Contents

List of Figures

List of Tables

List of Algorithms

Chapter I

Smart Card Security

Kostas Markantonakis [I.1], Keith Mayes [I.2], Michael Tunstall [I.3],
Damien Sauveron [I.4], and Fred Piper [I.5]

In recent years smart cards have become one of the most common secure computing devices. Their uses include such diverse applications such as: providing secure wireless communication framework, banking and identification. Direct threats to smart card security can be invasive (attacks that alter the chip inside the card), analysis of a side channel, induced faults, as well as more traditional forms of attack. This chapter will describe the various attacks that can be applied to smart cards, and the subsequent countermeasures required in software to achieve a secure solution. Invasive attacks are considered beyond the scope of this chapter and can be mitigated with software countermeasures. A case study on the various generations of the European mobile telephone networks is given as an example of how the deployment of countermeasures has changed due to the various attacks described in this chapter.

I.1 Introduction

Smart cards are used to ensure secure execution and secure storage. A secure smart card has to guarantee that secret information cannot be retrieved or modified by an

[I.1] Smart Card Centre, Information Security Group, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK, k.markantonakis@rhul.ac.uk

[I.2] Smart Card Centre, Information Security Group, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK, keith.mayes@rhul.ac.uk

[I.3] Smart Card Centre, Information Security Group, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK, m.j.tunstall@rhul.ac.uk, http://www.geocities.com/mike.tunstall/

[I.4] XLIM — UMR CNRS 6172, University of Limoges, 123 avenue Albert Thomas — 87060 LIMOGES Cedex, FRANCE, damien.sauveron@unilim.fr, http://damien.sauveron.free.fr/

[I.5] Information Security Group, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK, f.piper@rhul.ac.uk

unauthorised entity. This information could be secret keys used in cryptographic algorithms, an e-purse balance, etc. Smart cards are generally not vulnerable to classical software attacks implemented against PCs, but their unique form as led to different methods of attack. These attacks are briefly described below; a more thorough description including a description of the necessary countermeasures is given in subsequent sections.

Side Channel Attacks: Sensitive systems that are based on smart cards use well-known cryptosystems. Generally, these cryptosystems have been subject to rigorous mathematical analysis in the effort to uncover weaknesses in the system. The cryptosystems used in smart cards are therefore not usually vulnerable to these types of attacks. Since smart cards are small, physical objects that can actually be carried around in ones pocket, adversaries turned to different mechanisms of attack. Side channel analysis is a class of attacks that seek to deduce information in a less direct manner. This is achieved by extracting secret information held inside devices, such as smart cards, via the monitoring of information that leaks naturally during its operation.

The first attack of this type was a timing attack against RSA and some other public-key cryptographic primitives [42]. This was followed by attacks using the power consumption [43] and electromagnetic emanations [28] as a side channel to extract information.

Fault Attacks: Since the publication of an fault attack against the RSA in 1997 [12] fault analysis has come to the foreground as a possible means of attacking devices such as smart cards. The vast majority of papers written on this subject involved very specific faults (usually a bit-flip in a specific variable) that are extremely difficult to produce. However, there have been several attacks published that have enough freedom in the type of fault required that they can be realised with current fault injection methods.

Fault attacks have been implemented against chips that could be used in smart cards [4]. This has meant that countermeasures against this type of attack need to be implemented in embedded devices to protect against this class of attack.

Multi-Application Security: Modern smart cards usually incorporate a java virtual machine that is capable of interpreting multiple applets present in the smart card's Non-Volatile Memory (NVM). There are therefore two security considerations that need to be taken into account when implementing a java virtual machine: secure bytecode interpretation and secure resource partitioning. Bytecode needs to be implemented such that the security of the smart card cannot be compromised by a malicious applet writer, and resources managed such that one applet cannot access the secrets of another applet. This is a relatively new problem in smart card operating system design, as previously it was not possible to load arbitrary applications into smart cards.

A secure smart card needs to include all of the above security considerations. There are also other security aspects that need to be taken into account to have a totally secure solution. There are hardware features that are used to prevent reverse engineering of a chip, its operating system, and any secret keys held within. A discussion of this topic is beyond the scope of this chapter but needs to be taken into account when choosing a chip for use as a smart card. This can be mitigated by including more stringent software countermeasures to achieve a secure solution when inadequate hardware security is present.

The specific case of side channel and fault attacks applied to smart cards is covered in Section I.2. This is followed by a discussion of multi-application platform security in Section I.3. In Section I.4 a case study is given detailing the changes in security in the European mobile telecommunications standards in response to various smart card based attacks. This is followed by a summary in Section I.5.

I.2 Smart Card Specific Attacks

Sensitive systems that are based on smart cards use protocols and algorithms that have usually been subjected to rigorous analysis by the cryptographic community. Attackers have therefore sought other means to circumvent the security of protocols and algorithms used in smart card based systems. As smart cards are small, portable devices they can easily be put in a situation where their behaviour can be observed and analysed. The simplest form of analysis of this type is intercepting all the communication between a smart card and its reader. Tools are readily available on the Internet [62] that allow the commands to and from a smart card to be logged. This problem is relatively easy to solve with secure sessions, as proposed in [31], but highlights the ease of man-in-the-middle type attacks against smart cards.

More complex attacks can be realised by monitoring information that leaks naturally during a smart cards processing of information, referred to as side channel attacks. Smart cards can also be attacked by inducing a fault during their normal processing to change the chips behaviour. Comparing a faulty result with a standard response can then be used to make deductions about secrets held by a smart card. These attacks are referred to as fault attacks. The following sections describe these two classes of attack in more detail.

I.2.1 Side Channel Attacks

The first example of a side channel attack was proposed in [42]. This involved observing the differences in the amount of time required to calculate a RSA signature for different messages to derive the secret key. This attack was conducted against a PC implementation but a similar analysis could be applied to smart card implementations. It would be expected to be more efficient against a smart card as more precise timings can be achieved with an oscilloscope or proprietary readers. An example of equipment capable of acquiring this sort of information is shown in Figure I.1. The I/O trace can be seen on the oscilloscope as the yellow trace that will allow the exact amount of clock cycles taken by a given command can be determined (the smart card being analysed is in the reader in the top left corner of the

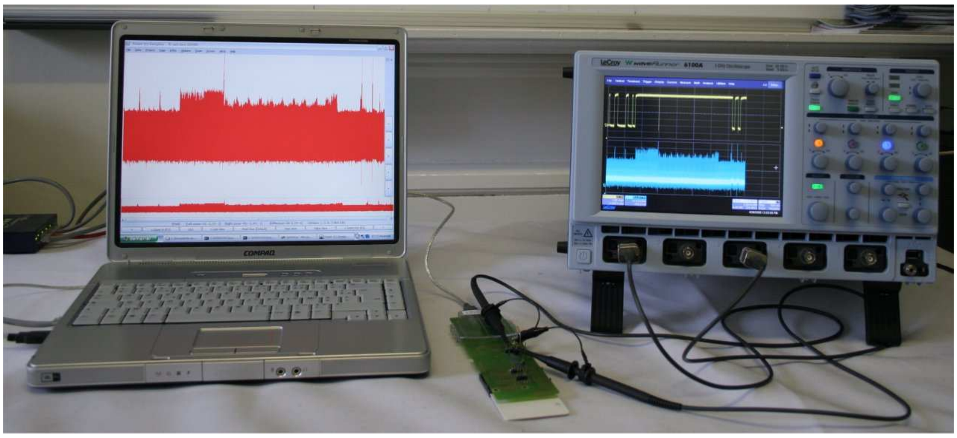

Figure I.1. Data Acquisition Tools.

image). For this reason, the amount of time taken for a command is often constant or, if variable, not related to any secret information.

The most common form of side channel attack is the analysis of the power consumption, originally proposed in [43]. This is measure by placing a resistor in series with a smart card, between the card and earth. The potential difference across this resistor is measured with an oscilloscope. This gives a measurement of current in the circuit between the chip and the resistor. The equipment shown in Figure I.1 is capable of taking these sort of measurements. The current at each point in time during a command is shown as the blue trace on the oscilloscope. An enlargement of this trace is shown on the computer screen, where the minimum, average and maximum of the points represented by one pixel can be seen. This is referred to as the power consumption in the literature, and the attacks based on these measurements are referred to as power analysis attacks. There are two main types of power attack; these are simple power analysis (SPA) and differential power analysis (DPA).

Simple Power Analysis. This is the analysis of one, or the comparison of a few, power consumption traces. An attacker will look for repetitive patterns and distinctive events to reverse engineer the algorithm being used and to try and derive any secrets being manipulated. Figure I.2 shows a power consumption trace taken during the execution of an AES implementation, using equipment similar to that shown in Figure I.1, where the 10 rounds of the DES algorithm can be observed. A pattern that repeats 9 times can be seen, with a much smaller tenth pattern, that correspond to the rounds of an AES.

