Low Voltage Fault Attacks on the RSA Cryptosystem

Alessandro Barenghi
DEI
Politecnico di Milano
Milan, Italy
barenghi@elet.polimi.it

Guido Bertoni
STMicroelectronics
Agrate Brianza, Italy
guido.bertoni@st.com

Emanuele Parrinello
Politecnico di Milano
Milan, Italy
emanuele.parrinello@mail.polimi.it

Gerardo Pelosi
DIIM
Università degli Studi di Bergamo
Dalmine (BG) - Italy
gerardo.pelosi@unibg.it

***Abstract*—Fault injection attacks are a powerful tool to exploit implementative weaknesses of robust cryptographic algorithms. The faults induced during the computation of the cryptographic primitives allow to extract pieces of information about the secret parameters stored into the device using the erroneous results.**

Various fault induction techniques have been researched, both to make practical several theoretical fault models proposed in open literature and to outline new kinds of vulnerabilities.

In this paper we describe a non-invasive fault model based on the effects of underfeeding the power supply of an ARM general purpose CPU. We describe the methodology followed to characterize the fault model on an ARM9 microprocessor and propose and mount attacks on implementations of the RSA primitives.

***Index Terms*—Low voltage Fault Attacks, Hardware Security, RSA attacks.**

I. Introduction

The security of devices running cryptographic primitives not only depends on the underlying mathematical assumptions, but also strongly relies on a sound implementation on the target hardware or software platform.

Attacks targeting implementations instead of algorithms themselves are usually categorized into two main groups: side-channel attacks and fault attacks.

Side-channel attacks try to infer part of the secret information employed by cryptographic algorithms through statistical methods applied on physical parameters measured externally on the device (e.g., power consumption, timing information, electromagnetic radiations).

Fault attacks aim at inducing errors during the computation of a cryptographic primitive by altering either the control flow or the internal-state data of the cryptographic algorithm implemented on the target device. Depending on the fault model, the erroneous cryptograms may leak sufficient information to extract secret key material from the target implementation, even if the cryptographic algorithm is mathematically secure and endowed with countermeasures against other side channel attacks.

In this paper we describe a non-invasive fault model based on the effects of underfeeding the power supply of a general purpose CPU. Our methodology involves keeping the level of voltage supplied to the processor constantly lower than the nominal one. We characterize the faults happening during computations in these imposed working conditions both under a quantitative and a qualitative perspective. We propose some attack techniques which exploit the newly characterized fault model and provide experimental evidence of their practical feasibility.

The remainder of the paper is organized as follows. Section II describes related work. Section III describes the method employed to delineate a characterization of the faults which may be injected through underfeeding a general purpose CPU. Section IV presents three possible attack models deemed feasible with the newly introduced fault model and Section V reports the experimental evaluation for two of them. Finally, Section VI summarizes our conclusions and points towards future research directions.

II. Related Work

Following the classification proposed in [1], fault induction techniques may be split into two main categories: those inducing *transient* faults and those inducing *permanent* faults.

While the latter are usually much more powerful, as far as the information leakage goes, the former are considered the sensible scenario against which devise defenses due to their repeatability and lack of tampering evidences left on the device.

Transient faults are known to be injectable through several methods, namely: irradiation of the device, EM-inducted disturbances, clock phase shiftings, and alterations in the power supply.

The first technique relies on altering the state of the circuit by irradiating directly the silicon die through the use of a concentrated light beam, either polarized (laser beams) or unpolarized (common flashes) [2]. The beam is usually timed in order to achieve changes in the values stored in SRAM cells or in registers and either allowing modification or inferences on the values previously contained. The alterations may be as precise as a single bit assuming it is possible to focus the beam on a spot as wide as a single gate. This constraint is becoming increasingly difficult to comply with, since the new etching technologies are able to print sub-visible wavelength wide gates.

EM-induced faults are a recent technique by Schmidt and Hutter [3] and can be achieved through small electrical discharges generated near the sensitive device with the help of a pair of small electrodes. The technique can be timed, although not with clock cycle accuracy, and has the advantage of avoiding the decapsulation of the chip. On the other hand, there is no way to aim the fault at small sensitive zones of the chip. Moreover, packages providing inbound EM-shielding (e.g. grounded metal heat spreader ones) are able to thwart the attack.

As far as the non-package lesive techniques go, it might be possible to insert phase shifts on the clock line through manipulating the position of the rising and falling edges. This tampering may induce instruction skipping in small microcontrollers, therefore altering the control flow of the algorithm, possibly leaking sensitive information. Until now, no practical implementations of these attacks have been proposed in the open literature.

Another transient fault induction technique relies on the capability of altering the yield of the power supply line. A first method consists of inserting tiny, well timed glitches, realized with either spikes or temporary brown-outs, aimed at disrupting the value held on the input lines of flip-flops during their setup time. This causes incorrect values to be stored in latches thus possibly resulting in either instruction skips or data corruptions. A practical example of an attack brought to a plain square-and-multiply RSA software implementation on a PIC Microcontroller through this technique is given in [4].

Another method of injecting fault relies on constantly underfeeding a device to alter the values stored by its bistables due to the slowdown in the logical gate setup time. In [5] the authors report a faulty behavior of the lines at the end of the longest combinatorial cones of a smart card implementation of AES, and exploit it in order to carry a successful attack using the method proposed by Piret *et al.* in [6]. This method has the advantage of not being influenced by either the etching technology of the chip or the packaging, and relies only on the very reasonable assumption of being able to alter the feeding line of the chip. Until now, only a single experimental result of a succesful attack employing this technique by Amiel *et al.* is reported in [7].

