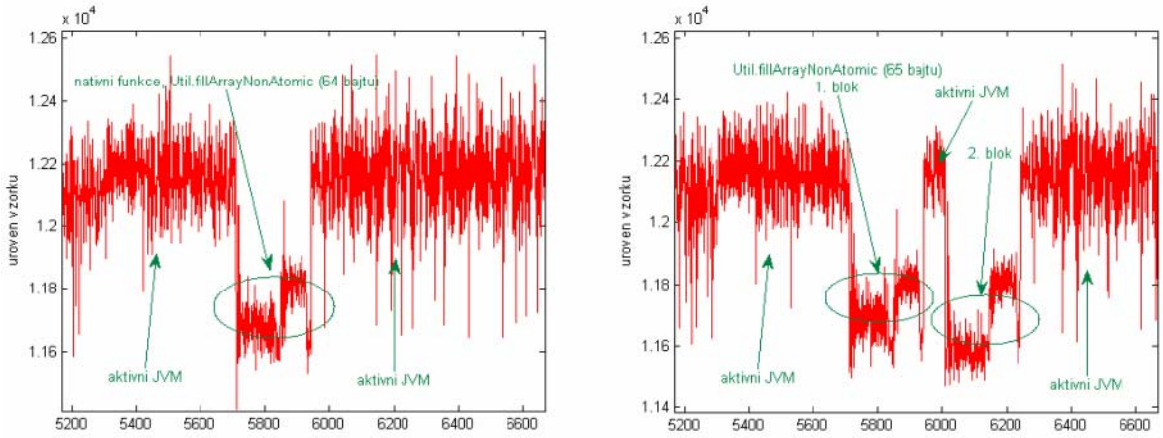

Figure 2 – Different power trace when writing 64B and 65B into memory

d. Power analysis

The power analysis is based on measuring the dependency of the execution on the power consumption [Ko99]. As smartcards do not have their own source of energy and depend on external energy supply, the current consumption of the card can be measured in a simple way. Assumptions are similar as for the time analysis except that the variance in power consumption is measured instead of the execution time. Simple power analysis works directly with the measured power trace without any utilization of more sophisticated statistical analysis. The differential power analysis employs more sophisticated statistical analysis techniques to remove noise from the signal and to discover less obvious dependency of the power consumption on the processed data.

Simple power analysis (SPA)

(Simple power analysis) can be used for revealing the secret data handled by the smartcard, reverse-engineering of smartcard code (PIN verification procedure), determining the position of sensitive operations (for subsequent use in the DPA) and others.

Reveling secret key

Most commonly exploited dependency between the consumption and the data is the hamming weight (number of bits equal to one in an operand) or the hamming distance (number of different bits between two operands). Attacks are based on the assumption that simple operations like XOR or writing into memory will consume more energy if the hamming distance of arguments is higher as more transitions of transistors from 0 to 1 and from 1 to 0 is required. See Figure 3 for an example of the leakage in the case of 8-bits operands.

If an attacker knows the Hamming weight of each of the *k* *n*-bit words of a secret key, then the brute-force search is reduced from 2^{kn} to $\left\lceil \frac{\sum_{m=0}^{n} \binom{n}{m}^2}{2^n} \right\rceil^k$

The probability that a random n-bit key will have the hamming weight equal to m is (n over m)/2^n. If we know that the hamming weight of the key is equal to m, we must search (n over m) possible keys. The expected number of searched keys is thus the number of keys with the weight m weighted by probability, that random key will have weight m for all possible values m = 0…n.

DES example: 2^{56} keys reduced to 2^{40} (*n* = 8, k = 7).

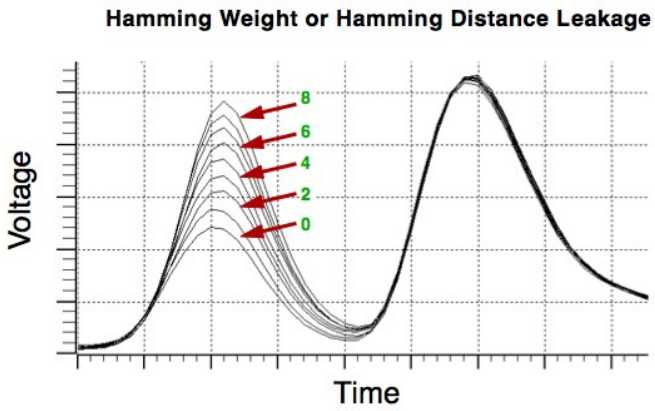

Figure 3 - Hamming weight or distance leakage, source [Slo02]

Reverse engineering

Cryptographic smartcards are often expected to offer protection not only of the cryptographic keys, but also of the executed code. In contrast to Kerkhoff‘s principles, designers might think that the smartcard environment can offer a protection against disclosure of some details of the implemented code. However, different operations have different power consumptions which can be exploited to reverse-engineer the executed code (see Figure 4).