Much smaller differences can be seen in the power consumption. By zooming in on specific areas the difference of 1 or more clock cycles can be used to identify secret information. For example, A compiler can implement the bitwise permutations used in DES in an insecure manner. An example of an algorithm that could be produced is given in Algorithm I.1, where each bit in buffer X (a buffer of n bits, where $(x_0, x_1, \ldots, x_{n-1})_2$ is the binary representation) is tested to determine whether a bit

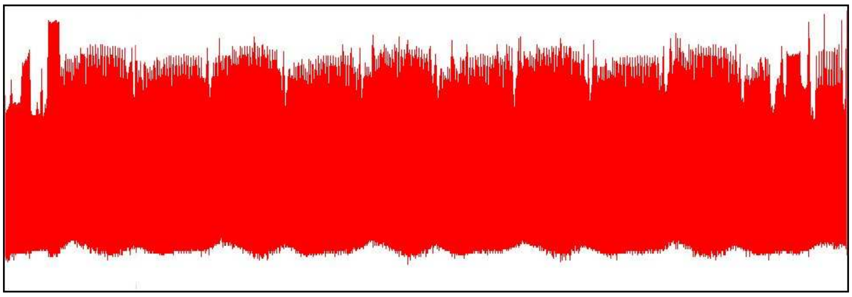

Figure I.2. Power consumption during the execution of an AES implementation.

in buffer Y should be set.

Algorithm I.1 Unsecure bitwise permutation function

```
Input: $X = (x_0, x_1, \dots, x_{n-1})_2$, $P[\cdot]$ containing the indexes of the permutation
Output: $Y = (y_0, y_1, \dots, y_{n-1})_2$
1.  $Y := 0$;
2.  for $i = 0$ to $n - 1$
3.    if $(x_{P[i]} = 1)$ then $y_i := 1$;
4.  return $Y$;
end.
```

This is not a secure implementation due to the conditional test present in step 3. The setting of one bit in buffer Y will take a certain amount of time when the tested bit in buffer X is equal to 1. If a power consumption trace for this function is compared to a trace taken during the manipulation of a known key, for example all zeros, the point where the two traces differ will reveal the first manipulated bit that is not equal to the known key. In the case where the known key is all zeros, the power consumptions diverge where the manipulated bit of the unknown key is equal to one. All previous bits were therefore equal to 0. The power consumption trace can then be shifted to take into account the time difference due to this bit and the process repeated for the rest of the key.

This problem can be avoided by implementing the bitwise permutation as shown in Algorithm I.2, where each bit is taken separately for buffer X and assigned to buffer Y. This is usually more time consuming than Algorithm I.1 as each bit needs to taken from the relevant machine word.

Algorithm I.2 Secure bitwise permutation function

```
Input: $X = (x_0, x_1, \dots, x_{n-1})_2$, $P[\cdot]$ containing the indexes of the permutation
Output: $Y = (y_0, y_1, \dots, y_{n-1})_2$
1.  for $i = 0$ to $n - 1$
2.    $y_i := x_{P[i]}$;
3.  return $Y$;
end.
```

It is not always possible to change an algorithm in this manner. For example,

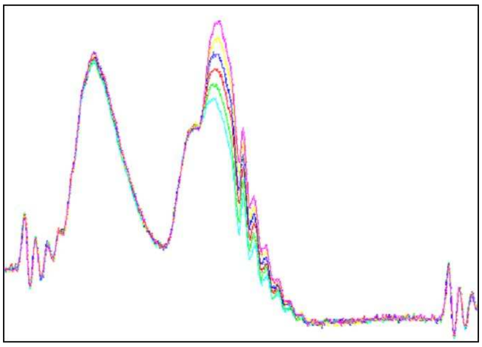

Figure I.3. Superimposed acquisitions of one clock cycle showing the data dependence of the power consumption.

the modular exponentiation used in RSA is exceedingly difficult to secure against this type of attack, as branches in the algorithm cannot be avoided. An example of a power consumption trace that shows the square-and-multiply algorithm is given in [40].

Differential Power Analysis. In this type of attack the smart card is forced to run the same command numerous times with different inputs. For each input the corresponding power consumption is captured and an analysis is performed on the data gathered to make inferences on the secret information held inside the card. These techniques require many more acquisitions than with simple power analysis and a certain amount of treatment *a posteriori*.

Differential power attacks [43] are based on the relationship between the current consumption of the hamming weight of the data manipulated at a particular point in time. The variation in current consumption is extremely small and acquisitions cannot be interpreted individually as the information is very small when compared to the usual amount of noise in a given acquisition. These differences can be seen in Figure I.3, and can be amplified by using DPA.

If two average curves are produced (one where the bit is equal to one and the other where it is not), the effect of the modified bit on the current consumption becomes visible in the difference between the two averages, an example of this is shown in Figure I.4. A peak in this waveform will occur at the moment the bit used to divide the acquisitions into two sets was manipulated. This occurs because the rest of the data can be assumed to be random and will therefore provide the same average. Figure I.4 shows a sample DPA trace showing five different points in time where the bit used to divide the acquisitions is manipulated.

This can be used to break a secret key algorithm if an attacker forms a hypothesis about the exit of a given function e.g. an S-box. This technique can be used to verify the hypothesis as peaks, as shown in Figure I.4, will be present. This would appear to be analogous to an exhaustive search but the output of 1 S-box in DES depends on 6 bits of the secret key. An attacker therefore only needs to test hypotheses on these 6 bits to determine what they are. The process could then be repeated for

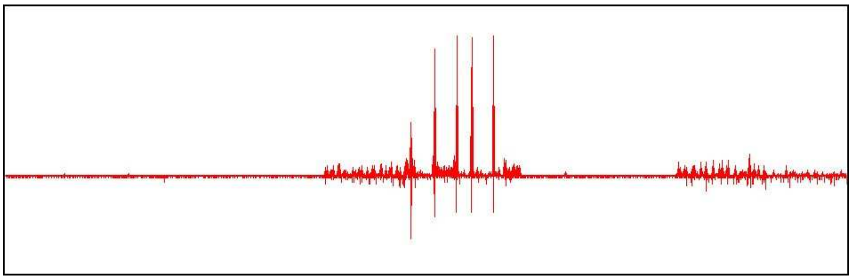

Figure I.4. A DPA trace.

each S-box to derive the entire key.

A method of improving this method to reduce the existence of false positives is given in [14].

Countermeasures. There are several different countermeasures for protecting algorithms against attacks based on the power consumption, although some of the countermeasures merely complicate the attack process. The combination of all the countermeasures below makes attacking a secure algorithm exceedingly difficult, as each countermeasure needs to be taken into account.

Constant Execution: As described above, the time taken by an algorithm should remain constant, so that no deductions on secret information can be made. This extends to individual processes being executed by a smart card. If a process takes different lengths of time depending on some secret information and the difference in time is made up by a dummy function, there is a good chance that this will be visible in the power consumption as detailed in Section I.2.1. It is therefore important that an algorithm is written so that the same code is executed for all the possible values that the secret information being manipulated can take.

Random Delays can be inserted at different points in the algorithm being executed i.e. a dummy function that takes a random amount of time to execute can be called. The algorithm can no longer be said to comply with the constant execution criteria given above, but any variation is completely independent to any secret information. This does not provide a countermeasure, but creates an extra step for an attacker. In order to conduct any power analysis an attacker needs to synchronise the power consumption acquisitions *a posteriori*. The effect of conducting statistical power analysis attacks in the presence of random delays is detailed in [18].

Randomisation (or data whitening) is where the data is manipulated in such a way that the value present in memory is always masked with the same random. This randomisation remains constant for one execution, but will vary from one acquisition to another. This mask is then removed at the end of the algorithm to produce the ciphertext. Some ideas for building countermeasures were proposed in [15], and an example of this sort of implementation can be found in [3].

The size of the random is generally limited as S-boxes need to be randomised before the execution of the random so that the input and output values of the S-box leak no information. This is done using an algorithm such as Algorithm I.3. As shown the random used for masking the input data can be no larger than n, and the random used for the output value can be no larger that x.

Algorithm I.3 Randomising S-box Values

Input: $S = (s_0, s_1, s_2, \ldots, s_n)_x$ containing the S-box, $\mathbf{R}$ a random $\in [0, n]$, and r a random $\in [0, x)$

Output: $RS = (rs_0, rs_1, rs_2, \ldots, rs_n)_x$ containing the randomised S-box

```
1.  for $i = 0$ to $n$
2.      $rs_i := s_{(i \oplus \mathbf{R})} \oplus r$;
3.  return $RS$;
end.
```

In the case of AES both $\mathbf{R}$ and r will be one byte, which means that the random mask during the calculation is likely to be one byte. There are theoretical attacks against this protection method [52] but an actual implementation has yet to be published.

This is not possible in the case of RSA, where the calculation methods do not facilitate the method described above. A method for randomising the calculation of RSA is given in [40], where the signature generation (i.e. $s = m^d \pmod{n}$ where $n = p \times q$) can be replaced by:

$$s = \Big((m + r_1 n)^{d + r_2 \phi(n)} \pmod{r_3 n} \Big) \pmod{n} \qquad \text{(I.1)}$$

where $\phi(\cdot)$ is Euler's totient function and, r_1, r_2 and r_3 are small random values. This does not provide a totally secure algorithm as the modular exponentiation itself also has to be secured against SPA attacks. A discussion of these algorithms is given in [17].

Randomised Execution is the manipulation of data in a random order so that an attacker does not know what is being manipulated. In the case of Algorithm I.1 an attacker would not know which bit is being manipulated at any given point in time. Given the $n!$ possible combinations, an attack by simple power analysis would be extremely difficult.

This also inhibits any statistical analysis of the power consumption, as this relies on the same unknown variable being treated at the same point in time. As an attacker cannot know the order in which the data has been treated, this provides an extremely efficient countermeasure when combined with randomisation. An example of this technique applied to DES is described in [51].