III. Power Hunger Faults on General Purpose Processors

In this paper we are presenting a new fault model which relies on uniformly underfeeding a general purpose CPU.

Lowering the voltage at which a device is fed raises the setup time for the latches of the circuit and slows down the propagation of signals on the bus lines. What we are expecting is to see a growing number of faults appearing when the feeding voltage is pulled under the nominal operating range. Due to tiny differences in the circuitry we foresee that some of the bistables will fail to hold the correct values more often than others, thus showing strong spatial locality in the faults. We are not timing the underfeeding in order to obtain a more widely applicable technique which will not be relying on sub-clock accurate injectors. This enables us to apply this method despite the rising clock frequencies and the shrinking etching technologies which represent a serious impairment to other techniques.

We will now describe the workflow which led us to a full characterization of the fault model using as target platform an ARM-9 microprocessor[1] due to its widespread diffusion in both embedded architectures and low power general purpose computers. Since the SoC has three separate supply lines, one for the core, one for the I/O buses and one for the memory interface, we chose to interact with the one feeding the computational part, due to its tight coupling with the execution flow of the binaries run on the device.

The experimental platform used to investigate the effects of the fault is an ARM based development board, specifically a SPEAr Head200 [9] built by ST Microelectronics, fed through an Agilent 3631A power supply with a 1mV precision. The voltage measures were taken with an Agilent 34420A voltmeter with a nV precision probe. The probe programs are written in ANSI C, compiled into ARM binaries through a development toolchain based on the GCC compiler. The object code is subsequently loaded on the platform through the U-Boot embedded bootloader [10].

A. Voltage Range Exploration

The first step in evaluating the fault model is to determine the undervolting level at which faults begin to appear and the lower bound of the working voltage range. The bottom threshold for the voltages used during the experiments is easily determined since the UART interface stops outputting anything under a specific voltage level, thus impeding the retrieval of any information from the system. In order to detect the upper bound of the fault induction range we tested the correct functioning of the CPU using a simple probe program whose core loop is reported hereafter.

```
for(a=i=0; i<1000000; i++){
  a = a + 1;
  if(a != i+1){
     printc('?');
     if(a!= i+1){
        printc('#');
        a = i+1; // fix the fault
        if (a != i+1) a = i+1;
     }
  }
}
```

The aforementioned code increments a variable a million times, and checks if a fault has happened exactly after the increment. A redundant check has been added in order to lower the likelihood of a false positive occurring in the detection. We consider an actual fault to have happened only if both checks confirm it.

Figure 1 shows the increase of the number of faulty outcomes of the computation when sweeping the voltage range at which the CPU starts to malfunction. The represented values were computed averaging 500 thousands runs of the code for each voltage level probed. The picture delineates a linear

[1]Specifically, an ARM926EJ-S [8].

growth of the percentage of faulty computations triggered by the lapse in the voltage supply. The voltage point at which the probability of a faulty outcome matches the one of a correct computation is highlighted by a dashed line. We expect that, below this threshold voltage, multiple faults will be taking place within a single run, thus increasing the likelihood of a miscomputation. In order to characterize the causes of the errors in the computations, we split the results into three groups according to the number of faults which triggered them.

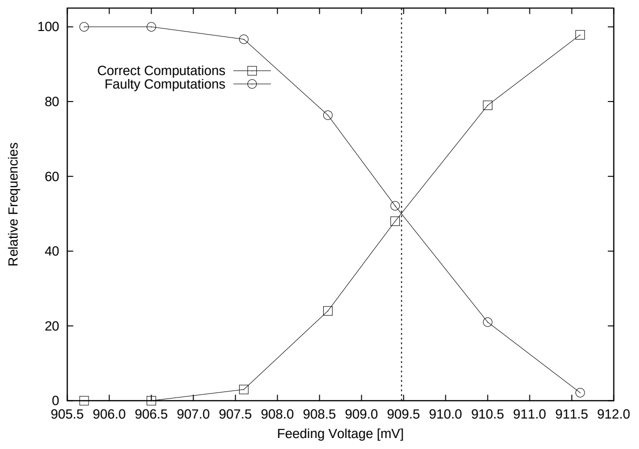

Figure 1. Percentage of correct and wrong computations averaged over 500 thousands runs.

Figure 2 shows a 1.5 mV wide voltage range where a single fault happens. Above the aforementioned threshold voltage the probability of having a faulty computation triggered by a single fault ranges from 2% to 40%. This probability dwarfs the one of having multiple faults contributing to the erroneous result. When working under the threshold, the number of possible faults starts growing, and multiple fault scenarios start to dominate the fault profile. All the previous figures are obtained by keeping constant the number of instructions actually executed by the CPU during the measurements. It is sensible to assume that a growth in the executable code size will be met by an analogous rise in the probability of a fault appearing during the computation. This observation suggests that the most useful voltage point for controlled, and thus exploitable, fault injection is the farthest reachable from the threshold. Using this criteria in order to determine the working voltage point will lead to a longer fault collection campaign due to a reduced incidence of faults.

