Low Voltage Fault Attacks to AES and RSA on General Purpose Processors

Alessandro Barenghi*, Guido Bertoni‡, Luca Breveglieri*, Mauro Pellicioli§, Gerardo Pelosi†
* DEI - Politecnico di Milano, Milan, Italy
Email: {barenghi,brevegli}@elet.polimi.it
‡STMicroelectronics, Agrate Brianza, Italy
Email: guido.bertoni@st.com
§Politecnico di Milano, Milan, Italy
Email: mauro.pellicioli@mail.polimi.it
†DIIMM - Università degli Studi di Bergamo, Dalmine (BG), Italy
Email: {gerardo.pelosi@unibg.it}

***Abstract*—Fault injection attacks have proven in recent times a powerful tool to exploit implementative weaknesses of robust cryptographic algorithms. A number of different techniques aimed at disturbing the computation of a cryptographic primitive have been devised, and have been successfully employed to leak secret information inferring it from the erroneous results. In particular, many of these techniques involve directly tampering with the computing device to alter the content of the embedded memory, e.g. through irradiating it with laser beams.**

In this contribution we present a low-cost, non-invasive and effective technique to inject faults in an ARM9 general purpose CPU through lowering its feeding voltage. This is the first result available in fault attacks literature to attack a software implementation of a cryptosystem running on a full fledged CPU with a complete operating system. The platform under consideration (an ARM9 CPU running a full Linux 2.6 kernel) is widely used in mobile computing devices such as smartphones, gaming platforms and network appliances.

We fully characterise both the fault model and the errors induced in the computation, both in terms of ensuing frequency and corruption patterns on the computed results.

At first, we validate the effectiveness of the proposed fault model to lead practical attacks to implementations of RSA and AES cryptosystems, using techniques known in open literature. Then we devised two new attack techniques, one for each cryptosystem. The attack to AES is able to retrieve all the round keys regardless both their derivation strategy and the number of rounds. A known ciphertext attack to RSA encryption has been devised: the plaintext is retrieved knowing the result of a correct and a faulty encryption of the same plaintext, and assuming the fault corrupts the public key exponent. Through experimental validation, we show that we can break any AES with roughly 4 kb of ciphertext, RSA encryption with 3 to 5 faults and RSA signature with 1 to 2 faults.

I. Introduction

Key requirements during the design of a complex system are its *fault resistance* and *fault tolerance*: the former refers to the ability of a system of being able to work properly in an environment where hazards are present; the latter represents the property of being able to exhibit a controlled behaviour when faults occur. A key point in the design of fault tolerant systems is to obtain a *graceful degradation* of the performances, that is to design devices able to gradually lose functionality instead of exhibiting *catastrophic failures* when faults occur.

Taking into account the fact that also complex computational systems are designed to fail gracefully, it is sensible to assume that there exists a category of faults which will disrupt only a very small amount of the ongoing computation, leaving the rest of the system untouched, regardless of its complexity. These errors are usually corrected through the use of redundancy either in the form of replicated units or correction codes. However these are usually employed only in industry grade chip design, where it is expected that the component will work in a hazardous environment. Consumer grade devices usually are not designed with these concerns due to the milder expected deployment setting and the additional costs involved which would drive higher the price of the unit.

Regardless of the presence of error correcting countermeasures, it can be noticed that the graceful degradation property may sometimes be an undesired one, i.e. when the computing device is performing security related tasks, in particular when calculating cryptographic primitives.

During the computation of cryptographic primitives, small, traceable changes in the behaviour, strongly correlated to the internal state of the device, represent a serious threat to the security they are supposed to provide. In particular, if it is possible to bind with precision the induced faults to a specific, non catastrophic, change in the output, the leaked information may be used in order to break the cryptoscheme.

This attack methodology fully embodies the side channel attack paradigm, since it exploits informative content leaked from an implementation of the cipher, which is strongly related to the encryption or decryption process.

A number of hazards which can be introduced into the working environment of a device will cause faults. In particular irradiation with light or lasers, glitches on the clock signal or in the power supply, have been used until now in order to successfully induce controlled faults in small computing components such as microcontrollers or memory storage devices [26], [27]. All these techniques rely on the fact that the disturbed device is reasonably small and suffer from the evolution in lithographic etching technologies, which enable higher clock rates. This makes more difficult to correctly irradiate the sensitive zone

of the chip or to disturb the execution during a specific clock cycle.

Willing to provide experimental evidences of the correctness of this perspective, this work presents the first characterization of a controlled fault model in a complex computational platform: specifically a general purpose ARM9 CPU running a full fledged Linux operating system and its applications to cryptanalisys. The choice of this platform was driven by its widespread adoption as computational platform in almost all current smartphones, network appliances, portable gaming platforms and low power computers.

Our choice of fault induction technique was done aiming at picking a fault injection methodology independent of the technological progress, able to act on complex systems and as cheap as possible, in order to obtain the widest applicability.

Our chosen methodology is to underfeed constantly a circuit during the whole computation time: an approach which until now has never been tried for systems including a general purpose CPU. Underfeeding a circuit is a known cause of faults, and lends itself well to the purpose of this research since it represents a simple and effective alteration of the environment, which can be achieved without leaving any evidences of having tampered with the device. This is an advantage with respect to irradiation techniques, which require delicate procedures in order to remove the packaging from the silicon chip to be effective. Moreover, the equipment required in order to explore the effect of a gradual decay of the quality of the working environment, is reasonably cheap and does not compromise the correct working of the device afterwards.

In the next Section we are going to provide a summary of the most pertinent open literature contributions concerning practical fault attacks on microcontrollers. A comprehensive literature archive on the subject is provided by [9]–[11].

The target of the first part of this work (Section II) is to provide a full characterisation of the model of faults occurring when gradually lowering the power feed input to the circuit, and the errors they induce on the outputs. Since the system is able to operate at more than one working frequency for power saving issues, the fault characterisation takes into account this factor, too.

In the second part of this work consists of Section IV and Section V. In Section IV we analyze the AES cryptosystem and recall a known attack which correctly fits our fault model. Moreover we propose a new attack technique able to recover the full cipher key for any AES cipher and is not necessarily bound to our fault model. In Section V we recall a well known attack which is employable with our fault model and propose a new one able to successfully decipher an RSA ciphertext assuming a fault injection in the public exponent and the possess of a correct ciphertext.

Part of the RSA attack techniques and of the fault model characterization is described in [7] of which this work is an extension.

The third part of this work (Section VI) reports the results of the experimental campaign conducted in order to ascertain the applicability of the algorithmic methods presented in the previous sections, thus validating the practical cipher breaking abilities proposed.

Finally, in Section VII we draw the conclusions, summarizing our original contributions.

II. Fault characterisation and induced errors

This section provides a complete characterization of the faults happening when a general purpose CPU is constantly underfed in terms of position, shape and timing and subsequently delineates the induced errors on the computed outputs together with the methodology followed in order to build it. A complete description of the working environment is provided in order to properly outline the workflow we followed in order to coalesce the new fault and error model.

A. CPU Architecture and Experimental Settings

The processor architecture taken into account in this study is the ARMv5TE, in particular the version implemented by the ARM9E microprocessor. This choice was driven by the vast diffusion of this CPU, which is nowadays the dominant choice for smartphones, network appliances, portable gaming platforms and low power computers, thus quite likely to be used also to compute cryptographic primitives while in possess of a possible attacker.