I.2.2 Fault Attacks

The first example of a theoretical fault attack was presented in 1997 [12] as a method of injecting faults in to RSA signature generation when using the Chinese Remainder Theorem. This was followed by several other attacks on public key algorithms [5, 41] and an equivalent for private key algorithms was proposed in [9], usually based on the effect of a 1 bit error during the computation of the algorithm under study. As no implementations were forthcoming interest in this type of attack waned. One of the first publications describing an implementation of this type of attack was presented in 2002 [4], and described an implementation of the attack against the RSA signature scheme and some countermeasures. This revitalised interest in this type of attack, which is now an important aspect of smart card security.

Modelling the Effect of a Fault. There are several known mechanisms for injecting faults into microcontrollers. These include variations in the power supply to create a glitch or spike [11], white light [63], laser light [6] and eddy currents [60]. The models for the faults that can be created by these effects can be summarised as follows:

Data randomisation: the adversary could change an arbitrary amount of data to a random value. However, the adversary does not control the random value and the new value of the data is unknown to the adversary.

Resetting Data: the adversary could force the data to the blank state, i.e., reset a given byte, or bytes, of data back to `0x00` or `0xFF`, depending on the logical representation.

Modifying opcodes: the adversary could change the instructions executed by the chip's CPU. This will often have the same effect as the previous two types of attack. Additional effects could include removal of functions or the breaking of loops. The previous two models are algorithm dependent, whereas the changing of opcodes is implementation dependent.

These three types of attack cover everything that an attacker could hope to do to an algorithm. In most cases it is not usually possible for an attacker to create all of these possible faults. Nevertheless, it is important that algorithms are able to tolerate all types of fault, as the fault injection methods that may be realisable on a given platform are unpredictable. While an attacker might only ever have a subset of the above attacks available, if that attack is not taken into account it may have catastrophic consequences the security of a protocol or algorithm.

All of these models assume that a fault injected into a chip will change a machine word rather than a single bit. This model was based on studies of faults caused by the effects of the radiation present in the upper atmosphere [75], which is important in the design of vehicles designed to travel in the upper atmosphere or space. The effects of radiation are random, whereas an injected fault is intentional and will target a given point in time.

There have not been any publications that detail practical implementations of attacks that exploit the change of 1 bit of information in a chip. It is possible to use a lasers simulate the effect of radiation in a circuit [38], but smart cards will generally use scrambled/randomised layouts so the possibilities are limited. More success has been achieved by targeting larger areas to provoke a large fault.

Injecting Faults in Algorithms. In this section two of the most widely known attacks will be described. The first fault attack published in [12] and implemented in [4], which is detailed below:

If we assume that the calculation of an RSA signature i.e. $s = m^d \pmod{n}$, where $n = p \times q$, the two primes upon which the security of RSA is based. If this is calculated using the Chinese remainder theorem the following values are calculated,

$$\begin{aligned} s_p &= m^{(d \pmod{p-1})} \pmod{p} \\ s_q &= m^{(d \pmod{q-1})} \pmod{q} \end{aligned} \tag{I.2}$$

which can be combined to form the RSA signature s using the formula $s = a \times s_p + b \times s_q \pmod{N}$, where:

$$\begin{cases} a \equiv 1 \pmod{p} \\ a \equiv 0 \pmod{q} \end{cases} \text{and} \begin{cases} b \equiv 0 \pmod{p} \\ b \equiv 1 \pmod{q} \end{cases}$$

This can be calculated using the following formula:

$$s = s_q + \Big((s_p - s_q) \times q^{-1} \pmod{p}\Big) \times q \tag{I.3}$$

If this is calculated correctly, and then recalculated but with a fault injected during the computation of s_p or s_q, information can be derived on one of the primes used to create n.

If s_q is changed to s'_q, then the signature generated will be of the form $s' = a \times s_p + b \times s'_q \pmod{n}$. The difference between s and s' gives:

$$\begin{aligned} \Delta &\equiv s - s' \\ &\equiv (a \times s_p + b \times s_q) - (a \times s_p + b \times s'_q) \\ &\equiv b(s_q - s'_q) \pmod{n} \end{aligned} \tag{I.4}$$

As $b \equiv 0 \pmod{p}$ and $b \equiv 1 \pmod{q}$ it follows that $\Delta \equiv 0 \pmod{p}$ (but $\Delta \not\equiv 0 \pmod{q}$) meaning that Δ is a multiple of p (but not of q). Hence, a GCD calculation gives the secret factors of n, i.e. $p = \gcd(\Delta \pmod{n}, n)$ and $q = n/p$.

This attack could potentially be achieved with any of the fault models given above. This is because any fault in one of the modular exponentiations given in Equations I.2 will be enough to conduct the attack. This includes modifying the variable entering the equation before the start of the computation of the modular exponentiation.

Another attack that was proposed shortly after the RSA attack was published targeting DES [9]. This assumed a 1-bit fault injected into the last few of rounds of a DES implementation. This was generalised to allow for a larger fault and became

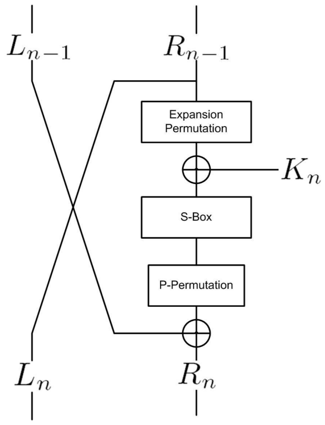

Figure I.5. The DES round function for round n.

a frequently cited attack within the smart card industry. The most popular form of this attack is described in [6], and is repeated here.

DES can be considered as a transformation of two 32 bit variables (L_0, R_0), i.e. the message, though sixteen iterations of the function as shown in Figure I.5 to produce the ciphertext (L_{16}, R_{16}). The Expansion and P permutations are bitwise permutations, and are generally not considered when studying DES. This means that the round function can be simplified to:

$$\begin{aligned} R_n &= S(R_{n-1} \oplus K_n) \oplus L_{n-1} \\ L_n &= R_{n-1} \end{aligned} \tag{I.5}$$

where $S(\cdot)$ is the S-box function. The existence of the initial and final permutation is also ignored as they do not contribute to the security of the algorithm.

The last round, as described above, can therefore be expressed in the following manner:

$$\begin{aligned} R_{16} &= S(R_{15} \oplus K_{16}) \oplus L_{15} \\ &= S(L_{16} \oplus K_{16}) \oplus L_{15} \end{aligned} \tag{I.6}$$

If a fault occurs during the execution of the fifteenth round, i.e. R_{15} is randomised by a fault to become R'_{15}, then:

$$\begin{aligned} R'_{16} &= S(R'_{15} \oplus K_{16}) \oplus L_{15} \\ &= S(L'_{16} \oplus K_{16}) \oplus L_{15} \end{aligned} \tag{I.7}$$

If we XOR R_{16} and R'_{16} we get:

$$\begin{aligned} R_{16} \oplus R'_{16} &= S(R_{15} \oplus K_{16}) \oplus L_{15} \oplus S(R'_{15} \oplus K_{16}) \oplus L_{15} \\ &= S(R_{15} \oplus K_{16}) \oplus S(R'_{15} \oplus K_{16}) \\ &= S(L_{16} \oplus K_{16}) \oplus S(L'_{16} \oplus K_{16}) \end{aligned} \tag{I.8}$$

This provides an equation where only the last subkey, K_{16}, is unknown. All of the other variables are visible in the ciphertext. This equation holds for each S-box in the last round, which means it is possible to search for key hypotheses in sets of six bits. All 64 possible key values corresponding to the XOR just before each S-box are exhausted to generate a list of possible key values for these key bits. After this, all the possible combinations of the hypotheses can be searched though with the extra 8 key bits that are not included in the key to find the entire key.

If R'_{15} becomes random then the expected number of hypotheses that are generated can be predicted. For a given input and output difference there are certain number of values that could create the pair of differences, as described in [8]. The expected number of hypotheses for the last subkey will be around 2^{24}, giving an overall expected keyspace of 2^{32}.

If the attack is repeated to acquire two faulty ciphertexts the intersection of the two keyspaces can be taken. This will greatly reduce the keyspace that will need to be searched through to derive the key. It would be expected that two faulty ciphertexts with the properties described above would give around 2^6 hypotheses for the last subkey, leading to an exhaustive search of around 2^{14} for the entire key.

An implementation of this is described in [30] where two faulty ciphertexts are acquired, leading to a small exhaustive search to find the DES key used. Again, it would be expected that any of the fault models given would enable this attack to be conducted. As any fault during the calculation of R_{15} will give a ciphertexts with the correct properties.

This attack is also extended in [30] to allow for any faults from the eleventh round onwards, the details of which are beyond the scope of this article. Another attack [32] attempts an attack of a similar nature but using faults at the beginning of the DES algorithm.

This gives two different attacks against commonly used algorithms. There are a plethora of other attacks that target other algorithms, but they cannot be listed here. These two attacks were chosen to represent the fault attacks that have been proposed and implemented.

Countermeasures. There are various types of countermeasure that can be implemented to defend against fault attacks. These are usually based on existing techniques used for integrity purposes. It would normally be expected that anomaly sensors would be implemented in a smart card that would detect an attempted fault attack. However, this cannot be relied upon as a new fault injection technique may be able to circumvent these.

A detailed list of the possible fault resistant hardware implementations is given in [6]. Some of the software countermeasures that can be used are listed below.

Checksums can be used to verify the integrity of data at all times. This is especially important when data is being moved from one memory area to another. For example, a fault injected in an RSA variable can compromise the secret key, as described above.

Variable redundancy is the reproduction of a variable in memory and functions are performed on each variable independently. The result will be known to be correct if both answers are identical.