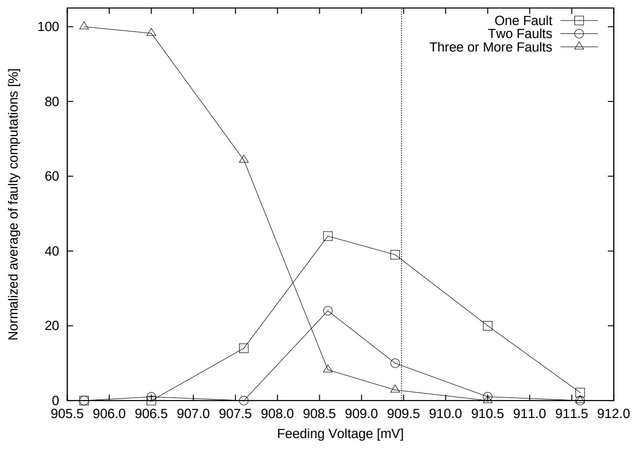

Figure 2. Distribution of the quantity of the injected faults as a function of the voltage.

B. Fault Characterization

Having ascertained the possibility of injecting a reasonably low number of faults per computation, we moved on to a finer grained characterization. The assembly level instruction in a code flow may be split into three categories according to the architectural units composing the CPU which are used to complete them. The three categories are arithmetical-logical operations, memory operations and branch instructions.

Memory instructions represent the most expensive operation class in terms of power consumption, therefore they are allegedly the most vulnerable to underfeeding issues.

In order to ascertain this, we recompiled the same probe program instructing the compiler to keep the variables in the CPU registers during the whole computation.

The execution of the tuned program showed no faults, thus indicating that the wrong values detected by the checks were uniquely to be ascribed to memory operations, while both arithmetical-logical and branch instructions ran correctly regardless of the voltage.

The low voltage fault immunity exposed by the CPU registers is to be ascribed to the low capacitance design of their implementation. This requirement is dictated by the architectural need of fast accesses to the component, which is mandatory in order to design efficient units.

The next step in the characterization of this new kind of fault was checking if both `load` and `store` instructions were equally affected. In order to verify which memory instructions are affected by faults, it is possible to use the value held in the registers as a fault free value for checks. We set up a probe program which moved to and from the memory all-zero and all-ones words, and ran it multiple times, sweeping the whole voltage range found before.

The programs running `load` instructions turned out to be the only ones reporting misexecutions.

The ARM architecture sports three different supply lines, one for the memory interface, one for the I/O buses and one for the core. Since we are mangling only the voltage level of the last line, the aforementioned behavior may be sensibly ascribed to the fact only the memory operations which are storing values on the underfed part (i.e. the `load` operations which store information in the registers) suffer from the lack of power. On the other hand the store operations are placing the data on a properly fed part of the architecture.

C. Fault Localization

The experiments run up to now characterize the faults as affecting only `load` instructions and, as far as their number in a single execution goes, depending on the supplied voltage.

We are now willing to investigate whether the faulty behavior of the `load` instructions is depending either on the referenced memory address or on the loaded value.

In order to understand this key point, a probe program was designed to overwrite a one million 32-bit word array with 1s, and subsequently to check the values which were loaded back into the registers, while keeping the voltage in the single fault functioning range. During this test the data cache included in the ARM9 processor was disabled.

Figure 3 shows the number of faults occurred while performing 10^6 `load` operations of a 32-bit integer from the aforementioned array. In order to analyze the data, the probed memory has been partitioned in 40 kB wide zones. We encountered 1864 faulty `loads` while running the program, thus we are expecting an average of 18.64 faults per zone in case of a uniform distribution. The dashed line in Fig. 3 indicates the expected number of faults occurring for each zones, assuming a uniform distribution of the faults over the memory. To confirm the hypothesis of a uniform distribution

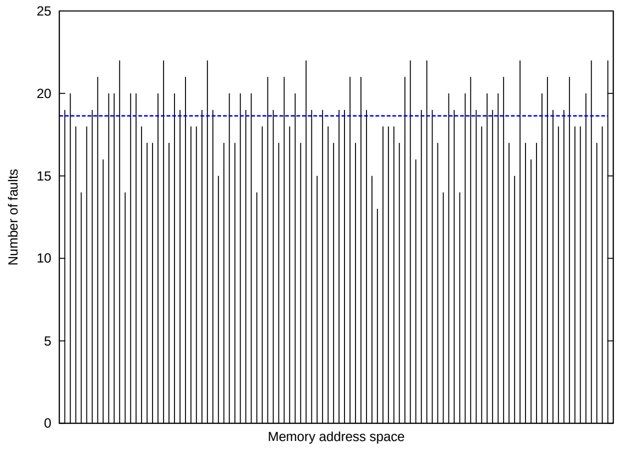

Figure 3. Distribution of the quantity of the injected faults as a function of the position in the address space. The dashed line indicates the expected average value in the hypothesis they are uniformly spread.

of the faults over the whole address space, we modeled the position hit by the fault as a random variable and we conducted a Pearson χ^2 test to assess the goodness of fit. The results confirmed our hypothesis with a confidence level near to 100%.

After investigating the dependence of the faults from the memory address of the loaded value, we decided to evaluate their impact on the fetched values.

Examining the faulty values we noticed that the corruption pattern is fixed within each retrieved word, independently of the address at which the word was stored.