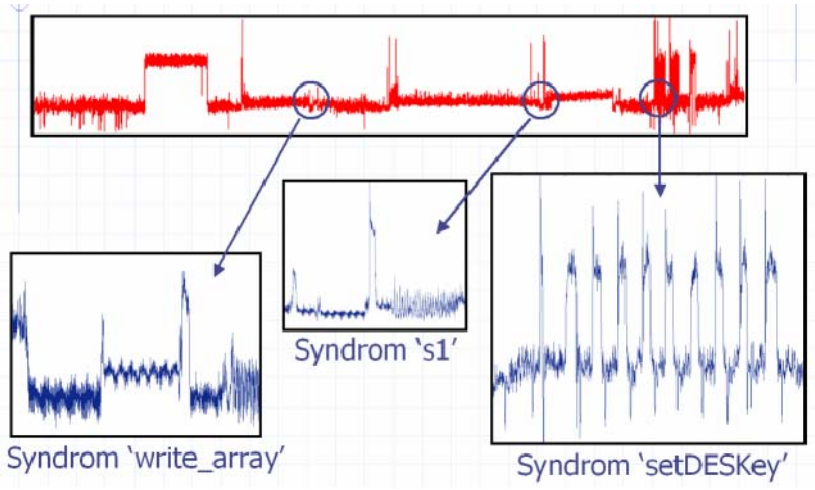

Figure 4 - Reverse-engineering of Javacard applet

Brute-forcing correct PIN value

An example of the importance of the reverse-engineering can be the PIN verification procedure. As the typical PIN consists of four digits only, the number of allowed attempts to verify the PIN must be limited to prevent brute-force search over all 10000 possible values. The typical number of allowed attempts is 3-5. The power analysis can reveal the way how the PIN verification procedure is implemented. The correct implementation of the PIN verification procedure is:

1. Decrease the retry counter
2. Verify PIN
3. Increase retry counter, if correct PIN was supplied.

Initial counter decrease is important against an attacker who is able to turn the power supply off before the result of PIN verification is stored in the memory (in the retry

counter). See Figure 5 for the correct implementation and Figure 6 for an incorrect implementation.

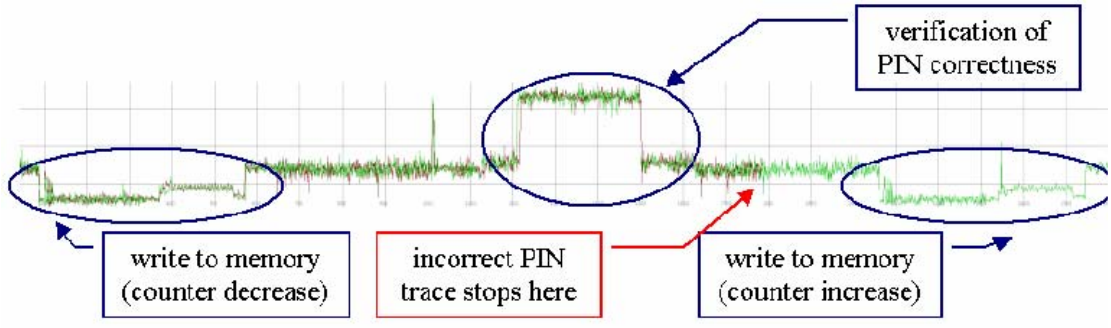

Figure 5 - Correct implementation of the PIN verification procedure

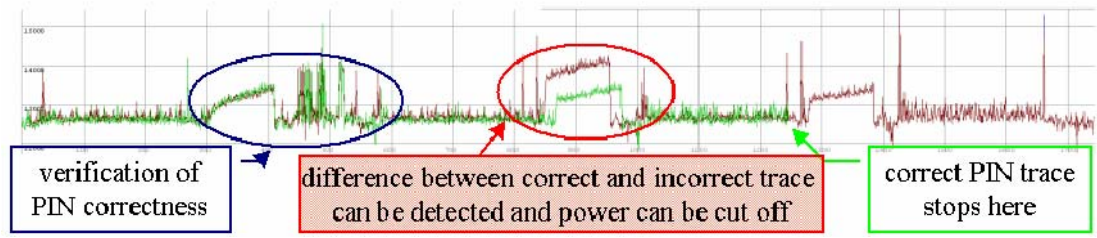

Figure 6 - Incorrect implementation of the PIN verification procedure

Differential power analysis (DPA)