Execution redundancy is the repetition of the same function, part of the function or it's inverse. In the case of the DES algorithm it would be prudent to repeat the first 3 rounds to protect against the attack described in [32], and the last 5 rounds to protect against the attack described in [9, 30]. This greatly increases the execution time but is less costly than repeating the entire algorithm.

In the case of RSA the simplest solution to protect the signature generation function is to verify the result with the public key. This is efficient as signature verification is extremely fast when compared to signature generation. In some standards it is not always possible to have access to the public key, countermeasures have therefore been proposed that verify certain conditions after signature generation has taken place. An example of this type of countermeasure, and an account of previous methods, is given in [16].

Execution Randomisation: If all the functions are conducted in a random order it is not be possible to determine exactly where a fault needs to be injected. The presents a similar problem to that described in Section I.2.1, as an attacker is unsure of what function is being attacked. However, this merely slows an attacker as an attack can be repeated until successful. An example of this type of countermeasure on the context of fault attacks is given in [55].

Ratification counters and baits: A countermeasure described in [6] involves including small functions in sensitive code that perform a calculation and then verify the result. When an incorrect result is detected a fault in known to have been injected. The reaction to such an event would be to decrement a counter. When this counter reaches 0 the smart card would then cease to function. When combined with random delays, as described in Section I.2.1, this can be a very effective countermeasure.

In order to achieve a secure implementation the above countermeasures would need to be combined with those presented in Section I.2.1. This implies a significant overhead when implementing cryptographic algorithms for smart cards but is necessary to defend against modern attack techniques.

I.3 Smart Card Platform Security

The aim of this section is to provide an overview of the issues, apart from the underlying hardware functionality, affecting the overall concept of smart card security. Therefore, it will explore the functionality offered by multi-application smart card platforms and it will also highlight the main security concerns behind them.

I.3.1 The Evolution of Smart Card Platforms

The evolution of smart card operating systems (SCOS) and platforms is very closely coupled with the various improvements of the underlying smart card hardware. A typical smart card of the mid 90's had approximately 1-3 Kbytes of ROM, less than 128 bytes of RAM and 3-6 Kbytes of Electrically Erasable Programmable Read Only

Memory (EEPROM). Within this very restrictive processing environment the role of the smart card operating system was naturally limited.

In most cases, the role of these early smart card operating systems (i.e. first generation) was to mainly offer the basic functionality defined within the various smart card standards [35, 36, 37] and, more specifically, to present a protected file system. This was because as smart card microprocessors were mainly perceived as secure storage devices. The main drawback of these operating systems was that the smart card applications developers had very little flexibility in terms of developing a smart card application, as the functionality offered was mostly implemented in the Read Only Memory (ROM) of the card. Although, some of these operating systems [29, 56] claimed that they offered multi-application smart card functionality, the reality was different.

The next generation (i.e. second generation) of smart card operating systems, e.g. Multos [47] emerged between 1995 and 1999 with the introduction of more powerful microprocessors consisting of 6-8 Kbytes of ROM, 128 bytes of RAM and 6-12 Kbytes of EEPROM. The smart card microprocessors were capable of hosting what it often referred to as the "monolithic" [34, 48] smart card operating systems. The main characteristic of the "monolithic" operating system was that its functionality was enhanced compared to the previous generation due to a greater underlying set of commands. Smart card applications and operating systems were still very closely coupled together within the ROM memory of the card and, although they were more advanced, portability was a major concern.

The third generation of smart card operating systems came into existence from approximately 1999 onwards. The market requirements were more mature in terms of application interoperability, hardware independence, and post issuance capabilities. Among the main characteristics of these operating systems was that applications and operating system functionality were written to the EEPROM and ROM respectively, see Figure I.6. Moreover, the presence of a Hardware Abstraction Layer (HAL) ensured that application portability should no longer be an issue.

I.3.2 The Different Multi-application smart card Platforms

Among the main characteristics of secure Multi-application smart card platforms was application interoperability and inter application isolation. Smart card issuers would like to be able to develop smart card applications that would be interoperable irrespectively of the underlying smart card hardware. Similarly, they were looking for an underlying platform that would be able securely isolate one application from another by performing efficient memory management.

The two principal Multi-application smart card standards are Java Card [21, 67, 68, 69] and MULTOS [47]. However, there are two other technologies namely the Smartcard.NET [39, 64] of Hive-Minded and Multi-application BasicCard ZC6.5 [74] of ZeitControl. The main Multi-application smart card platforms are presented below, excluding the Java Card that will be discussed in the next section.

WfSC. Around 1998, Microsoft launched a new business proposition under the name of Windows for Smart Cards (WfSC). The whole concept was defined in [45],

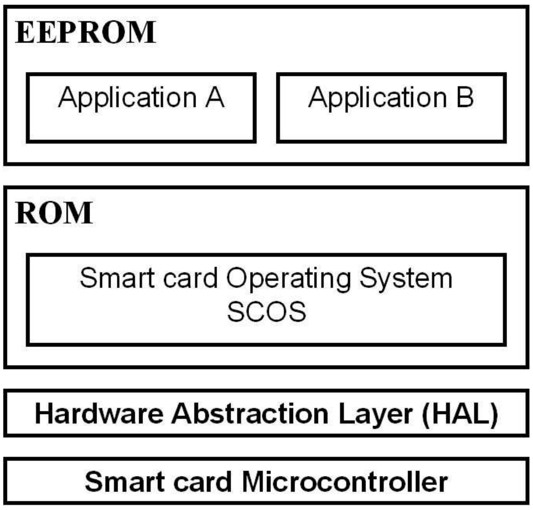
EEPROM
Application A
Application B
ROM
Smart card Operating System
SCOS
Hardware Abstraction Layer (HAL)
Smart card Microcontroller

Figure I.6. The typical architecture of a third generation SCOS.

where the underlying platform was capable of performing dynamic application management and the applications were developed in Visual Basic (VB) or C. The philosophy of the proposed architecture [61] was that smart card programming should become part of mainstream programming rather than to force programmers to acquire new competences and new models of programming. In fact, under this framework a WfSC project was simply another type of Visual BASIC or Visual C++ project. A couple of years afterwards, Microsoft dismantled the WfSC team as it became evident that it was no longer directly supporting it. The most notable observation about WfSC was that the overall security functionality was reviewed and guaranteed by Microsoft, which, along with the fact that it was a “closed” system, created barriers in terms of the technology adoption.

Multos. The Multos [47] smart card operating system was specifically designed for smart card microprocessors. Security was taken into serious consideration right from its conceptualisation. An industrial consortium, named MAOSCO, was created in order to promote MULTOS as a multi-application operating system for smart cards, to manage the specifications MULTOS, and to provide the licences and certifications of MULTOS services. Multos is an operating system that also offers dynamic application management and secure application isolation right at the operating system kernel. Some of the key elements of the MULTOS platform are:

- a highly protected architecture;
- the possibility of having several applications on the same card;
- independence of the applications compared to the underlying platform;
- compatibility with industrial standards such ISO 7816 and EMV [12].

Applications can be developed in a language optimised for smart cards: MEL (Multos Executable Language) based on the Pascal P-codes and the virtual machine described in [72]. From a security point of view, Multos achieved one of the highest certified security levels (i.e. EL6 of ITSEC [44] which corresponds to EAL7 of Common Criteria [20] security evaluations) ever achieved by a commercial product. Although this provided some security reassurance it also increased the overall cost of the platform.

The ZeitControl BasicCard. The concept of BasicCard came into existence around 1996 with the first card being released in 2004, i.e. MultiApplication BasicCard ZC6.5 [74]. The BasicCard was programmed in ZeitControl BASIC and it also contained a virtual machine. The main difference to the other card was that it was the only one capable of supporting floating numbers [61]. Among the main advantage of the Basic card was that it was a low cost card and it offered a relatively good starting point for smart card programmers

I.3.3 Java Card

Java was among the very first programming languages that took security into account and, at the same time, enjoyed wider acceptance. The general Java language can be described as an "object-oriented (with single inheritance), statically typed, multithreaded, dynamically linked and has automatic garbage collection [where] application developers can use a number of Application Programmers Interfaces (API) when developing new applications" [27]. It is generally accepted that the Java programming language provides a secure model that prevents programs from gaining unauthorised access to sensitive information. At the same time, since Java is based on a runtime byte code interpreter, the portability issue is successfully addressed.

Although Java Card may be perceived as another flavour of traditional Java, the reality is different. The first Java Card API (version 1.0) [65] was released in October 1996 as the initial attempt to bring the benefits of the Java language into the smart card world. Since then, with major enhancements taking place around 1999 with the release of the Java Card API version 2.0 [66], further versions were released incorporating major improvements. Java Card preserves the object-oriented scope of the Java programming language, but it is essentially a subset of normal Java and does not support such things as: Dynamic class loading, security manager, object cloning, threads, and large primitive data types.

The internal architecture of the Java Card specification is illustrated on the right hand side of Figure I.7. At the bottom layer of the smart card architecture is the hardware (i.e. the actual smart card microprocessor). In the next layer, we encounter the smart card operating system (SCOS). On top of the SCOS, we have the Java Card Virtual Machine (VM). Both the Java Card VM and the SCOS are written in the native language of the microprocessor. This abstraction hides the manufacturer's proprietary technology with a common language interface.