All the faults occur as single bit flip-downs and their position changes only when testing different chip samples.

D. Instruction Faults

After having a well defined model for the faults occurring during the loads from memory, we tried to achieve control flow alterations through corruptions in the instruction fetch phase. We therefore disabled the instruction cache present in the ARM microprocessor in order to fetch the instructions directly from the main memory, and devised a test program in order to spot possible faults which may modify the opcode of the instructions through altering its binary encoding.

In particular, the affected instructions will be transformed into the ones having a binary encoding differing only by a flip-down of the faulty bit. For instance, through a single bit flip-down a possible instruction swap is the following one.

```
AND R1,R1,#0x42 // Fault Free
EOR R1,R1,#0x42 // Faulty
```

Since the "and" and the "exclusive-or" instructions have a radically different behavior, it is possible to alter the inner working of the algorithm through swapping them.

An important feature of the ARM architecture is also the opportunity of executing all the arithmetical-logical instructions conditioned by a binary predicate. It is therefore possible to flip the check of every predicate and obtain instruction skipping as depicted in the following code sample.

```
ADDNE R1,R1,#0x42 // Fault Free
ADDEQ R1,R1,#0x42 // Faulty
```

Moreover, since also the branch driving mechanism relies on the same condition bits of the predicate checking one, the control flow of the program may be equally altered if the condition bit of a branch is flipped.

```
BNE LOOP // Fault Free
BEQ LOOP // Faulty
```

We have been able to reproduce all the aforementioned alterations on our chip samples. Since the alterations are chip dependent, the exploitation of this kind of fault requires to know precisely which bit is affected by the fault thus determining which instruction swaps are performed. It is possible to define all the possible mutations, depending on which bit is altered and therefore devise specific attacks exploiting each one of them.

E. Underfeeding Inducible Faults

In order to exploit the type of faults we are able to induce, we now propose description of the fault model which aims at being independent of a specific chip instance used. Albeit originating from the same cause, i.e. faulty `load` operations, we may distinguish two different effects of the faults: *data corruption* and *instruction swap*. For the sake of clarity, we will deal separately with the two outcomes in order to point out better how they can be exploited.

Data related faults are representable as a transient change in the value of a t-bit wide variable c during an execution. In particular they are single bit flip-downs placed in a fixed position embedded into the microprocessor word. The faulty value $\widetilde{c}$ equals the correct one c minus a power of two 2^ϵ where ϵ is the position of the fault. Possible values of ϵ are expressed in the form $\epsilon = k\,w + i$ with w equal to the word length, $i \in [0, w-1]$ and $k \in [0, \frac{t}{w}]$.

Instruction related faults define a one way swap between two classes of instructions whose binary encoding of the opcode differs by only a single bit. The swap occurs only a limited amount of times, determined by the level of underfeeding of the device, and may be reduced up to a single one in the whole computation of a target algorithm.

IV. Attacks

To exhibit a some practical applications of our fault model, we now propose three attacks, exploiting fault injection through power draining, to a software implementation of the RSA cryptosystem on an ARM-9 processor.

Throughout the description of the attacks, we will use the following notation: let p and q be two large primes and let $n = pq$ be the RSA modulus. Let e, d be two unitary elements in $(\mathbb{Z}^*_{\varphi(n)}, \cdot)$ representing the public and private exponent bound together by the congruence $d = e^{-1} \bmod \varphi(n)$. Let $t = \lceil \log_2 \varphi(n) \rceil$ denote the length of their binary encodings. Having $m, c \in \mathbb{Z}^*_n$, we denote a generic RSA plaintext-ciphertext pair as $c = m^e \bmod n$. Having $m, s \in \mathbb{Z}^*_n$, we denote a generic RSA message-signature pair as $s = m^d \bmod n$.

A. Bellcore Attack

The Bellcore attack [11] enables to factor the modulus n through inducing an error during the computation of the exponentiation phase of any RSA primitive implemented using the Chinese Remainder Theorem.

Let $s = CRT(m_p, m_q)$ denote the CRT recombination of the value $s = m^d \bmod n$ from the two values $s_p = m^d \bmod p$ and $s_q = m^d \bmod q$:

$$s = \big(s_p + p\,((s_q - s_p)(p^{-1} \bmod q) \bmod q)\big) \bmod n$$

If a fault occurs during the computation of s_q while the computation of s_p remains error free, we may denote the faulty value of s_q as $\widetilde{s_q} = s_q + \Delta$. Therefore, the faulty CRT recombination will yield $\widetilde{s} = CRT(s_p, \widetilde{s_q})$, given by:

$$\widetilde{s} = s + p\,\big(\Delta(p^{-1} \bmod q) \bmod q\big) \bmod n$$

Since the value $\widetilde{s} - s$ shares a nontrivial factor with the modulus n, it is possible to extract $p = \gcd(\widetilde{s} - s, n)$ efficiently through Euclid's Algorithm.

Moreover, as showed in [12], the modulus factorization is also computable using only the message m and one faulty computation of the signature $\widetilde{s}$, through calculating $p = \gcd(\widetilde{s}^e - m, n)$.

The main advantage of this technique is that any kind of fault induced in the computation of one of the two values to be recombined with the CRT will yield a useful faulty computation regardless of precise timing and placement.