Our target chip is an an ARM926EJ-S [5]: a 32-bit RISC Harvard architecture CPU with 16 general purpose registers and a 5 stage pipeline. The ARM processor has a full MMU and separate data and instruction caches each 16 Kb wide, coupled with a 16 entry write buffer which avoids stalls in the CPU when memory writebacks are performed. In particular the ARM926EJ-S is also endowed with a hardware Java bytecode interpreter able to run directly Java bytecode.The richness of the available features justify the vast popularity achieved by this model in consumer mobile devices.

The CPU is embedded in a system on chip mounted on a development board, specifically a SPEAr Head200 [30] built by ST Microelectronics, which is used as reference board to design ARM based devices equipped with 64 MB DDR RAM clocked at 133 MHz, 32 MB of on-board Flash storage, 2 USB Host ports, a RS-232 Serial Interface and 100 Mbps Ethernet network card. The system is endowed with an U-Boot [16] embedded bootloader, which is able to load the binary to be run via TFTP [18] protocol. This allows the board to either run a specific binary, compiled to be independently executed on the ARM9 CPU, or to boot a full fledged operating system. In the following experiments, raw binaries were employed in order to characterise with precision the fault model of this system. On the other hand, for the sake of practical applicability, all the attacks to cryptosystems were lead with a full vanilla Linux 2.6 kernel (DENX distribution) employing an NFS [6] partition as root filesystem. All the binaries were compiled with the GCC 3.4 based development toolchain for ARM9 provided by Codesourcery [12]. All the fault characterisation tests were performed on more than one instance of the board, reporting analogous results: for the sake of clarity we present the results on a single board.

Two experimental workbenches were employed during this work: the first, aimed at producing a precise characterisation of the effect of power supply lapses was endowed with a high precision power supply. The second one, aimed at carrying the attacks with a lower budget, employed less expensive equipment, without loss in the efficacy of the attacks. The two workbenches are identical except for the change in the Power Supply Unit (PSU).

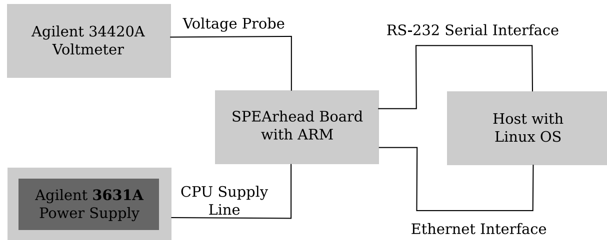

Figure 1. Workbench used in order to accurately characterise the voltage range

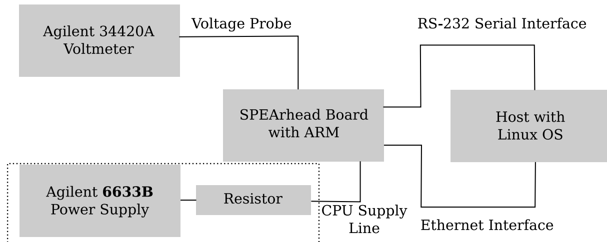

Figure 2. Workbench used in order to perform the error characterisation and the attacks

The first workbench is depicted in Figure 1: the board was at first fed through an Agilent E3631A PSU [3] having a precision of 1mV while on the second workbench, depicted in Figure 2, was fed through an Agilent 6633B [2] power supply with a 0.01V precision. In order to achieve rough sub-centivolt precision we used a resistive partitor with a common commercial grade resistor: whilst this solution does not provide the same accuracy on controlling the voltage as using a high precision PSU, it proved effective enough to successfully bring all the attacks. All the voltage measures were taken with an Agilent 34420A [1] voltmeter with a nV precision probe which was already available to us, nonetheless the needed precision was only up to 0.5mV. The board was connected to a PC both with a null modem cable and an Ethernet cable: the first provided an interface with the Linux shell running on the ARM chip, while the second was used to provide the network connection needed for both booting the board via TFTP and providing the storage via NFS.

B. Graceful Degradation of Outputs

The first experiment run on the target chip was aimed at investigating whether the appearance of the errors in the system followed the gradual behaviour we expected. The ARM9 processor 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 critical importance for the correct execution of the binaries run on the device.