The steps for creating a Java application (the so called applet), download it and execute it on the smart card are the following: First of all, the application

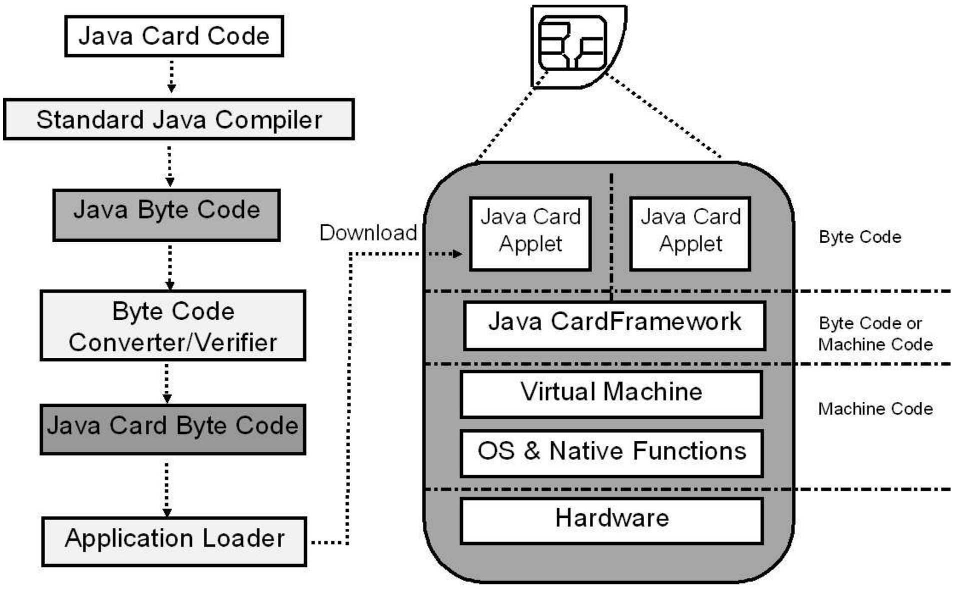

Figure I.7. The Java Card application development cycle and the Java Card architecture.

programmer must take into account the Java Card APIs and develop an application by using a standard Java development environment. Since there is rarely a byte code verifier in the card, the newly created Java code classes should be verified externally. This implies that the Java Card VM may rely on a digital signature before accepting code from an external source. An external converter is also utilised in order to reduce the size of the actual application. As soon as the Java classes are verified, the application code is ready for loading onto the card via the external application loader. The whole procedure is summarised on the left hand side of Figure I.7.

The Java Card platform security feature are inherited from the Java language and several enhancements are also provided through the Java Card framework and run-time environment. The fact that certain Java functionality was removed from the Java Card framework in order to ensure that the technology is ported to smart cards introduced some potential security problems.

The discussion around the security issues introduced is very closely coupled with the security functionality offered by the underlying platform. Therefore, in the next few sections the reader will be presented with the differences and additional functionality introduced by the Java Card platform in relation to the most notable Java Card attacks and vulnerabilities.

I.3.4 Java Card Security

Although there are many attacks directed against the Java Card platform, it is not possible to provide their details here, as they have been explained extensively in the academic literature. However, our discussion on the Java Card risks will also include particular references to the differences with traditional Java that remove certain security problems.

The lack of dynamic class loading within Java Card makes the overall static code verification process simpler. However, it is not possible to guarantee the type safety of classes that are about to be loaded. Furthermore, the fact that only a single Java Card application can be active and executing at any point in time (single threading) further simplifies the overall Java Card security model. Java Card has been extensively criticised in that the programming functionality is seriously restricted within the Java Card API, which can be particularly restraining when new powerful methods need to be implemented, e.g. cryptographic operations. However, the existence of a mechanism that will bypass the Java language syntax and type safety features could allow a programmer to execute arbitrary code, which would have been a major concern for smart card vendors and issuers as it could create serious access violations. Among the most notable properties related to the Java Card security model are the following [49, 73]:

- Garbage collection
- CAP verification
- Applet loading
- The firewall mechanism
- Transaction atomicity
- Native methods
- Exceptions

The lack of a garbage collection mechanism, which is an optional feature, can be a major concern. The idea in normal Java is that unused space is collected and reused. This process is achieved though the existence of a complex mechanism. The so-called "garbage collector" is a relatively large in terms of code size and it was therefore not possible to be defined as a mandatory object within the Java Card specifications. This opens the opportunity to create a malicious Java Card applet that leaks memory resulting in a denial-of-service attack.

The CAP file is a representation of the original Java binary file, modified due to the limited resources available to a given smart card. It is stated in [73] that since the CAP file is restricted in size, it has also lost some of the inherent Java language security features. It is considered to be possible to modify the CAP file and change it in a way that will be able to threaten the security of the underlying platform. Therefore, it is very important to be able to verify that the CAP file adheres to the Java Card security rules. Although Java Card developers initially believed that it was impossible to include a Java Card verifier on a Java card, due to the various resource constraints. Over the last few years on-card Java verifiers have been developed [19, 46, 59]. Some of these solutions "allow to bypass the signature step of the application without jeopardising the card security". However, the existence of a defensive VM that will dynamically execute the byte codes may be preferable to a stand alone verifier will statically check during the load time [61].

Application downloads, especially at post issuance, lies within the core concept of a multi-application smart card. If applications can be added deleted or modified at post issuance, then the issuers are presented with a relatively powerful and flexible platform that will meet their future requirements. However, the existence of an application downloading mechanism within the Java Card API is not defined. The Global Platform [31] specification comes into play, in order to handle the security sensitive operation of post issuance application downloading. The details of the Global Platform specification are widely known and beyond the scope of this analysis. The Global Platform specification is an important element of Java Card security as it takes care of further security risks. For example, it strengthens the applet firewall (to be explained below) by the existence of a card manager. The Global Platform card manager is responsible, among others, for ensuring that only trusted code (bearing the appropriate credentials) can be downloaded to the card. Therefore, the existence of trusted applications (that have been previously verified) facilitates the existence of an elaborate application sandbox. This in turn requires the existence of thorough application testing and verification process.

The Java Card firewall mechanism is responsible for isolating applications from each other. It ensures that no application is allowed to access another application or its data in an authorised way. The firewall is considered of paramount importance in a multi-application smart card environment, as there is always the risk of inter-application attacks. By default the Java Card [73] applications are only allowed to access data objects that they own. The mapping between what is owned by an application is done during the application downloading and installation process. It is obvious that this process requires the existence of memory protection mechanisms that will perform the necessary checks. The role of the firewall is also to control authorised data sharing. In terms of firewall security there are plenty of attacks that have been described in the literature. “Two attacks against the firewall mechanism (AID) impersonation and illegal reference casting [54] that provides access to all interface methods of a class, along with other attacks originating from problems in the specifications [73]” [22]. A further type of attacks is based on “type confusion” and it is further described in [7].

Transaction atomicity is a further important concept that aims to improve the overall stability of the Java Card platform. Smart cards are often operating in environments in which a user can remove a smart card from its reader at any point in time. In all cases, smart cards, and particularly Java cards, should be in a position to recover when the normal operation is disrupted, e.g. when power is lost. Transaction atomicity guarantees that any updates to a single persistent object or class will be atomic (i.e. either not performed at all or fully performed). Automatic transaction integrity describes how the virtual machine behaves when power is lost during the update of a field in an object. On the other hand, block transaction integrity describes the virtual machine behaviour when the application programmer identified a specific part of his code that needs to be executed in one piece. For example consider a code example presented in [70] ($CommitBuffer.begin()$, ATC++,

ATC++, *CommitBuffer.commit*()). The ATC is a simple Application Transaction Counter. If such a block is defined (*CommitBuffer.begin*()) and not ended with (*CommitBuffer.commit*()) an automatic rollback is performed at the end of the application execution or when card is powered back on.

Native methods are not allowed by the Java Card specifications. However, Java Card application developers would love to be able to access specific calls that will allow them to perform low level tasks, e.g. by obtaining direct access to the any cryptographic coprocessors. As mentioned before, implementing some of the Java Card API functionality requires using native methods. The existence of native methods will completely violate the language based encapsulation model [50] and this one of the main reasons that smart card vendors decided not to offer such functionality. The existence of native methods would also compromise portability.

Exceptions are thrown by the Java Card VM (when internal runtime problems are encountered) or they can be thrown programmatically (Checked exceptions, Unchecked exceptions). The Java Card VM catches the exceptions that are not caught by the application. The Java Card application programmers can define their own exceptions by declaring subclasses of the class Exception. The wide utilisation of exceptions is another interesting topic within Java card. It is mentioned in [49] that unhandled exceptions may cause denial of service attacks. The significance of exception handling was recognised in the Java Card API version 2.1 when the "Exception" class was redesigned.

Java Card CAP file reverse engineering is a potentially serious type attack. It requires obtaining access to a CAP file and running it through a de-compiler, to find out as much information as possible about the application. The next step requires importing certain changes and minor modifications, in order to change the behaviour of the application. However, although minor changes in the CAP file create further changes in the complete file, it is possible to perform pattern changes [23] that will reveal certain information of the application execution.

I.4 GSM and 3G Security

Smart cards play an important role in mobile telecommunications where their security functionality combined with tamper-resistance, underpins the control and viability of the industry. However this was not always the case and the use of smart cards in 2G & 3G systems (such as GSM and UMTS) is largely due to problems that occurred in early systems. The lessons learned form the evolution of the security in mobile phones have played a large part in defining the security issues discussed in Sections I.2 and I.3.

I.4.1 1G - TACS

Total Access Communication System (TACS) - was an analog cellular phone systems working at 900MHz and initially deployed within various European countries including UK, Italy, Austria and Spain. TACS was derived from the US cellular system known as Advanced Mobile Phone Service (AMPS). The first commercial TACS system was deployed in the UK and was put into service by 1985.