B. e-th Root Extraction Attack

The target of this attack is to retrieve the input message encrypted through RSA using a correct and a faulty encryption of the same message. A practical applicative scenario could be the retrieval of the session key during an RSA-KEM [13] handshake. This assumes that the party which is in charge to choose the session key re-encrypts the same value in case a faulty encapsulation occourred. To the best of the author's knowledge this technique has not yet been used in order to mount a fault based attack.

Consistently with the fault model described in Section III-E, the injected fault is supposed to corrupt a single bit of the public exponent e in a fixed position $\epsilon \in [0, t-1]$. When this happens, we are able to obtain a faulty ciphertext, henceforth denoted as $\widetilde{c}$, which can be represented as $\widetilde{c} = m^{e-2^\epsilon} \bmod n$.

We observe that the number of possible positions of the fault are exactly t/w where w indicates the word length of the CPU. Thus, to spot the location of the faulty bit, we need to iterate the plaintext retrieval algorithm (Algorithm IV.1) t/w times and check through re-exponentiation if the retrieved message is correct. This is still computationally feasible since t/w grows logarithmically w.r.t. the exponent. Through this check we can also discard all the faulty ciphertexts generated by errors which did not alter the exponent, since the re-exponentiated value will fail to match the ciphertext for all the possible positions of the fault.

From now on, we will be in need to calculate inverses over $(\mathbb{Z}^*_n, \cdot)$. We want to point out that, even though $(\mathbb{Z}^*_n, \cdot)$ encompasses zero dividers, this will not be a problem since, in the event that the chosen number $\widetilde{c}$ is not a unitary element of $(\mathbb{Z}^*_n, \cdot)$, this implies that being a zero divider, is therefore a multiple of either p or q. This immediately implies that if we compute $\gcd(\widetilde{c}, n)$ we will obtain a nontrivial divider of the modulus, thus breaking the cryptoscheme.

Since we can compute inverses over $(\mathbb{Z}^*_n, \cdot)$, we are able to operate subtractions between the exponents of the faulty and the correct ciphertext through multiplying $c\,\widetilde{c}^{-1} \bmod n$. This will enable us to reduce the exponent of c to one, thus effectively recovering the plaintext: Algorithm IV.1 describes how the retrieval is performed.

Through exploiting the single-bit nature of the fault, at first (line 2) the algorithm computes the value of the message raised to 2^ϵ where ϵ is the alleged position of the flip-down (taken among the few possible ones). Then the first loop (lines 4–7) eliminates the ones in the exponent higher than the fault position through dividing it by the aforementioned value, which is repeatedly squared in order to cover all the possible positions.

Once all the bits of the public exponent with position greater than ϵ have been cleared, the algorithm triggers a chain of subtractions between m^{2^ϵ} and the ciphertext with reduced exponent (lines 8–18) and swap their roles when the exponent of the first number is smaller than the one of the second (lines 16–18). Starting from line 8, the algorithm employs two temporary values, x and y in order to store the two numbers involved in the reduction. Since the chain of subtractions required to reduce the public exponent to one would be $O(e)$ long, we would like to subtract a convenient multiple of the minuend each time. In particular, the function AlignExp (line 12) computes the maximum number of times the minuend can be doubled while being smaller than the subtrahend, and returns it together with the multiplied exponent. In order to maintain coherence between the stored value of the y exponent e_{y_s} and the actual exponent of y, we need to square a correct number of times (line 13). Using this technique we reduce the number of subtractions to $O(\log_2 e)$ thus making computationally feasible the retrieval of the message. This chain of subtractions will eventually lead to the recovery of the message since the possible values generated for the exponent are strictly decreasing and belong to the subgroup $(\langle \gcd(e,\ 2^\epsilon)\ \rangle,\ +)$ over $(\mathbb{Z}_{\varphi(n)}, +)$ therefore, being e and 2^ϵ coprime, we will surely generate 1.

It is easily possible to generalize the algorithm assuming the position of the fault is completely unknown but fixed. In this case the Algorithm IV.1 has to be run at most $t = \lceil \log_2 \varphi(n) \rceil$ times in order to check all possible fault locations.

Algorithm IV.1: PLAINTEXT RETRIEVAL

```
   Input: $c,\ \widetilde{c},\ k_{pub} = (n, e),\ \epsilon$
   Output: $m$: potential plaintext
   Data: $t = \lceil \log_2 \varphi(n) \rceil,\ c = m^e \bmod n,$
         $\widetilde{c} = m^{e-2^\epsilon} \bmod n,\ \epsilon \in [\ 0,\ t-1\ ]$: fixed fault
         position.
1  begin
2      $m_\epsilon \leftarrow c\ \widetilde{c}\ ^{-1} \bmod n$ /* $m_\epsilon \leftarrow m^{2^\epsilon} \bmod n$ */
3      $a,\ b \leftarrow \widetilde{c},\ m_\epsilon$
4      for $j \leftarrow \epsilon + 1$ to $t - 1$ do
5          $b \leftarrow b^2 \bmod n$
6          if $e_j = 1$ then
7              $a \leftarrow a\ b\ ^{-1} \bmod n$
       /* $a = m^{e \bmod 2^\epsilon} \bmod n,\ e \bmod 2^\epsilon < \varphi(n)$ */
8      $x,\ y \leftarrow m_\epsilon,\ a$
9      $e_x,\ e_y \leftarrow 2^\epsilon,\ e \bmod 2^\epsilon$
10     $e_z \leftarrow e_x / e_y$ /* integer division */
11     while $e_z \neq 1$ do
12         $e_{y_s}, ns \leftarrow$ ALIGNEXP$(e_x, e_y)$
13         $y_s \leftarrow y^{2^{ns}} \bmod n$
14         $z \leftarrow x\ y_s^{-1} \bmod n$
15         $e_z \leftarrow e_x - e_{y_s}$
16         if $e_z < e_y$ then
17             $x, y \leftarrow y, x$
18             $e_x, e_y \leftarrow e_y, e_x$
19     return $z$
20 end
```