Willing to detect the frequency of the appearance of faults in order to determine how fast they appear during the execution of a program, 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;
     } /* if */
  } /* if */
} /* for */
```

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. This program was run multiple times while decreasing the voltage of the power supply of 1mV at a time: 500 thousand runs were performed for each voltage level probed and the results output by the code were stored. Figure 3 represents the percentage of correct computations over 500 thousand runs, for each voltage level probed. The errors in the output grow linearly with the lapse in the voltage supply, thus confirming our hypothesis of a gradual degradation in the quality of the results. The dashed line in the figure points out the voltage point where the faulty computations are in the same number as the correct ones.

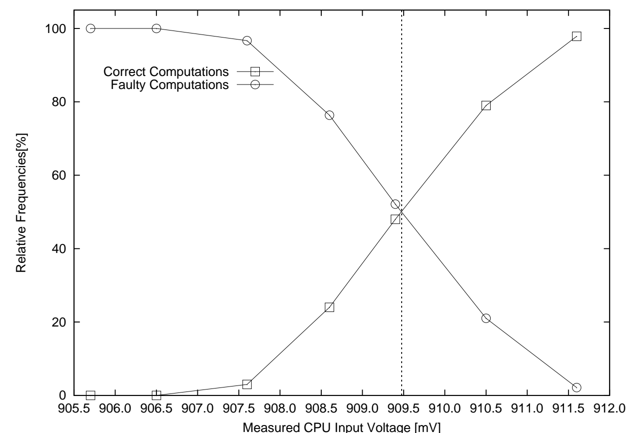

Figure 3. Percentage of correct and wrong computations averaged over 500 thousand runs.

After ascertaining that the faulty computations of a program are happening slowly, we moved on to consider the number

of faults appeared during each single computation. Since our target will be to inject single faults, we are interested in seeing whether the errors in the outputs of the former computations were caused by one or more faults during their executions.

Re-classifying the faulted runs from the former experiments according to the number of faults, it is possible to observe, as shown in Figure 4, that it is present a 1 mV wide voltage range where only a single fault happens with dominant probability. Moreover, in the adjacent 1 mV range, 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, while keeping still in the voltage range where less than a half of the computations is faulted. When working under the 50% faulty computation threshold, the number of possible faults starts growing, and multiple fault scenarios start dominating the fault profile as it can be expected in a degrading environment. After lowering even more the power supply voltage, the board stops outputting data from the RS-232 interface used to communicate, thus preventing the results from being collected. It is therefore clear that, in the voltage region where the correct computations are the majority and the faulty ones begin to appear, the faults occurring in the computations are single and not represented by small bursts. Whilst results until now have been obtained

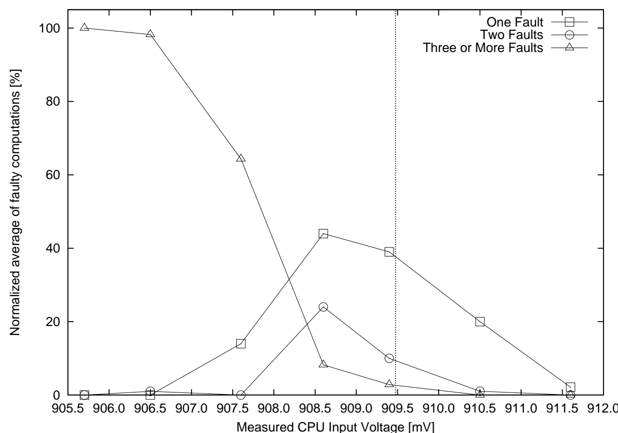

Figure 4. Distribution of the quantity of the injected faults as a function of the voltage.

while keeping the number of machine instruction per binary fixed for each measurement, and since there is no specific timing in the insertion of the hazard causing faults, it is reasonable 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 its computation. This is not an obstacle to injecting single faults, since the fault incidence per single execution may be tuned for different length binaries lowering or raising the voltage accordingly.

C. Fault Type Characterisation

Having ascertained the possibility of injecting a single fault per computation, we proceed to investigate which kind of fault has actually hit the computation, through characterising its effect on the executed code.

Analysing the binary at assembly level, all the possible instructions executed by a CPU 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. Since memory instructions represent the most expensive operation class in terms of power consumption, 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 in order to avoid any memory operations while computing the result. Only the instruction cache of the CPU was enabled, leaving thus only the data loading operations uncached. 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 drop. All the experiments were conducted through collecting erroneous outputs after the binary had been running for a couple of seconds: this allowed the caching of the whole probe program due to its tiny size.

The low voltage fault immunity shown by the CPU registers is to be ascribed to the low capacitance design of their implementation, which yields faster switching times than the average logic, thus compensating partially for the slowdown induced by the lapse in the supply. This feature is mandated by the architectural need of providing fast accesses to the component, which is critical in order to design efficient units and is thus to be expected in all the common CPUs.

Since only the memory operations are affected by faults, the next natural step was to check if all the instructions (i.e. both `loads` and `stores`) were equally affected. In order to distinguish which kind of memory instructions are affected by faults, it is possible to use a register-held value as a fault free reference for computing checks. We set up a probe program which loaded and saved from the memory zero and one filled words, and ran it multiple times while sweeping the whole voltage range of the previous campaign.

The only instructions to report faulty results were the `load` instructions, while all the write instructions were safely performed. This behaviour may be sensibly ascribed to the fact that only the memory operations which store values on the underfed part (i.e. the `load` operations which store information in the registers) suffer from the lapse in the power supply. On the other hand the `store` operations place the data on a properly fed part of the architecture. Moreover, the ARM926EJ-S (similarly to all modern CPUs) is endowed with a 16 entry write buffer between the CPU and the memory, in order to perform aggressive instruction reordering. The presence of the buffer helps cutting down the capacitive load of the path to the memory and thus helps to perform correct writes.

D. Fault Spread

The experiments run up to now characterise 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 behaviour of the `load` instructions depends on the referenced memory address from which the load is performed.

In order to understand this key point, a probe program was designed to overwrite a one million 32-bit word array with `1`s, and subsequently to check the values which were loaded back into the registers, while keeping the voltage in the single fault functioning range. To avoid any possible disturbances, during this test the data cache of the ARM9 processor was disabled, thus forcing the CPU to load each value from the main memory.

Figure 5 shows the number of faults occurred while performing 10^6 `load` operations of a one-filled 32-bit integer from the aforementioned array. In order to analyse the data, the probed memory was clustered into 40 kB wide zones. We encountered 1864 faulty `load`s while running the program, thus we are expecting an average of 18.64 faults per zone in case the faults fall uniformly over the address space. The dashed line in Fig. 5 indicates the expected number of faults occurring for each zone, assuming a uniform distribution of the faults over the memory. To confirm the hypothesis of a uniform distribution of the faults over the whole address space, we modelled 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 higher than 99.99%.

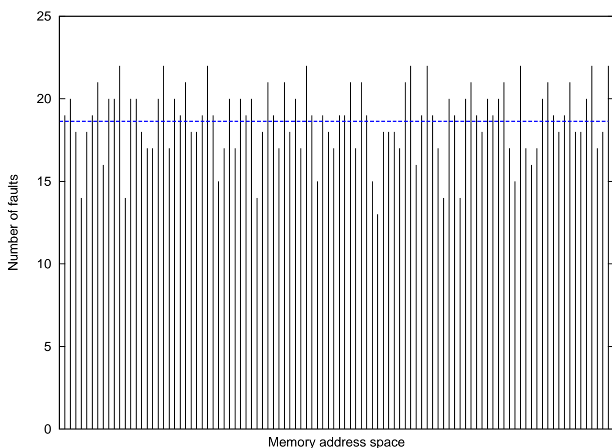

Figure 5. Distribution of the quantity of the injected faults as a function of the position in the address space. The dashed line indicates the expected average value in the hypothesis they are uniformly spread.

E. Error Characterisation and Effects of Frequency Scaling

After fully characterising the frequency and the conditions of occurrence of the faults, the natural target of the investigation becomes characterising the kind of errors induced in the computations by the faults. Through analysing the data collected during the last experiment, we were able to notice that all the faulty loads were affected by flip downs in the bit value loaded. Willing to ascertain if only flip downs were possible, we ran the same memory exploration program changing the loaded value to both a zero-filled 32 bit word and to some random values. In all the cases only bit flip downs occurred, and there was never a single instance of a flip up. When analysing the position of the bits which are flipped down during the faulty loads, we detected that only a very small number of flip down patterns were present (namely 4) and one of them accounted for more than the 50% of the fault occurrences. When repeating the tests on different boards, the recurring patterns changed, but their number and frequency did not, allowing us to deduce that some bits within a word are more sensible to flip downs when the CPU performs a load operation while undervolted. This may be ascribed to the different capacitive load of the signal lines of the CPU, is due to routing issues which may force the I/O lines for a register to have different lengths.