Early mobile phone systems such as TACS started life using analogue technology and handsets that did not contain smart cards. There was however a security solution embedded in the handsets and network, designed to ensure that only valid users were allowed access to communications services and that they were billed appropriately. This solution was limited to some extent by the available technology but it is probably fair to say that the technical emphasis of the day was primarily to get the radio system to work rather than on its security.

When an analogue phone was switched on, or made a call request, it transmitted (in clear) two important pieces of information; the SNB and the ESN. The SNB was the Subscriber Number (telephone number) and the ESN was the handset Electronic Serial Number. When the network received this information it checked that the transmitted ESN matched the stored value within the network for that particular SNB. In theory the ESN was fixed per phone and the SNB could only be changed by the operator, however many phones were easily tampered with, meaning that both fields could be reprogrammed to create clones.

Typically, an attacker would purchase a radio scanner to monitor the frequencies used by the analogue networks and set up near to an area where analogue phones were likely to be used e.g. motorway services or airport. ESN-SNB pairs could then be captured and this information programmed into another handset. The network would then permit the use of the handset and the charges for the calls made on the "cloned" phone would appear on the legitimate customer's bill. The end result was a major headache for the mobile operators who not only lost revenue because of cloning but also from customers making false claims of cloning to dispute their bills. By 1995 the problems in the UK had become so severe that Members of Parliament were briefed on the issues [57].

I.4.2 2G - GSM

The industry reaction was to ensure that the next generation of digital mobile communication system (GSM) being standardised by ETSI [24] would be designed to avoid clear transmissions of security keys and not be reliant on the discredited tamper-resistant capabilities of handsets. The answer was the introduction of smart cards in the form of Subscriber Identity Modules (SIMs) that are described in detail by GSM 11.11 [25]. From an access and security viewpoint the SIM had to provide integrity, authentication and help ensure confidentiality.

The identity that is important to the GSM network is the International Mobile Subscriber Identity (IMSI) stored in the SIM. It is not the users telephone number but a unique number associated with a customer account. It cannot be altered but is otherwise not very well concealed, as when a user first arrives on a network the IMSI is transmitted in clear over the radio interface. Thereafter a Temporary Mobile Subscriber Identity (TMSI) is allocated and used but this just adds a little extra inconvenience for the attacker. The IMSI can also be read by putting a SIM card in a smart card reader (providing it is not PIN-locked).

The stronger authentication security comes from an algorithm and a 128 bit secret key (Ki) that is stored in the SIM and mimicked in a network system called

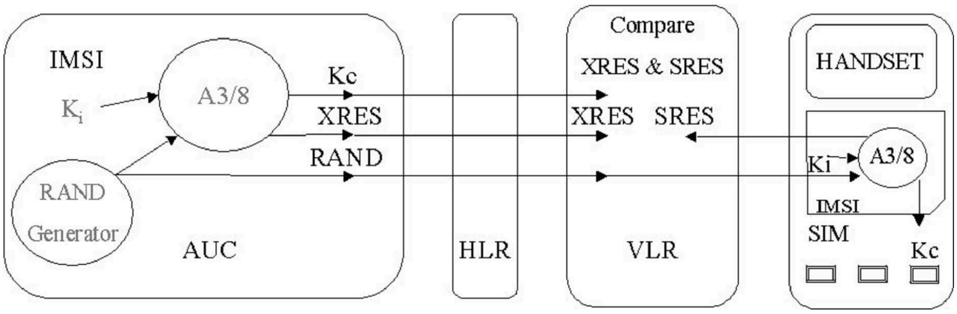

Figure I.8. GSM authentication.

the Authentication Centre (AuC). Logically the AuC is part of the Home Location Register (HLR) that holds a mapping of all the IMSIs and Kis. The AuC generates the necessary challenges and expected responses to ensure that the SIM card is genuine. The algorithm is sometimes referred to as A3/8 and is in fact two algorithms, one to produce the authentication result (A3) and the other (A8) to produce a radio cipher key. A basic diagram is shown in Figure I.8.

When the user wishes to access the GSM network it is not sufficient just to present the IMSI, but also to prove that a valid SIM card is present. This works as detailed in Algorithm I.4.

Algorithm I.4 Accessing the GSM network

```
1. The AuC Generates a 128 bit random number (RAND).
2. The AuC runs the algorithm which uses RAND and secret key (Ki) as inputs.
3. The expected result is produced (XRES) plus a cipher key (Kc) for encrypting radio transmissions.
4. RAND, XRES and Kc. The triplet is handed to a Visitor Location Register (VLR) that is in the same location area as the user.
5. RAND is passed to the SIM via the radio network and the handset.
6. The SIM runs its copy of the algorithm and generates a result (SRES) and a cipher key (Kc).
7. SRES is sent back to the VLR.
8. If SRES equals XRES the user is authenticated.
9. Kc is then used for ciphering subsequent communications.
end.
```

This introduction of the tamper-resistant SIM solution was a huge leap forward and overcome the problems so prevalent in the analogue phone system i.e. it prevented simple cloning and offered some confidentiality on the radio interface. History has shown that the SIM has done a pretty good job at maintaining mobile communications over a period of many years, although cannot be claimed to be perfect and its weaknesses have been well publicised. Like all smart cards it can be subject to the security attacks described in earlier sections, but the smart card industry has been vigilant in this respect and modern cards implement robust good countermeasures and are difficult to tamper with.

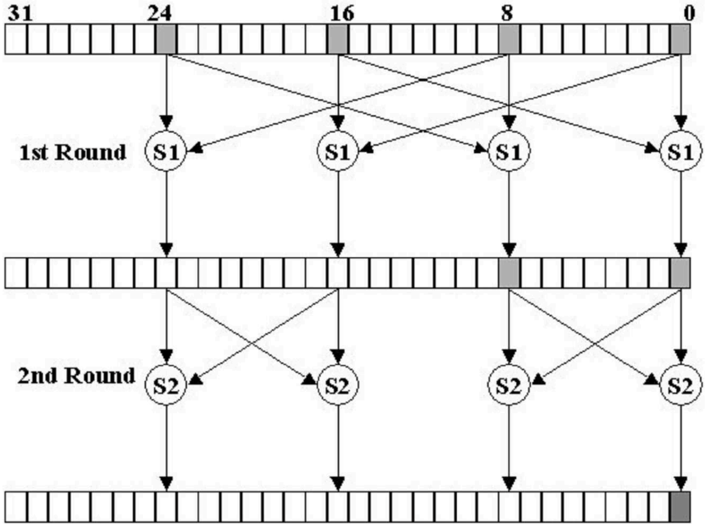

Figure I.9. Compression Rounds in Comp128-1.

However, this does not help if a poorly designed security algorithm is used that will reveal its secrets to a logical or brute force attack. There is one particular algorithm called COMP128-1 that is invariably given as an example of, not only a weak authentication algorithm, but also to illustrate the perils of "security by obscurity". Whilst the 128 bit Ki should have held off brute force attacks, the leakage of the design documentation was swiftly followed by researchers at Berkley [13] extracting a secret Ki in about 130K-160K algorithm executions. The researchers had spotted a flaw that made the algorithm vulnerable to collisions i.e. different RAND challenges giving the same output. The algorithm used multiple rounds of compression based on the Ki and RAND values, but referring to Figure I.9 one can see that the output bytes of the second round are only dependent on 4 input bytes. The final gift to the attackers was that a collision occurring at the second round propagates to the final result. This is described in more detail within the literature [13].

The immediate response was to introduce an authentication retry count into the SIM card so that it would cease to function before the required number of executions had been performed. However, techniques were developed to attack the subsequent rounds of the compression that brought the number of required trials down to 20'000. Side channel leakage was also investigated at IBM [58] bringing the number of executions down to the range of 16 to 1000 to extract the key using a particular form of SPA.

The inescapable conclusion is that there is no reasonable excuse for a GSM network to still issue COMP128-1 SIMs. A common misunderstanding is that it is "the" GSM algorithm whereas it was really only an example made available to members of the GSM association. Standardisation only defined the method of authentication and network operators are free to specify or develop their own algorithms. The standards also permit the use of AuCs that support multiple authentication algorithms

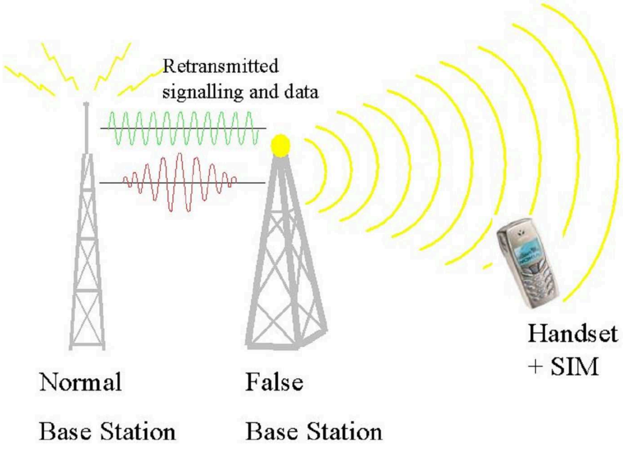

Figure I.10. False Base Station Attack Scenario.

which means that networks can introduce new algorithms whilst gradually phasing out legacy SIMs

Whilst the problems inherent to COMP128-1 were perhaps avoidable there were other security gaps due to the overall GSM security architecture. This is not to say that GSM was badly designed but just that it was a solution to the problems of the previous generation of phones, notably to stop a cloning mechanism, prevent simple radio eavesdropping and make it harder to discover the subscriber's identity. Because of this, GSM is still vulnerable to a type of attack called man-in-the-middle, which becomes possible when a false base station is used as illustrated in Figure I.10. The attack is described is described in GSM and UMTS [33], and is possible because there is no mutual authentication and the user has no way of proving that the network or base station is legitimate.