C. Secret Key Extraction Attack

The target of this attack is to retrieve the private exponent d when the device is signing messages with an RSA implementation performing the exponentiation through plain square-and-multiply. The attack scenario is the one in which the attacker has access to a decrypting device and is allowed to choose arbitrary ciphertexts to be fed while injecting faults.

The key points of this attack are sketched in [14], where it is assumed a generic fault hypothesis which is compliant with our fault model.

The fault model used in this context is the one concerning the swap of instructions with similar binary encodings; in particular, our purpose is to exploit the substitution of the `not equal` condition in the instruction opcode with an `equal` condition. In the following code snippet is reported an example of the check section of the left-to-right square-and-multiply exponentiation used in order to perform an RSA signature:

```
// in the S&M loop
...
MOV r2, #0 // load constant value 0
CMP r1, r2 // compare with exponent bit
BNE MULT // branch to multiply section
...
```

Every time the fault flips the condition checked by the branch instruction in the code, the effect on the result is the same of a bit flip in the value of d. Therefore there are two possible values for the corrupted signature $\widetilde{s}$: $\widetilde{s} = s^{d-2^\epsilon} \bmod n$ or $\widetilde{s} = s^{d+2^\epsilon} \bmod n$ depending on whether the value that the check misjudged is a zero or a one.

In order to discover both the position and the effect of the fault it is possible to use the method described in Algorithm IV.2.

The key idea is to use the faulty signature to extract the information on the secret exponent one bit at a time. First of all, it is necessary to precompute a lookup table, A, of all the values $A_i = m^{2^i} \bmod n$ for $i \in [0, t-1]$ and store it. Since the value $s/\widetilde{s} \bmod n$ or $\widetilde{s}/s \bmod n$ may be equal to $m^{2^\epsilon} \bmod n$ depending on whether a flip down or a flip up occurred, it is possible to search both values in the precomputed table. If one of the two values matches an entry, we both know the position of the fault and the value of the bit of the exponent d. This procedure can be iterated until a reasonable number of bits of the exponent is known.

In order to give a quantification of the number of faults

Algorithm IV.2: SECRET KEY RETRIEVAL

```
   Input: $m,\ s,\ k_{pub} = (n, e)$
   Output: $d = \langle d_{t-1}, d_{t-2}, \ldots, d_1, d_0 \rangle$: recovered secret
           key
   Data: $t = \lceil \log_2 \varphi(n) \rceil, R = t \ln(t+1)$
         $A = \langle \forall i \in [0,\ t-1], A_i = m^{2^i} \bmod n \rangle$
1  begin
2      for $i \leftarrow 0$ to $t - 1$ do
3          $d_i \leftarrow$ NIL
4      for $i \leftarrow 0$ to $R - 1$ do
5          $\widetilde{s} \leftarrow$ FAULTYSIGN$(m)$
6          for $i \leftarrow 0$ to $t - 1$ do
7              if $s\ \widetilde{s}^{-1} \bmod n = A_i$ then
8                  $d_i \leftarrow 1$ /* flip down */
9                  break
10             if $\widetilde{s}\ s^{-1} \bmod n = A_i$ then
11                 $d_i \leftarrow 0$ /* flip up */
12                 break
13     return $d$
14 end
```

needed to discover the value of t unknown bits we consider a random variable X counting how many faults are injected in a run until all the bits have been hit and aim at finding its expected value. Let X_j denote the random variable counting the number of injected faults needed to cover the $j+1$-th bit assuming that all the others up to the j-th are covered. The random variable X_j follows a geometric distribution with parameter $\frac{t-j}{t}$. Therefore, the probability that, after k injections, an untouched bit gets hit at a $j+1$-th position is given by $\text{Prob}(X_j = k) = \frac{t-j}{t}\left(\frac{j}{t}\right)^{k-1}$. The expected value of X_j is $\mathbb{E}[X_j] = \frac{t}{t-j}$. Since the original X is actually $X = \sum_{j=0}^{t-1} X_j$, it is possible to model the expected value of X as $\mathbb{E}[X] = t\sum_{j=1}^{t}\frac{1}{j} \leq t\ln(t+1)$.

This gives us an average number of faults $R = t\ln(t+1)$ to be injected successfully in order to retrieve all bits. The attack may be run up to the complete discovery of all the bits of d or until when the possible values of the exponent are few enough to be checked through brute force.

V. Experimental Results

After proposing a fault model backed by experimental confirmations and delineating a number of attacks exploiting it, this section presents the results of the experimental campaign conducted in order to assess the practical feasibility of the Bellcore and e-th root extraction attack. We delegate to a future development the experimental realization of the secret key retrieval attack reported in Section IV-C due to the long times involved in reproducing significant instances of the fault.