Willing to complete the analysis of the error patterns, we decided to run the same error pattern detection experiment while varying the CPU frequency according to the allowed working range. Through piloting the clock generator on the board we were able to scale the frequency of the CPU mimicking a real world scenario where the ARM processor is often run at frequencies lower than the maximum allowed in order to save power. It is possible to choose among a number of frequency settings which alter globally the working frequency through writing in the PLL generator (Phase Locked Loop) register interface. This causes both the board and the CPU to switch their working frequency: the board is ran at the frequency written in the register while the CPU is run at twice the set value.

The clock setting is retained until the board is rebooted, but it is possible to customise the deployment model in order to either lock it permanently or to leave the frequency scaling to the operating system. We wanted to investigate whether the faulty behaviour had any changes while working in different frequency environments, therefore we locked the running frequency in order to collect homogeneous samples of the behaviours.

In a real world scenario this may happen to be the actual working environment since it is quite common to lock the CPU frequency at a lower value than the maximum allowed in order to save power. The possibility that the frequency choice is left to the operating system does not impair our analysis since the CPU will be running at a constant frequency in discrete timeslices, thus reporting the same faulty behaviour per-timeslice.

Table I reports the result of the experiments performed and shows how, regardless of the frequency at which we are running, the error patterns are few and characterised by one, which is dominant as far as the occurrence frequency goes.

F. Effects of the Errors on the Computation

After fully characterising the kind of errors induced by our fault injection technique, the last part of this enquire sums

Table I
NORMALIZED FREQUENCIES OF THE DIFFERENT ERROR PATTERNS PER CPU CLOCK SETTING

CPU Clock [MHz]	Loaded Pattern	[%]
140	{3}	58.76
	{21}	10.01
	{3,21}	31.22
224	{21}	100.0
266 (Full)	{10,16}	38.72
	{10}	53.23
	{9,10,11,12,15,16,21}	2.95
	{9,10,16}	3.34

up the possible effects on the computation caused by such errors. Albeit originating from the same cause, i.e. faulty `load` operations, we may distinguish two different effects of the faults depending on whether the load was related to an instruction fetch or to a data load. In the latter case a *data load error* occurs, while, in the former case *instruction swapping* may occur. For the sake of clarity, we will deal separately with the two outcomes in order to distinguish their possible effects on the computation of cryptographic primitives.

1) Data Load Errors: Data related errors 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 within 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 $\varepsilon = k\,w + i$ with w equal to the word length, $i \in [0, w-1]$ and $k \in [0, \frac{t}{w}]$. These changes in the loaded value are very precise in the way they cause the alterations and therefore may easily leak sensitive information, as it will be shown by successfully conducting attacks in the next sections.

2) Instruction Swap Errors: Bit flipping during an instruction fetch may alter either the opcode or the arguments of the instructions, depending on which bit is affected by the flip down. 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. In the case of the ARM architecture, this may result in either a swap of one kind of instruction with another one or in a reversal of the triggering condition of a conditional instruction.

An example of a possible instruction swap through a single bit flip-down 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, thus leading to the possible computation of a weaker version.

Given that the ARM architecture allows the conditional execution of all the arithmetical-logical instructions, and stores the kind of condition in a suffix of the opcode, as Figure 6 depicts, it is possible for the error to actually invert the condition of the predicate instruction. For instance, in the following code sample, the two instructions share the same opcode except for the zero-condition bit setting :

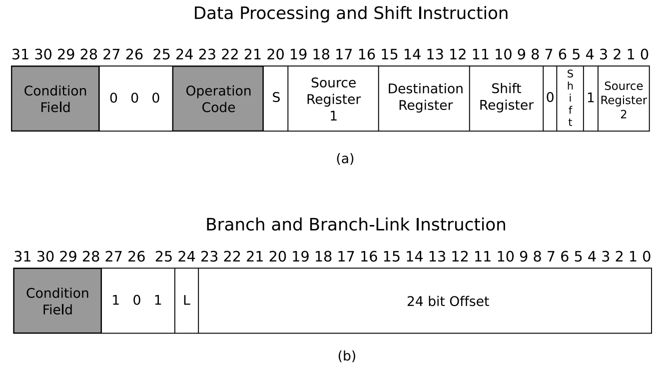

Figure 6. Excerpt from the ARM ISA description [5] depicting branch and data processing instructions. Grayed areas point out the interesting fields during fault injection.

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

This behaviour could lead to mis-executions of the algorithm leaking significant content, especially if the conditional instructions are directly related to the key value (e.g. in the common square and multiply algorithm used to perform fast exponentiation).

Moreover, since also the branch instructions rely on the same condition bits of the common conditioned, the control flow of the program may be equally altered if the condition bit of a branch is flipped like in the following sample :

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

This kind of alteration may lead to substantial control flow alterations, which can turn into lowering the number of times a loop is executed or skipping it altogether, thus providing substantial reductions in the complexity of a cryptographic primitive computed on the device.

We have been able to reproduce all the aforementioned alterations on our chip samples through running the probing programs without enabling the instruction cache and thus the code loading operations to be performed directly from the memory. 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. Nonetheless, since our methodology of probing does not compromise the computing architecture, it is possible to scan a sample chip in order to understand which of these code mutations are performed and devise specific attacks.

III. RELATED WORK

The open literature does not provide any examples of attacks to a general purpose CPU ; instead all the known contributions are focused on smaller computing devices such as micro-controller and smart cards. We may distinguish the practical attacks to real world systems according to the techniques

proposed to inject the faults, since these directly affect the fault model attainable and therefore the applicable attacks.

A first technique relies on altering the state of a Microchip PIC16F84 by irradiating directly the silicon die through the use of a concentrated light beam, either polarized (laser beams) or unpolarized (common flashes) [29]. The beam is usually timed in order to achieve changes in the values stored in SRAM cells 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 very difficult to comply with, since the new etching technologies are able to print sub-visible wavelength wide gates. Moreover, this fault induction technique relies on depackaging the chip thus leaving sensible evidences of the tampering involoved.

A reasonable way to avoid depackaging the chip is the EM-disturbances based faults injection recent technique proposed by Schmidt and Hutter [25] using a 8-bit microcontroller with 256 Bytes RAM as testbed device. The injection of faults is 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, but there is no way to direct the fault in a precise manner. 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 smart cards, therefore altering the control flow of the algorithm, possibly leaking sensitive information. A description of such a technique is presented in [4].

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 AVR Microcontroller through this technique is given in [24].

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. While this has never been tried for a full CPU, in [27] the authors report a faulty behavior of the lines at the end of the longest combinatorial cones of an ASIC AES co-processor embedded in a smart card, and exploit it in order to carry a successful attack using the method proposed by Piret *et al.* in [23].

For a full fledged collection of works on the subject, we refer the interested reader to [9]–[11].

IV. Symmetric Key Applications - Attacks to AES

A. Overview of the AES Block Cipher