The differential power analysis is described in [Ko99]. The basic DPA algorithm can be viewed as a clever way of averaging traces of an encryption with different plaintexts in such a way that a part of the secret key can be revealed. See Figure 7 for the algorithm overview.

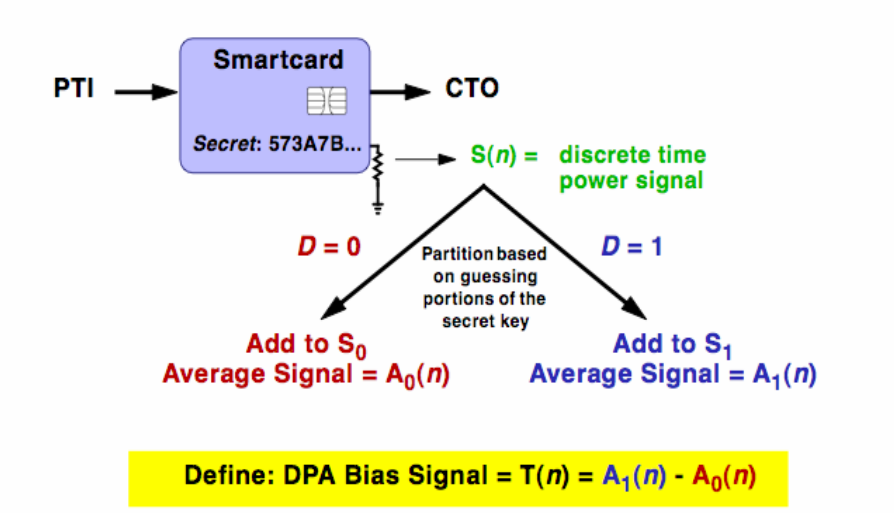

Figure 7 - Basic DPA algorithm, source [Slo02]

The DPA algorithm against DES-like ciphers works as follows:

1. Random input plaintext P_i is sent to the card for encryption
2. Power consumption of $P_i^k \oplus SecretKey^k$ is measured => $S_i(n)$
3. Steps 1, 2 are repeated multiple times (10^3 - 10^5) resulting in pairs [P_i, $S_i(n)$]

4. Choose suitable partitioning function D (e.g. Hamming weight here)
5. For all possible values of $SecretKey^k$ do
    a. For all pairs $[P_i, S_i(n)]$ compute $w = HamWeight(P_i^k \oplus SecretKey^k)$ and apply partitioning function D
        i. if $w > k / 2$ then add $S_i(n)$ to group S_0
        ii. if $w < k / 2$ then add $S_i(n)$ to group S_1
    b. Compute average from $S_0 \Rightarrow A_0$ and $S_1 \Rightarrow A_1$
    c. Compute difference bias $T(n) = A_1(n) – A_0(n)$
    d. Display T(n) (see example on Figure 8)
6. Correct key guess from step 5 should display most significant spikes in 5d.

Note: P_i^k stands for the k-bit block of the i-th plaintext. $SecretKey^k$ stands for the k-bit block of the (expanded) secret key. Size of the block k (in bits) depends on the attacker‘s choice (denoted as k-bit DPA), typically between 1 to 8 bits. Lower size of the block results in faster execution of the DPA algorithm in the step 5, but requires more power traces in step 1.