A. Bellcore Attack Evaluation

The first campaign was conducted to explore the feasibility of the attack to the CRT based version of RSA. The employed C-code implements RSA using Montgomery Multiplication [15] and performs the two exponentiations needed by the CRT through plain square-and-multiply following the algorithm delineated in Section IV-A. None of the countermeasures known in the literature were enabled [16], [17]. In a first phase, the whole algorithm is run in a continuous loop in order to determine the voltage point at which the faults begin to appear. This allows us to tune the induced number of faults to a single one per algorithm run.

Once the correct voltage point has been determined, we ran three experimental campaigns of 10000 runs each to compute RSA signatures with modulus sizes: 512, 1024, 2048 bits respectively. The binary code was directly loaded on the platform through the U-Boot embedded bootloader, thus running without any underlying operating system. The data cache of the ARM9 microprocessor was disabled for the whole duration of these experiments. Subsequently, all the faulty computation results were collected and the gcd between them and the modulus was computed to retrieve one of the two factors. Table I shows the percentage of exploitable faults obtained during the campaign. As expected, the number of injected faults grows with the size of the modulus since the number of `load` operations employed increases. For large modulus sizes, the exploitable faults represent the vast majority of the

Table I
PERCENTAGES OF INJECTED FAULTS OVER 10000 RUNS, WITHOUT ANY UNDERLYING OS. THE FIRST COLUMN SHOWS THE PERCENTAGE OF INJECTED FAULTS DURING RSA-CRT COMPUTATIONS, WHILE THE SECOND COLUMN REPORTS THE NUMBER OF FAULTS EXPLOITED TO FACTOR THE MODULUS.

Module Size	Faulted RSA Computations	Exploitable Faults
512	7%	3%
1024	12%	8%
2048	25%	19%

occourred faults since the modular exponentiation requires a greater number of `load` operations due to the increased number of multiple precision multiplication operations.

Willing to investigate a scenario closer to a real world implementation, we decided to mount an attack while running the binary over a full fledged Linux 2.6.15 kernel, enabling the 16KiB data cache embedded in the ARM9 microprocessor.

Table II shows the results of the attack. Coherently with the

Table II
PERCENTAGES OF INJECTED FAULTS OVER 10000 RUNS, RUNNING ON LINUX 2.6.15. THE FIRST COLUMN SHOWS THE PERCENTAGE OF INJECTED FAULTS DURING RSA-CRT COMPUTATIONS, WHILE THE SECOND COLUMN REPORTS THE NUMBER OF FAULTS EXPLOITED TO FACTOR THE MODULUS.

Module Size	Faulted RSA Computations	Exploitable Faults
512	6.6%	4.6%
1024	5.4%	5.0%
2048	39%	39%

previous results, the gap between the injected and exploitable faults closes as the module size grows. The steep increase in the success rate of the attack when moving up from 1024 to 2048 bit of modulus size may be ascribed to the lapsing of the effectiveness of data cache, which in turn forces the CPU to load the required values from the main memory, thus raising the fault occurrence rate. The rates of successful attacks for the 2048 bit modulus are even higher than the previous experiment where no operating system was present. This is to be ascribed to the frequent register spill operations forced by the multitasking operating system, which lead to extra `load` operations of the values elaborated in the algorithm.

In order to evaluate a well known and widespread open source implementation of RSA, we decided to mount the last voltage underfeeding attack to RSA-CRT using OpenSSL 0.9.1i [18] on the previous test set. In the attacked implementation both message blinding and signature verification attack countermeasures were disabled. The significant difference between the results in Table III and the previous ones lies in the fact that the OpenSSL library has a more fault sensitive internal structure due to a deeper layering of the encryption primitive calls. The result shown in this section prove that our fault model is practically viable in order to successfully deliver the Bellcore attack with a reasonable number of induced faults. The vast experimental campaign demonstrates the feasibility on a widely deployed platform constituted by Linux running

Table III
PERCENTAGES OF INJECTED FAULTS OVER 10000 RUNS, RUNNING ON LINUX 2.6.15. THE FIRST COLUMN SHOWS THE PERCENTAGE OF INJECTED FAULTS DURING OPENSSL RSA-CRT COMPUTATIONS, WHILE THE SECOND COLUMN REPORTS THE NUMBER OF FAULTS EXPLOITED TO FACTOR THE MODULUS.

Module Size	Faulted RSA Computations	Exploitable Faults
512	7.27%	6.77%
1024	4.4%	4.2%
2048	13.27%	11.73%

on an ARM9 microprocessor.

B. Evaluation of the e-th Root Extraction Attack

The second experimental campaign was conducted in order to ascertain the possibility of extracting the message from a ciphertext through the technique described in Section IV-B. The platform used for the experiment was the same employed for the second experiment of the previous section, that is a C-code implementation of RSA based on Montgomery Multiplication running on Linux 2.6.15. This time, the algorithm employed was a plain square-and-multiply modular exponentiation used to encrypt a message with a full sized public exponent e. For each computation, the input message was mapped into the Montgomery domain before the exponentiation and was mapped back at the end of the computation.

Considering modulus sizes of 512, 1024, and 2048 bits respectively, we underfed the supply power line of the ARM9 microprocessor and completed for each of them an experimental session with 1000 faulty runs of the RSA encryption primitive.