Another vulnerability relates to the ciphering that is actually carried out in the handset using an algorithm usually called A5/1 (others are available) using the cipher key (Kc) generated by the A8 algorithm in the SIM. There are academic papers [10] describing attacks on the A5/1 algorithm but this seems not to have resulted in a practical criminal exploitation. This is partly due to the difficulty in collecting sequences of ciphertext and corresponding plaintext but more likely because there are easier ways to access the transmitted information.

Aside from the false base station method, there are other opportunities because ciphering is only applied between the user and the base station, everywhere else in the network the information is unprotected. This means that a hacker needs physical access to the network but there are miles of cabling to attack as well as microwave links used between some base stations and core network equipment.

I.4.3 3G - UMTS

In designing the successor to GSM the goal was to further exploit the capabilities of the SIM (referred to as a UMTS SIM or USIM in this context) to remove security weaknesses in the GSM architecture and its algorithms, especially in the light of

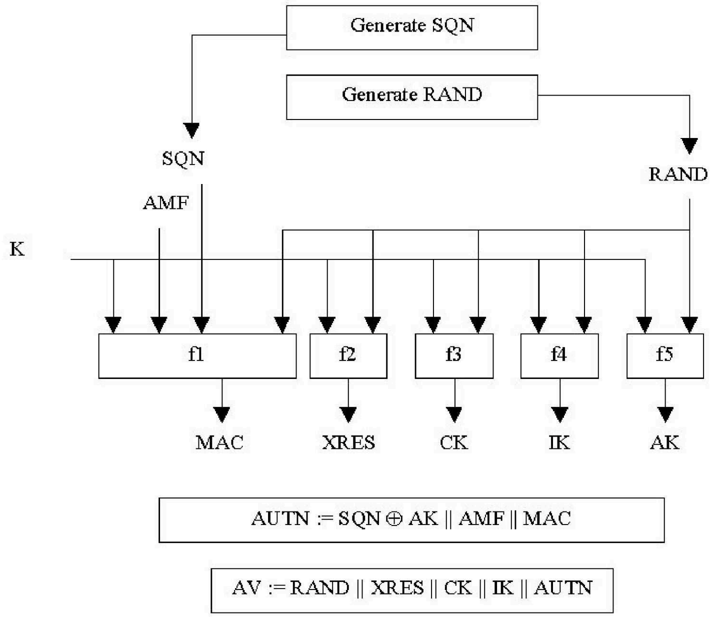
Generate SQN
Generate RAND
SQN
AMF
RAND
K
f1
f2
f3
f4
f5
MAC
XRES
CK
IK
AK
AUTN := SQN ⊕ AK || AMF || MAC
AV := RAND || XRES || CK || IK || AUTN

Figure I.11. UMTS Network Authentication Vector Generation - source [2].

massive advances in computing power. This work was originally driven by ETSI but was eventually handed over to the 3GPP organisation [1]. The most important change was the introduction of mutual authentication between the user and the network based on a strong algorithm. Whereas the description of COMP128-1 was intended to be a secret, the ETSI SAGE team [26] defined a default 3G algorithm (Milenage) based on AES that has been open to expert public scrutiny. The architecture is also improved compared to GSM. The cipher key is longer and there are new integrity (IK) and anonymity (AK) keys plus a sequence number (SQN) to provide replay attack protection. The complete solution at the network end is shown in Figure I.11 with further detail being found in [2].

The basic authentication process from the user/USIM perspective is as follows: The AuC generates an Authentication Vector (AV), this containing a secret value (MAC) that is generated with knowledge of the subscribers key (K) and the sequence number (SQN). This process is shown in Figure I.11. To check the MAC the USIM has to follow the sequence shown in Figure I.12. The USIM generates AK from its own K and the RAND sent from the AuC using function f5. AK is then used to recover the SQN, if this is valid the USIM will then generate the expected value of MAC called XMAC. If XMAC and MAC are identical the network is authenticated. The USIM can then calculate RES, CK and IK that are used to authenticate the USIM to the network.

This solves many of the anticipated problems, but whilst ciphering no longer terminates at the base station and protects some signalling information, it is not end-to-end. The end-point is in fact further back in the network than GSM at a node called the Radio Network Controller (RNC) and this safeguards a lot more of

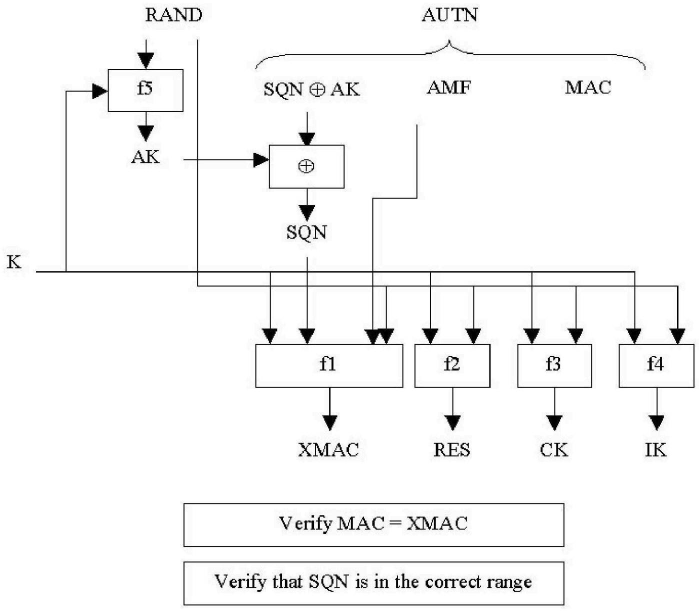

Figure I.12. USIM Authentication Vector Handling - source [2].

the back-haul, the core interconnect remains unencrypted. Another problem is that users may dynamically switch between GSM and UMTS networks based on coverage, performance and service issues. The hand-over between the networks is regarded by some as a security weak point and researchers have sought to use combinations of GSM attacks to try and undermine UMTS [53].

The above text has introduced and compared only the most fundamental security functions of GSM and UMTS that are underpinned by the tamper-resistant functionality of SIM and USIM smart cards, and shown how the security has improved over time in response to fraud. However the (U)SIM provides a range of auxiliary security functions including PIN management, file access control, remote file and application loading and management either locally or Over The Air (OTA). There is also increasing interest in providing APIs (such as JSR177 [71]) in handheld devices so that the security functionality may be accessed and exploited to help protect handset applications and services. In fact with the introduction of Java card and Global Platform functionality the USIM in common with other UICC based smart cards, has become a powerful multi-application Platform. Whilst this brings new opportunities it also beings new challenges and possibilities for attack.

I.5 Summary

Smart card security can be analysed like many other systems as a stack of services each relying on the security of the lower levels. For example, the application developers should be able to rely on the security of the underlying platform. The platform developer relies on the security of the cryptographic algorithms and the hardware being used.

This chapter has provides an overview of the potential problems on smart card security and how these can be avoided. A case study on the European mobile telephone standards is used as a case study to show how security requirements have evolved over time due to the system being exploited.

Smart cards are often considered the magic solution to a problem; often this just removes a weak link in a protocol. The rest of the process also needs to be analysed to ensure that the whole process is secure.

Research in the domain of smart card security is a cyclic process. New attacks are developed against algorithm implementations, standards, APIs, etc. and countermeasures are proposed. The modifications are then reviewed for potential vulnerabilities and further countermeasures proposed if required. The aim of this process is to remain sufficiently ahead of what can be achieved by an individual attacker that smart cards remain secure throughout the period they are active.

Bibliography

[1] 3GPP organisation. `http://www.3gpp.org/`

[2] 3GPP organisation. 3GPP TS 33.102 3G Security; Security Architecture (Release 99) V3.13.0 (2002-12), 2002

[3] M.-L. Akkar and C. Giraud. An implementation of DES and AES secure against some attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptogaphic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pp. 309–318. Springer-Verlag, 2001

[4] C. Aumüller, P. Bier, P. Hofreiter, W. Fischer, and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptogaphic Hardware and Embedded Systems — CHES 2000*, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2002

[5] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu and T. Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults, the Proceedings of the *5th Workshop on Secure Protocols*, volume 1361 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 115–124, 1997

[6] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerers apprentice guide to fault attacks. *Proceedings of the IEEE: Special Issue on Cryptography and Security*, 94(2):370–382, IEEE, 2006

[7] G. Betarte, E. Gimenez, B. Chetali, and C. Loiseaux. FORMAVIE: Formal Modeling and Verification of Java Card 2.1.1 Security Architecture, In Proceedings of E-Smart 2002, pp. 215–229, 2002

[8] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In A. Menezes and S. Vanstone, editors, *Advances in Cryptology — CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pp. 2–21. Springer-Verlag, 1991

[9] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pp. 513–525. Springer-Verlag, 1997

[10] A. Biryukov, A. Shamir, and D. Wagner. Real time cryptanalysis of A5/1 on a PC, In B. Schneier, editor, *Fast Software Ecryption — FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pp. 1–18, Springer-Verlag, 2000