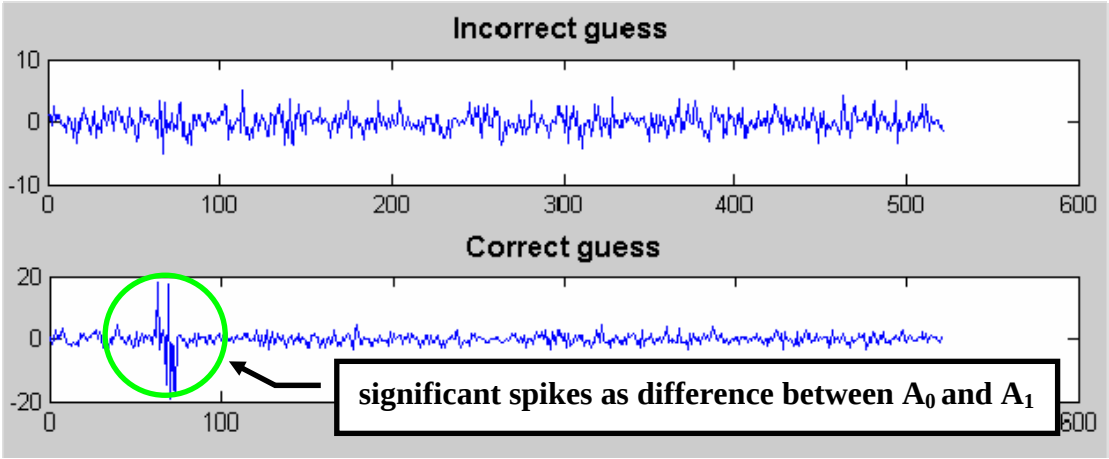

Figure 8 - DPA bias traces for correct and incorrect key guess

Power analysis countermeasures

Protection techniques against the power analysis must deal with a limited area of the smartcard chip and should not decrease the performance of the execution significantly. Countermeasures can be done both on hardware and software level:

Hardware countermeasures:

- **Noise generator –** Additional circuit that does nothing else than consuming random amount of energy is used and thus random noise is introduced into the power consumption. Advantages: relative simple design; it can be effective. Disadvantages: can be expensive to implement (consume chip area), might be disabled through tampering, not energy efficient (contactless cards), signal still present.
- **Power signal filtering –** Passive: Additional capacitators that straighten out the power trace. Active: Compensation techniques that react on actual power consumption. Disadvantages: might be disabled by tapering, passive are limited by chip size, active are likely to lag behind power supply changes.
- **Novel chip design –** Battery on chip, detachable power signals (two capacitators, one is powered from an external source, second one is used by

the chip – external power is never connected directly to the chip), special types of logics (dual rails) etc. Disadvantages: not always practical for legacy systems, signals can leak in other ways.

Software countermeasures:

- **Time randomization –** Insert random NOPs, random execution ordering. Advatages: easy and cheap to implement without modification of the existing hardware. Disadvatages: susceptible to advanced signal analysis (re-synchronization).
- **Masking techniques –** Random value is applied (xored) to original argument, original operations are performed with masked argument. Later, result is unmasked. Advatages: eliminate thread of 1st order DPA. Disadvantages: some crypto functions are difficult to mask, susceptible to 2nd order DPA.

Homework (choose one from following):

1. Modify applets *TypeAttackServer* and *TypeAttackClient* to include all proposed modifications of the basic attack (your own array in RAM and EEPROM, Util.arrayCopyNonAtomic and Util.arrayFillNonAtomic for read and write instead of high level buffer[index]).
2. Write a code (in C, Java or Matlab) that will detect the power trace of the verification of an incorrect PIN and trigger before the counter is decreased. Use the supplied traces of the correct and incorrect PIN verifications. Your program should be able to distinct between the correct and incorrect trace online, without "looking ahead", so if you start process value by value from begining, than you should say 'stop power' before power trace falls down from the big peak in the middle of the Figure 6.
3. Program a code (in C, Java or Matlab) that performs DPA on the supplied power traces and guesses the correct key. Use 4-bit DPA with Hamming weight decision rule to model dependency of power consumption on processed data. See '!Readme_dpa.txt' inside a zipped file for more information.

References

[Ko96] P.Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems", at www.cryptography.com/resources/whitepapers/TimingAttacks.pdf

[Ko99] P. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis," at http://www.cryptography.com/resources/whitepapers/DPA.pdf

[Slo02] R. Sloan et. al: „Smart-Card Security under the Threat of Power Analysis Attacks" at http://www.cs.uic.edu/~sloan/my-papers/ieee-messerges-proof.pdf

[AO00] M. Aigner, E. Oswald: Power Analysis Tutorial http://www.iaik.tugraz.at/aboutus/people/oswald/papers/dpa_tutorial.pdf