Let $T = 4 \lceil \log_2 e \rceil / w$ be the number of possible positions of a fault injected in the public exponent e, where w is the word length of the microprocessor (i.e. 32 bit). The constant factor 4 was due to the four possibly different alignments of the exponent e in the main memory caused by the compiler.

For each run, we needed to iterate the plaintext retrieval algorithm (Algorithm IV.1) at most T times and check through re-exponentiation if the retrieved message was correct. In this way all the faulty ciphertexts generated by errors which did not alter the exponent were easily recognized.

Table IV shows in the first column the percentage of exploitable faulty computations out of 1000 faulty runs of the RSA encryption primitive. The second column reports the time needed to execute a single run of Algorithm IV.1 on an Intel Core 2 Quad E6600 clocked at 2.4 GHz.

Table IV
ROOT EXTRACTION SUCCESS RATE OVER 10000 INJECTED FAULTS

Modulus	Exploitable Faults	Single Check and Retrieval Time
512	62.77%	0.263s
1024	20.23%	3.9845s
2048	36.42%	101.112s

Taking into account the high success percentages shown in the table together with the low computation times we state that the attack is feasible in practice. The worst case recovery time does not exceed 5 minutes and the average number of required faults is not greater than 5, since a single exploitable fault leads to the recovery of the whole enciphered message.

VI. CONCLUSION

In this paper we have presented a new fault injection model relying on constantly underfeeding a general purpose microprocessor. We have characterized in full the new type of induced faults in both position and corruption patterns, splitting the effects into two classes: *data corruptions* and *instruction swaps*. The most appealing features of the model are the cheapness, the ease of induction and the absence of forecoming hurdles bound to the evolution of the chip building techniques. The experimental campaign conducted proved that our fault model is practically viable in order to successfully mount both the Bellcore and the e-th root extraction attacks with a reasonable number of induced faults. We foresee as future developments in this field the practical implementation of the secret key extraction attack, the evaluation of the implementative cost of the possible countermeasures required to thwart this attack and the evaluation of the effectiveness of the existing ones. Another interesting direction of research is represented by the application of this fault model to different cryptographic primitives such as AES and pairing algorithms.

ACKNOWLEDGEMENTS

This work was partially supported by MIUR in the framework of the PRIN SESAME project.

REFERENCES

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006.

[2] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," in *CHES*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Çetin Kaya Koç, and C. Paar, Eds., vol. 2523. Springer, 2002, pp. 2–12.

[3] J.-M. Schmidt and M. Hutter, "Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results," in *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*, J. W. Karl C. Posch, Ed. Verlag der Technischen Universität Graz, 2007, pp. 61 – 67.

[4] J.-M. Schmidt and C. Herbst, "A Practical Fault Attack on Square and Multiply," *FDTC*, vol. 0, pp. 53–58, 2008.

[5] N. Selmane, S. Guilley, and J.-L. Danger, "Practical Setup Time Violation Attacks on AES," in *EDCC-7 '08: Proceedings of the 2008 Seventh European Dependable Computing Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 91–96.

[6] G. Piret and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD," in *CHES*, ser. Lecture Notes in Computer Science, C. D. Walter, Çetin Kaya Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 77–88.

[7] F. Amiel, C. Clavier, and M. Tunstall, "Fault analysis of dpa-resistant algorithms," in *FDTC*, ser. Lecture Notes in Computer Science, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds., vol. 4236. Springer, 2006, pp. 223–236.

[8] ARM, "ARM9 Family of General-Purpose Microprocessors, ARM926EJ-S Technical Reference Manual." [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198e/DDI0198E_arm926ejs_r0p5_trm.pdf

[9] STMicroelectronics, "SPEAr Head200, ARM926, 200k Customizable eASIC Gates, Large IP Portfolio SoC," May 2009. [Online]. Available: http://www.st.com/stonline/products/literature/bd/14388/spear-09-h042.htm

[10] W. D. et al., “Das U-boot Bootloader,” May 2009. [Online]. Available: http://www.denx.de/wiki/U-Boot

[11] D. Boneh, R. A. DeMillo, and R. J. Lipton, “On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract),” in *EUROCRYPT*, 1997, pp. 37–51.

[12] A. K. Lenstra, “Memo on RSA Signature Generation in the Presence of Faults,” September 1996.

[13] V. Shoup, “A proposal for an ISO-Standard for Public Key Encryption (version 2.1), manuscript,” December 2001. [Online]. Available: http://shoup.net/papers/

[14] F. Bao, R. H. Deng, Y. Han, A. B. Jeng, A. D. Narasimhalu, and T.-H. Ngair, “Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults,” in *Proceedings of the 5th International Workshop on Security Protocols*. London, UK: Springer-Verlag, 1998, pp. 115–124.

[15] P. L. Montgomery, “Modular Multiplication Without Trial Division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985. [Online]. Available: http://www.jstor.org/stable/2007970

[16] C. H. Kim and J.-J. Quisquater, “How can we overcome both side channel analysis and fault attacks on RSA-CRT?” in *FDTC ’07: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 21–29.

[17] R. B. Rsa, M. Ciet, and M. Joye, “Practical Fault Countermeasures for Chinese,” in *In Proc. FDTC*, 2005, pp. 124–131.

[18] M. J. Cox, R. S. Engelschall, S. Henson, and B. Laurie, “The OpenSSL Project,” May 2009. [Online]. Available: http://www.openssl.org/