The Advanced Encryption Standard (AES) [22] is a symmetric cryptographic algorithm originally requested and adopted by the National Institute of Standards and Technology (NIST) to replace the ageing Data Encryption Standard (DES) [21]. AES is an iterated block cipher which corresponds to a block size restricted version of the Rijndael [15], and can encrypt and decrypt 128-bit wide plaintext blocks using a key, whose size may be 128-bit, 192-bit or 256-bit. The Rijndael cipher was chosen among the other final candidates due to its ease of implementation on a wide range of 8-bit to 32-bit computing platforms as well as to its being amenable to high performance ad hoc hardware implementations. Moreover, the clarity and compactness of its design allowed a wide cryptanalytic scrutiny that helped to strengthen the confidence in its security level. In software, AES can be implemented with a fully symmetric structure using only bitwise XOR operations, table-lookups and 1-byte shifts. [15]

The cipher is designed to execute a number of round transformations on the input plaintext, where the output of each round is the input to the next one. The number of rounds r is determined by the key length: 128-bit uses 10 rounds, 192-bit 12 and 256-bit 14. Each round is composed by the same steps, except for the first where an extra addition of a round key is inserted and for the last where the last step (MixColumn) is skipped. Each step operates on 16 bytes of data (referred as the internal *state* of the cipher) generally viewed as a 4×4 matrix of bytes or an array of four 32-bit words, where each word corresponds to a column of the *state* table. The four round stages are: AddRoundKey (XOR addition of a scheduled round key for blending together the key and the state), SubByte (byte substitution by an S-box, i.e. a full lookup table for a non linear function), ShiftRow (cyclical shifting of bytes in each row to realise a inter-word byte diffusion), and MixColumn (linear transformation which mixes column state data for intra-word inter-byte diffusion).

The specification of the AES algorithm includes the description of a a KeySchedule procedure which is responsible for the computation of each 16-bytes round key k_j given the global input key k. The AES key scheduling process expands the cipher key k in a total of 4(r+1) 32-bit words with $r \in \{10, 12, 14\}$ according to whether the cipher key length s is equal to 4, 6 or 8 words, respectively. The resulting key schedule consists of a linear array of 32-bit words, denoted $W[0, \ldots, 4(r+1)-1]$. The first s words of W are loaded with the user supplied key. The remaining words of W are updated according to the following rule:

for $i = s, \ldots, 4(r+1)-1$ **do**
  if $i \equiv 0 \mod s$ **then**
    $W[i] = W[i-s] \oplus S[W[i-1] <<< 8]) \oplus RCON[i/s]$
  else if $s = 8$ **and** $i \equiv 4 \mod s$
    $W[i] = W[i-s] \oplus S[W[i-1]]$
  else

$$W[i] = W[i-s] \oplus W[i-1]$$

Where $RCON[\ldots]$ is an array of predetermined constants, $S[\ldots]$ is the array of precomputed constants corresponding to the substitution map of the cipher, and $<<<$ denotes the rotation of one byte of the word to the left.

The enciphering procedure is amenable to several software implementations which trade-off memory and computational resources in order to obtain the best performance for the specific architecture.

Specifically, the different steps of the round transformation can be combined in a single set of table lookups, allowing for very fast implementations on processors having word length of 32 bits or greater [15]. Let us denote with $a_{i,\,j}$ the generic element of the state table, with a the generic value of a byte variable, with $S[0,\ldots,255]$ the 256-bytes of the S-box table and with $\bullet$ a $GF(2^8)$ finite field multiplication [15]. Let T_0, T_1, T_2 and T_3 be four lookup tables, each viewed as a 256 sequence of 32-bit words, containing results from the combination of the round operations as follows:

$$T_0[a] = \begin{bmatrix} S[a]\bullet 02 \\ S[a] \\ S[a] \\ S[a]\bullet 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \\ S[a] \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \end{bmatrix}$$

These tables are used to compute the round stages operations as a whole, as described by the following equation, where k_j is the j-th word of the expanded key and $A_j = \langle a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j}\rangle$ is the j-th column of the state table considered as a single 32-bit word (with abuse of notation: $A_j = A_{j \bmod 4}$, $a_{i,j} = a_{i,j \bmod 4}$):

$$A_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \oplus k_j$$

The four tables T_0, T_1, T_2 and T_3 (called *T-boxes* from now on) make up for 4 KB of storage space and their main goal is to avoid performing the MIXCOLUMN and INVMIXCOLUMN transformations as these operations, in the original definition of Rijdael algorithm, perform Galois Field multiplication by fixed constants which map poorly to general purpose processors in terms of performance.

Notably, in the final round of the cipher there is no MIXCOLUMN operation, and also the KEYSCHEDULE algorithm requires pure substitution operations. Whilst these facts could represent an impairment in the use of T tables, it is possible to extract efficiently the S table through proper masking of the T tables.

Since the T-boxes may be derived also through rotating each word of T_0 by i bytes, $T_i[a] = \text{ROTBYTE}(T_0[a], i)$, $i \in \{0,\ldots,3\}$, to reduce the active memory footprint used within each round, each column of the state table may be also computed as:

$$A_j = T_0[a_{0,j}] \oplus \text{ROTBYTE}(T_0[a_{1,j-1}], 1) \oplus$$
$$\oplus \text{ROTBYTE}(T_0[a_{2,j-2}], 2) \oplus \text{ROTBYTE}(T_0[a_{3,j-3}], 3) \oplus k_j$$

This variation reduces the lookup tables to a single 1kB one, thus lowering the burden on the caches, while incurring in a penalty of only three extra rotates per column per round with respect to the 4 T-box implementation.

Decryption requires different tables from the encryption, therefore an AES implementation able to perform both encryption and decryption may require up to 8 kB of additional memory, which may extend to 16 kB if the last round operations are realised with ad-hoc tables.

When employing general purpose CPUs, endowed with wide D-caches, the T-box implementation proves more effective since the memory access latency is lower than the computation time that would be required in place of each T-box lookup. On the other hand, in cache constrained environments a valid alternative to the use of T-boxes is the computation of the entire AES rounds on the processor, memorising only the S-box and the inverse S-box tables needed to perform the substitution operations.

B. Effects of the Low Voltage Induced Errors on AES

Given the error model on the loaded data presented in Section II-F, we may expect that the errors induced during the computation of the AES cipher affect the results through alterations in the values loaded during each memory lookup. In particular, being the corruption characterised by a single bit flip down, we may safely assume that only a single byte of the state of the cipher is affected by a lone fault. Since the attack strategy proposed by Piret and Quisquater in [23] for the AES-128 cipher works under the hypothesis of a single byte error, it fits correctly the error characterisation we provided in Section II-F, and thus provides a proper framework to lead a successful attack. The attack works under a known ciphertext assumption, only requiring pairs of faulty and fault free ciphertexts generated from the same plaintext for each pair. The goal is to derive the cipher key using the informative content of the last round key, which is feasible as far as AES-128 goes. We have extended the attack technique proposed in [23], to recover any round key from the AES cipher, regardless of the key scheduling algorithm (i.e. regardless from the fact that the round keys are computed through the standardised KEYSCHEDULE algorithm or filled completely with a much longer cipher key), the key length or the number of rounds (even if exceeding the number of rounds set by the standard).

C. Piret and Quisquater's Attack to AES-128

The error hypothesis assumed by Piret and Quisquater in [23] considers the corruption of a single byte value between the last and the last-but-one MIXCOLUMN computation. The standard sets the number of rounds for AES-128 to $r = 10$, expands the cipher key into $r + 1$ round keys and removes the MIXCOLUMN operation from the last round. Therefore a

Algorithm IV.1: BASE ALGORITHM

Input: $\Delta = \{ \langle\delta_0, 0, \ldots, 0\rangle, \langle 0, \delta_1, 0, \ldots, 0\rangle, \langle 0, \ldots, \delta_u, \ldots, 0\rangle, \ldots, \langle 0, \ldots, \delta_{15}\rangle \}$ with $0 \leq u \leq 15$, $1 \leq \delta_u \leq 255$, and $|\Delta| = 255 \times 16$.
$\Delta^{'} = \{ d \,|\, d \leftarrow \text{MixColumn}(\delta), \forall \delta \in \Delta \}$

Output: $\bar{k}$: last round subkey

Data: All the states are represented through a 4×4 matrix, the cells are enumerated from top-left to bottom-right