[11] J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In R. N. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pp. 162–181. Springer-Verlag, 2003

[12] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importantce of checking computations. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 1997

[13] M. Briceno, I. Goldberg, and D. Wagner. GSM Cloning. 20 April 1998. `http://www.isaac.cs.berkeley.edu/isaac/gsm.html`

[14] E. Brier, C. Clavier and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pp. 16–29. Springer-Verlag, 2004

[15] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards approaches to counteract power-analysis attacks. In M. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pp. 398–412, Springer-Verlag, 1999

[16] M. Ciet and M. Joye. Practical fault countermeasures for chinese remaindering based RSA. In L. Breveglieri and I. Koren, editors, *Workshop on Fault Diagnosis and Tolerance in Cryptography 2005 — FDTC 2005*, pp. 124–131, 2005

[17] B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost solutions for preventing simple side-channel analysis: side-channel atomicity. *IEEE Transactions on Computers*, 53(6):760–768, IEEE, 2004

[18] C. Clavier, J.-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pp. 252–263. Springer-Verlag, 2000

[19] R. Cohen. The defensive Java virtual machine specification, Technical Report, Computational Logic Inc., 1997

[20] Common Criteria. `www.commoncriteria.org`

[21] Z. Chen. Java Card Technology for Smart Cards : Architecture and Programmer's Guide, Addison-Wesley, 2000

[22] S. Chaumette and D. Sauveron. Some Security Problems Raised by Open Multiapplication Smart cards, 10th Nordic Workshop on Secure IT-systems — NordSec 2005, 2005

[23] S. Chaumette and D. Sauveron, An efficient and simple way to test the security of Java cards, In Proceedings of the 3rd International Workshop on Security in Information Systems — WOSIS 2005, pp. 331–341. INSTICC Press, 2005

[24] European Technical Standards Institute, `http://www.etsi.org/`

[25] European Technical Standards Institute. GSM 11:11 - Digital cellular telecommunications system (phase 2+); Specification of the Subscriber Identity Module - Mobil Equipment (SIM-ME) interface, Version 8.3.0, 1999

[26] European Technical Standards Institute, Security Algorithms Group of Experts (SAGE). `http://portal.etsi.org/sage/sage_tor.asp`

[27] Europay International. MAOS Paltforms Status Technical Report, `www.europay.com`

[28] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: concrete results. In Ç. K. Koç, D. Naccache and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pp. 251–261. Springer-Verlag, 2001

[29] Gemplus. MPCOS Multi Application Payment Chip, Reference Manual Ver 4.0, 1994

[30] C. Giraud and H. Thiebeauld. A survey on fault attacks. In Y. Deswarte and A. A. El Kalam, editors, *Smart Card Research and Advanced Applications VI — 18th IFIP World Computer Congress*, pp. 159–176. Kluwer Academic, 2004

[31] Global Platfom. Global Platform Card Specification, Version 2.1, 2001, `http://www.globalplatform.org`

[32] L. Hemme. A differential fault attack against early rounds of (triple-)DES. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pp. 254–267. Springer-Verlag, 2004

[33] F. Hilebrand. GSM & UMTS, Wiley 2002

[34] Intercede Group plc. OpenPlatform, `http://www.intercede.com/Technology-OpenPlatform.htm`

[35] International Standard Organisation. ISO/IEC 7816, Information technology — Identification cards — Integrated circuit(s) cards with contacts — Part 4: Interindustry commands for interchange, 1995

[36] International Standard Organisation. ISO/IEC 7816, Information technology — Identification cards — Integrated circuit(s) cards with contacts — Part 5: Numbering system and registration procedure for application identifiers, 1994

[37] International Standard Organisation. ISO/IEC 7816, Information technology — Identification cards — Integrated circuit(s) cards with contacts — Part 6: Inter-industry data elements, 1996.

[38] D.H Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits, In *IEEE Transactions On Nuclear Science*, volume 39, pp. 1647–1653, IEEE, 1992

[39] Hive-Minded. Smartcard.NET, `www.hiveminded.com`

[40] M. Joye and F. Olivier. Side-channel attacks. In H. van Tilborg, editor, *Encyclopedia of Cryptography and Security*, pp. 571–576. Kluwer Academic Publishers, 2005

[41] M. Joye, J.-J. Quisquater, F. Bao, and R.H. Deng. RSA-type signatures in the presence of transient faults, In M. Darnell, editor, *Cryptography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pp. 155–160, Springer-Verlag, 1997

[42] P. Kocher. Timing attacks on implementations of diffe-hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pp. 104–113. Springer-Verlag, 1996

[43] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pp. 388–397. Springer-Verlag, 1999

[44] ITSEC. `http://www.ssi.gouv.fr/site_documents/ITSEC/ITSEC-fr.pdf`

[45] T. M. Jurgensen and S. B. Guthery. Smart cards : the developer's toolkit, Prentice Hall, 2002

[46] X. Leroy. Bytecode verification for java smart card. Software Practice & Experience, volume 32, pp. 319–340, 2002

[47] MAOSCO Ltd. The MULTOS^{TM}Specification, `http://www.multos.com/`

[48] C. Markantonakis. The case for a secure multi-application smart card operating system. In E. Okamoto, G. I. Davida, and M. Mambo, editors, *Information Security Workshop 97 — ISW '97)*, volume 1396 of *Lecture Notes in Computer Science*, pp. 188–197. Springer-Verlag, 1997

[49] G. McGraw and E. W. Felten. Securing java, J. Wiley & Sons, 1999

[50] G. McGraw, K. Ayer, and E. W. Felten. Jave Security meets smart cards, security enhancements in java card 2.1.1 will help multi-application smart cards take off in U.S. markets, Information Security Magazin, `http://www.infosecurity.com/articles/march01/cover.shtml`, 2001

[51] T. S. Messerges. *Power Analysis Attacks and Countermeasures for Cryptographic Algorithms*. PhD thesis, University of Illinois, Chicago, 2000

[52] T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In Ç. K. Koç and C. Paar, editors, *Cryptogaphic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pp. 71–77. Springer-Verlag, 2000

[53] U. Meyer and S. Wetzel, On the Impact of GSM Encryption & Man-in-the-Middle Attacks on the Security of Interoperating GSM/YMTS Networks. In Proceedings of IEEE International Symposium on Personal, Indoor and Mobile Radio Communications — PIMRC 2004, volume 4, pp. 2876–2883, IEEE, 2004.

[54] M. Montgomery, K. Krishna. Secure object sharing in Java card, In proceedings of the USENIX Workshop on Smart Card Tehnology — Smartcard '99, USENIX, 1999

[55] D. Naccache, P. Q. Nguyễn, M. Tunstall, and C. Whelan. Experimenting with faults, lattices and the DSA. In S. Vaudenay, editor, *Public Key Cryptography — PKC 2005*, volume 3386 of *Lecture Notes in Computer Science*, pp. 16–28. Springer-Verlag, 2005

[56] General Information Systems Ltd. OSCAR, Specification of a smart card filling system incorporating data security and message authentication, `http://www.gis.co.uk/oscman1.htm`

[57] Parliamentary Office of Science and Technology. Mobile Telephone Crime. In POST Briefing Note 64, 1995

[58] J. R. Rao, P. Rohatgi, H. Scherzer, and S. Tinguely. Partitioning attacks: or how to rapidly clone some GSM cards. In Proceedings of IEEE Symposium on Security and Privacy, pp. 31–41, IEEE, 2002

[59] E. Rose and K. H. Rose. Lightweight bytecode verification. In Formal Underpinnings of Java — OOPSLA '98, ACM, 1998

[60] D. Samyde, S. P. Skorobogatov, R. J. Anderson, and J.-J. Quisquater. On a new way to read data from memory. In *Proceedings of the First International IEEE Security in Storage Workshop*, pp. 65–69, IEEE, 2002

[61] D. Sauveron. *Étude et réalisation d'un environnemet d'expérimentation et de modélisation pour la technologie java card^{TM}. application à la sécurité*. PhD thesis, University of Bordeaux, Bordeaux, 2004

[62] Season 2 Interface. `http://www.maxking.co.uk/`

[63] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In B. S. Kaliski Jr. and Ç. K. Koç and C. Paar, editors, *Cryptogaphic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pp. 2–12. Springer-Verlag, 2002

[64] SmartCard Trends. .NET brings web services to smart cards, April/May Issue, 2004

[65] Sun Microsystems. Java Card API Ver 1.0, `www.javasoft.com/javacard/`

[66] Sun Microsystems. Java Card API Ver 2.0, `www.javasoft.com/javacard/`

[67] Sun Microsystems. Java Card 2.2.1 Application Programming Interface, 2003

[68] Sun Microsystems. Java Card 2.2.1 Runtime Environment (JCRE) Specificqtion, 2003

[69] Sun Microsystems. Java Card 2.2.1 Virtual Machine Specification, 2003

[70] Sun Microsystems. Java Card API 2.2.1 Reference Implementation, 2002, `http://www.javasoft.com/products/javacard/`

[71] Sun Microsystems. JSR 177 Expert Group. Security and Trust Services API (SATSA) for J2ME V1.0, 2004

[72] D. A. Watt and D. F. Brown. Programming Language Processors in java: compilers and interpreters, Prentice Hall, 2000

[73] M. Witteman, Java Card Security, Information Security Bulletin 8, pp. 291–298, 2003

[74] ZeitControl. BasicCard. `http://www.basiccard.com/`

[75] J. Ziegler. Effect of Cosmic Rays on Computer Memories, *Science*, volume 206, pp. 776–788, 1979