```
1  begin
2      Record a faulty ciphertext c̃ and a fault-free one c
       /* Set up of Candidate-Keys List   */
3      L ← ∅
4      foreach k ∈ {0, ..., 2^128 − 1} do
5          δ' ← InvSubByte(c⊕k) ⊕ InvSubByte(c̃⊕k)
6          if δ' ∈ Δ' then
7              L ← L ∪ { k }
       /* Key Selection Phase             */
8      while |L| > 1 do
9          Record a faulty ciphertext c̃ and a fault-free one c
10         foreach k ∈ L do
11             δ' ←
               InvSubByte(c ⊕ k) ⊕ InvSubByte(c̃ ⊕ k)
12             if δ' ∉ Δ' then
13                 L ← L \ { k }
14     return k̄     /*  L = { k̄ }         */
15 end
```

single corrupted byte value must be computed either during the execution of the 8th round ADDROUNDKEY operation or during the execution of the 9th round SUBBYTE and SHIFTROW operations.

Given a faulty ciphertext, $\widetilde{c} = \{\widetilde{c}_u, u \in \{0, \ldots, 15\}\}$, and a fault-free one, $c = \{c_u, u \in \{0, \ldots, 15\}\}$, the possible differences evaluated just before the last MIXCOLUMN add up to 255×16 different values. Such values can be listed through enumerating all the state tables resulting from changing a single fixed-position byte value, and then repeating the change for each one of the 16 bytes composing the state table, i.e.: $\Delta = \{ \langle\delta_0, 0, \ldots, 0\rangle, \langle 0, \delta_1, 0, \ldots, 0\rangle, \langle 0, \ldots, \delta_u, \ldots, 0\rangle, \ldots, \langle 0, \ldots, \delta_{15}\rangle \}$ with $0 \leq u \leq 15$ and $1 \leq \delta_u \leq 255$. The inter-byte diffusion operated by the MIXCOLUMN maps bijectively each difference value into another thus obtaining another set of differential state tables with the same cardinality of Δ: $\Delta^{'} = \{ \langle\delta^{'}_0, \ldots, \delta^{'}_u, \ldots, \delta^{'}_{15}\rangle \}$ with $0 \leq u \leq 15$ and $1 \leq \delta^{'}_u \leq 255$.

A base algorithm that summaries the main steps of the attack is described by Algorithm IV.1. The algorithm takes as input the list of all the differences that may occur just after the last MIXCOLUMN operation: $\Delta^{'}$. As a first step, the algorithm records a fault-free ciphertext c and a faulty one $\widetilde{c}$ of the same, unknown, plaintext. Then, for each possible value of the last round-key, k, computes the difference $\delta^{'}$ between the state tables corresponding to c and $\widetilde{c}$ just after the last MIXCOLUMN operation:

$$\delta^{'} = \text{InvSubByte}(c \oplus k) \oplus \text{InvSubByte}(\widetilde{c} \oplus k)$$

If $\delta^{'}$ is included in the set Δ' then the value of the corresponding subkey k is inserted in a list L of candidate keys. Subsequently, until L contains only a single key, another pair of faulty and fault-free ciphertext generated from an unknown plaintext is collected. Then, for each candidate key in L the differential value corresponding to the faulty and fault-free ciphertexts is computed. If the differential value is not included in $\Delta^{'}$ it is removed from the list of candidates, L. At the end of this sieving phase the list L will contain a single value for the last round key. Since the KEYSCHEDULE algorithm uses only invertible operations the knowledge of the last round key $\bar{k}$ is sufficient to retrieve the global input key.

Obviously, the computational complexity of Algorithm IV.1 is not practical since a scan over the whole key space is required (see lines 4–7). However, to initially fill the list of candidate keys L the authors of [23] proposed an experimental heuristic which considerably reduces the overall complexity of the differential fault attack in practise. Algorithm IV.2 reports the heuristic used to set up the candidate-keys list and replaces the impractical procedure reported in lines 4–7 of Algorithm IV.1.

The key intuition under the candidate-key sieving procedure is that: given the precomputed list $\Delta^{'}$ of all the possible differentials just after the last MIXCOLUMN and given a faulty and a fault-free ciphertext, if a candidate key allows to match a differential in $\Delta^{'}$ then such matching will hold (with high probability) also when considering the ciphertexts and the candidate key having non zero-values only in $x \geq 2$ byte positions. In such a way, it can be experimentally shown that the exploration space for a full-length candidate key shrinks very quickly. Actually, in order to set up the list L of candidate keys, Algorithm IV.2 considers two pairs of faulty and fault-free ciphertexts, i.e., $\langle c, \widetilde{c}\rangle$ and $\langle d, \widetilde{d}\rangle$ (lines 2–3).

Then, considering a copy of the ciphertexts, $\langle c^{'}, \widetilde{c}^{'}\rangle$ and $\langle d^{'}, \widetilde{d}^{'}\rangle$, where only the two left-most bytes have a non-zero value (lines 5–6), the algorithm fills a temporary list $L^{'}$ with candidate keys, k, having only the two left-most bytes with a non-zero value and such that the two left-most bytes of the differentials

$$\beta \leftarrow \text{InvSubByte}(c^{'} \oplus k) \oplus \text{InvSubByte}(\widetilde{c}^{'} \oplus k)$$
$$\gamma \leftarrow \text{InvSubByte}(d^{'} \oplus k) \oplus \text{InvSubByte}(\widetilde{d}^{'} \oplus k)$$

both match the two left-most bytes of any differential in $\Delta^{'}$ (lines 7–14).

For each key k in $L^{'}$ (line 17), a copy of the original ciphertexts having only the 2nd and the 3rd bytes with non-zero value is considered. Moreover, a temporary key $k^{'}$ having the 2nd byte copied from k and the 3rd byte assuming all values in $\{0, \ldots, 255\}$ is considered. If the 2nd and 3rd bytes of the computed differentials β and γ (lines 27–28) match the corresponding bytes of an element in $\Delta^{'}$ (lines 29–32) then

Algorithm IV.2: CANDIDATE-KEYS SKIMMING [23]

Input: $\Delta = \{ \langle\delta_0, 0, \ldots, 0\rangle, \langle 0, \delta_1, 0, \ldots, 0\rangle, \langle 0, \ldots, \delta_u, \ldots, 0\rangle, \ldots, \langle 0, \ldots, \delta_{15}\rangle \}$ with $0 \leq u \leq 15$, $1 \leq \delta_u \leq 255$, and $|\Delta| = 255 \times 16$.
$\Delta^{'} = \{ d \,|\, d \leftarrow \text{MixColumn}(\delta), \forall \delta \in \Delta \}$

Output: L: list of candidate-keys

Data: All the states are represented through a 4×4 matrix, the cells are enumerated from top-left to bottom-right

1 **begin**
2 Record a faulty ciphertext $\widetilde{c}$ and a fault-free one c
3 Record a faulty ciphertext $\widetilde{d}$ and a fault-free one d
4 $L^{'} \leftarrow \emptyset$
5 $c^{'} \leftarrow \langle c_0, c_1, 0, \ldots, 0\rangle$, $\widetilde{c}^{'} \leftarrow \langle \widetilde{c}_0, \widetilde{c}_1, 0, \ldots, 0\rangle$
6 $d^{'} \leftarrow \langle d_0, d_1, 0, \ldots, 0\rangle$, $\widetilde{d}^{'} \leftarrow \langle \widetilde{d}_0, \widetilde{d}_1, 0, \ldots, 0\rangle$
7 **foreach** $(a, b) \in \{0, \ldots, 2^8 - 1\}^2$ **do**
8 $k \leftarrow \langle a, b, 0, \ldots, 0\rangle$
9 $\beta \leftarrow \text{InvSubByte}(c^{'} \oplus k) \oplus \text{InvSubByte}(\widetilde{c}^{'} \oplus k)$
10 $\gamma \leftarrow \text{InvSubByte}(d^{'} \oplus k) \oplus \text{InvSubByte}(\widetilde{d}^{'} \oplus k)$
11 **foreach** $\delta, \delta^{'} \in \Delta^{'}$, $\delta \neq \delta^{'}$ **do**
12 **if** $\delta_0 = \beta_0$ AND $\delta_1 = \beta_1$ AND $\delta^{'}_0 = \gamma_0$ AND $\delta^{'}_1 = \gamma_1$ **then**
13 $L^{'} \leftarrow L^{'} \cup \{ k \}$
14 **break**
15 $L \leftarrow \emptyset$
16 **while** $|L^{'}| \geq 1$ **do**
17 $k \leftarrow \text{GetItem}(L^{'})$ /* $L^{'} \leftarrow L^{'} \setminus \{ k \}$ */
18 **for** $u \leftarrow 1$ **to** 15 **do**
19 $c^{'} \leftarrow \langle 0, \ldots, c_u, c_{u+1}, 0, \ldots\rangle$
20 $\widetilde{c}^{'} \leftarrow \langle 0, \ldots \widetilde{c}_u, \widetilde{c}_{u+1}, 0, \ldots\rangle$
21 $d^{'} \leftarrow \langle 0, \ldots, d_u, d_{u+1}, 0, \ldots\rangle$
22 $\widetilde{d}^{'} \leftarrow \langle 0, \ldots, \widetilde{d}_u, \widetilde{d}_{u+1}, 0, \ldots\rangle$
23 $k^{'} \leftarrow \langle 0, \ldots, k_u, 0, \ldots\rangle$
24 $match \leftarrow$ **false**
25 **foreach** $b \in \{0, \ldots, 2^8 - 1\}$ **do**
26 $k^{'}_{u+1} \leftarrow b$
27 $\beta \leftarrow \text{InvSubByte}(c^{'} \oplus k^{'}) \oplus$ $\oplus \text{InvSubByte}(\widetilde{c}^{'} \oplus k^{'})$
28 $\gamma \leftarrow \text{InvSubByte}(d^{'} \oplus k^{'}) \oplus$ $\oplus \text{InvSubByte}(\widetilde{d}^{'} \oplus k^{'})$
29 **foreach** $\delta, \delta^{'} \in \Delta^{'}$, $\delta \neq \delta^{'}$ **do**
30 **if** $\delta_u = \beta_u$ AND $\delta_{u+1} = \beta_{u+1}$ AND $\delta^{'}_u = \gamma_u$ AND $\delta^{'}_{u+1} = \gamma_{u+1}$ **then**
31 $match \leftarrow$ **true**
32 **break**
33 **if** $match =$ **true then**
34 $k_{u+1} \leftarrow k^{'}_{u+1}$
35 **break**
36 **if** $match =$ **false then**
37 **break** /* Discard k */
38 **if** $match =$ **true then**
39 $L \leftarrow L \cup \{ k \}$
40 **return** L
41 **end**

the value of the 3rd byte of k has been found (lines 33–35). The same operations are repeated for all the remaining bytes, until the whole candidate key k has been checked or the key is candidate discarded (lines 18–37). If a full-length candidate key is computed, it is added to a the list of candidates L, before analysing another item from list $L^{'}$.

After building the candidate list L, the selection of the last round key steps on following lines 8–13 of Algorithm IV.1.

In a real attack scenario the hypothesis to have a fault localised amidst the 8th and the 9th round, will be verified less than one time out of $r = 10$, but also assuming a *correct* fault with a rate of 1 out of 100, experimental evidence demonstrates that the attack is easily mounted against the AES encryption primitive in few minutes using off-the-shelf equipment.

The method proposed in [23] attacks successfully any SPN based cipher with diffusion layer linear with respect to the bitwise xor operation, notwithstanding the fact that the diffusion layer achieves perfect diffusion in a single pass or not.

In the case of the AES cipher the diffusion layer is not perfect and only spreads a single bit difference on a quarter of the inner state (i.e. diffuses a single byte change over a single column (word) of the inner state). The exploitation of this peculiarity of the AES diffusion layer allows to conceive a 32-bit word based implementation of the attack, which retrieves the whole last round key in four passes (one for each word of the last round key).

The key idea of the word based algorithm is rooted in the observation that a single byte fault happening before the last MIXCOLUMN operation will affect only four bytes of the ciphertext. It is thus possible to focus on the recovery of a single word of the last round key at a time, thus reducing the candidate space to 2^{32} at most.

Algorithm IV.3 details the tailored version of Algorithm IV.1 while retaining the same notation. In Algorithm IV.3, Δ now contains all the possible one byte inner state differences for a single word evaluated before the last MIXCOLUMN. Since the differences contained in Δ are computed on a single word, the ciphertexts, both faulty and fault-free, must be carved taking into account both the position of the target word within the round key under retrieval and the effect of the SHIFTROW operation.

Through iterating the Algorithm IV.3 for each of the four word of the **last** round key it is possible to retrieve it regardless of the original key length used.

However the knowledge of the last round key is not enough in order to derive the full cipher key when its length exceeds 128 bits.

D. Generalized AES Attack

The attack described in the previous section is able to recover only the last round key of the Square [14] based ciphers to which is applied, thanks to the peculiar structure of the last round. In the case of AES-128 recovering the

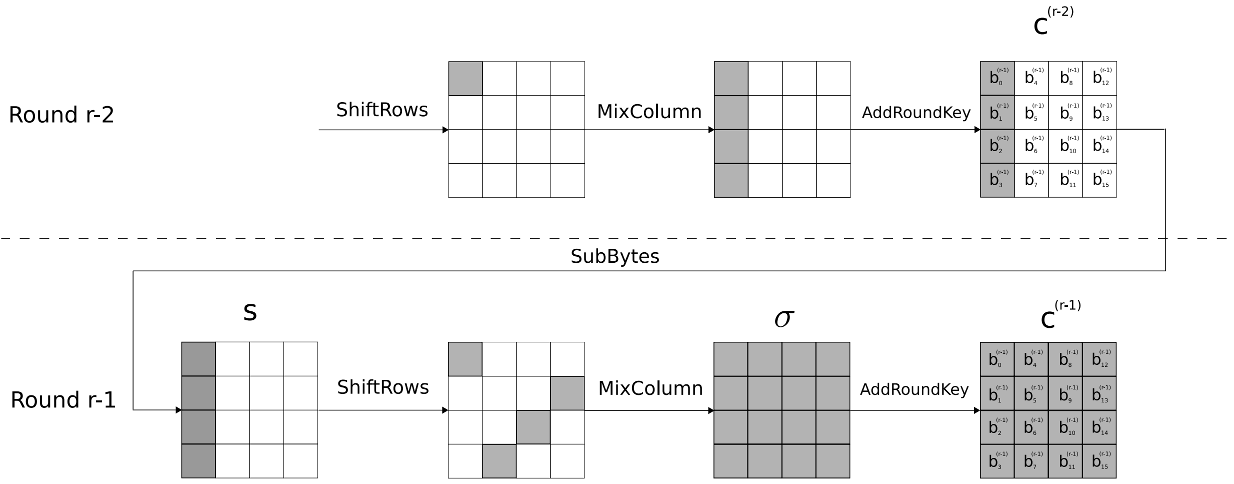

Figure 7. Impact of a single bit fault between the last-but-one and last-but-two MIXCOLUMN operations

Algorithm IV.3: AES WORD ORIENTED KEY RETRIEVAL

Input: $\Delta = \{\ \langle\delta_0, 0, 0, 0\rangle,\ \langle 0, \delta_1, 0, 0\rangle,\ \langle 0, 0, \delta_2, 0\rangle,\ \langle 0, 0, 0, \delta_3\rangle\ \}$,
$\forall\ u \in \{0, 1, 2, 3\}, \delta_u \in \{1, \ldots, 255\}$,
$|\Delta| = 255 \times 4$,
$j \in \{0, 1, 2, 3\}$ round key word index,
$\Delta^{'} = \{\, d \,|\, d \leftarrow \text{MixColumn}(\delta),\ \forall \delta \in \Delta \,\}$

Output: ϖ, j-th word of the last round subkey

1 **begin**
2 Record a fault-free ciphertext, and carve a word w and a faulty ciphertext, and carve a word $\widetilde{w}$ both according to j and taking into account the last ShiftRow operation
`/* Set up of Candidate-Words List */`
3 $L \leftarrow \emptyset$
4 **foreach** $v \in \{0, \ldots, 2^{32} - 1\}$ **do**
5 $\delta^{'} \leftarrow \text{InvSubByte}(w \oplus v) \oplus \text{InvSubByte}(\widetilde{w} \oplus v)$
6 **if** $\delta^{'} \in \Delta^{'}$ **then**
7 $L \leftarrow L \cup \{\, v \,\}$
`/* Word Selection Phase */`
8 **while** $|L| > 1$ **do**
9 Record a fault-free ciphertext, and carve a word w and a faulty ciphertext, and carve a word $\widetilde{w}$ both according to j and taking into account the last ShiftRow operation
10 **foreach** $v \in L$ **do**
11 $\delta^{'} \leftarrow \text{InvSubByte}(w \oplus v) \oplus \text{InvSubByte}(\widetilde{w} \oplus v)$
12 **if** $\delta^{'} \notin \Delta^{'}$ **then**
13 $L \leftarrow L \setminus \{\, v \,\}$
14 **return** ϖ `/*` $L = \{\varpi\}$ `*/`
15 **end**

last round key is enough also to reconstruct the whole key schedule.

For all the others key length employed in the AES this reconstruction cannot be performed only with the last round key due to lack of key material. In fact, the key scheduling strategy of AES uniformly spreads the informative content of the cipher key over the whole key schedule in a word wise fashion (see Section IV-A). It is therefore mandatory to retrieve at least as many words of the key schedule content as the ones composing the cipher key. Moreover the position of the recovered words needs to be such that they do not contain redundant information. In particular, the knowledge of a consecutive block of words (at least as wide as the cipher key) from the key schedule enables a successful cipher key reconstruction.

Since the aforementioned attack is bound to the lack of the MIXCOLUMN operation in the last round of the cipher, it is not able to actually invert the cipher any further. Thus, if either a different key scheduling strategy is employed (e.g. derive a round key from a single word of the original key, cycling though the words, instead of using the standard key schedule procedure), or if the key length is extended up to filling the whole key schedule resulting in an AES employing a $128r$ bit wide key, where r is the number of rounds, the previous attack strategy fails to break the cipher.

We devised a new attack technique which is able to pierce successfully a regular round of the AES cipher (i.e. one including the MIXCOLUMN), thus obtaining a method able to roll back the whole cipher and retrieve all the round keys regardless of their mutual relations, derivation strategy or the number of rounds.

We are therefore able to break the AES cipher even when used with the key lengths recommended for Secret and Top-Secret documents by NSA (192 or 256 bits). No results of a successful key extraction from either AES-192 or AES-256 are known.

Algorithm IV.1 and Algorithm IV.2 work under a known

ciphertext assumption with no particular requirements on the enciphered plaintexts, other than having pairs of faulty and fault free ciphertexts obtained from the same plaintext.

Our extension will require the enciphered plaintext to be the same for all the faulty ciphertexts needed, while retaining the assumption of not knowing the actual plaintext. This is not particularly hindering in practise since the number of required faulty ciphertexts is very small (16 at most).

Algorithm IV.5 is able to invert both the last (r-th) and the last-but-one $(r-1)$-th rounds of the AES cipher, thus retrieving the last two round keys ($k^{(r)}$ and $k^{(r-1)}$). In order to recover the last round key we employ Algorithm IV.3 (line 3). Subsequently, using the retrieved round key we invert the effect of the last round for all the ciphertexts available.

To perform the retrieval of the key $k^{(r-1)}$ we assume that an erroneous ciphertext is the result of a single byte fault occurred between the last-but-one MIXCOLUMN (round $r-2$) and the last-but-two MIXCOLUMN operation (round $r-3$). As depicted in Figure 7, this fault will result in a complete corruption of the state $c^{(r-1)}$ by the end of the last but one round, therefore, in order to distinguish the induced errors respecting our hypothesis from the non useful ones, we need to cope with the diffusing effect of the last MIXCOLUMN operation and to eliminate the obfuscation provided by the $(r-1)$-th ADDROUNDKEY.

In order to remove the effect of the ADDROUNDKEY the GETDIFFERENTIAL function (Function IV.4) at first inverts the last round for a faulty ciphertext $\widetilde{c}^{(r)}$ (line 3) using the last round key $k^{(r)}$ which has already been retrieved, and then computes the difference between the correct and faulty outputs of the last-but-one round. This differential information can be safely transformed through an INVMIXCOLUMN since the diffusion layer is linear w.r.t the xor operation, and subsequently passed through a INVSHIFTROW primitive to realign the bytes (line 4).

We are now able to distinguish a the effects of a useful fault for our purposes through examining the computed differential value (denoted as δ in Function IV.4) and checking whether it is non-zero for only a single word as depicted in Figure 7. In the case the fault is not useful, the function discards the faulty ciphertext and starts examining a new one. Once a useful fault has been found, the function GETDIFFERENTIAL returns both the non zero word differential and its relative position within the state.

Assuming the fault skimming issues are solved as described, the attack described in Algorithm IV.5 can be successfully mounted trying to recover the value of the four words of $c^{(r-2)}$ after the application of the SUBBYTE primitive (denoted by s (line 21) and depicted in Figure 7). We will therefore use four sets of candidates, one for each word of the state matrix to be recovered (s) (line 2).

After obtaining a fault free ciphertext and applying the attack proposed by [23] in order to recover the last round key $k^{(r)}$ (lines 3–4), the effect of the last round is inverted on the correct ciphertext obtaining $c^{(r-1)}$ (line 5).

In lines 7–13 the four candidates lists are filled one at a time until they all contain at least a value. The word differential value ϖ returned by the GETDIFFERENTIAL function is used in order to fill the list indexed by the value m, also returned by the same function.

In order to exploit the information provided by knowing that a single byte fault occurred, we now guess a word w of the s matrix, combine it with the correct differential ϖ and obtain an alleged faulty-correct $(\widetilde{w}, w)$ pair of state s words (line 9). The two words $w, \widetilde{w}$ may be separately processed through an INVSUBBYTE operation since they represent pure state information and an used in order to obtain the differential state value ζ which represents the alleged difference between $c^{(r-2)}$ and $\widetilde{c}^{(r-2)}$ (lines 10–11).

If the guess on the state word was correct, we are now in possess of a state differential ζ which, once processed through an INVMIXCOLUMN operation, will retain only a single non-zero byte in accordance with the verified fault assumption (as depicted in the first state matrix of Figure 7). In this case, the guessed state word w is added to the candidate list under processing L_m (line 12).

Once all the lists are filled with at least a single candidate word, a pruning phase takes place (lines 15–22). This second phase aims at reducing the number of candidates contained in each list to one, through further validation. In order to perform this pruning, a new differential ϖ is obtained from a fresh faulty ciphertext, and all the candidates for that differential word are checked for validity with the same criterion used to include the guesses in the candidate lists (lines 17–20). In the case a candidate word does not pass the check, it is removed from the list (lines 21–22).

After obtaining a single candidate for each four of the state word of s, it is possible to apply a SHIFTROW and a MIXCOLUMN operation to find the correct value of the σ state (see Figure 7). In order to to retrieve the $(r-1)$-th round key $k^{(r-1)}$, it suffices to compute $\sigma \oplus c^{(r-1)}$.

If needed, this procedure may be performed again at will, since it is possible to fully invert the effect of any round of the AES algorithm by removing the rounds one by one.

V. ASYMMETRIC KEY APPLICATIONS - ATTACKS TO RSA

In order to test the efficacy of the new fault model proposed in Section II against a public key cryptosystem, we chose to attack the RSA cryptosystem since, due to its vast adoption, it has undergone an extremely careful cryptanalytic scrutiny and thus represents an appealing target.

In this section we present two attack techniques, one of which is well known and will serve as a testbench for our fault model, while the other one has been designed from scratch.

The first one is the so called Bellcore attack to the RSA signing primitive, when implemented using the Chinese Remainder Theorem. Its aim is to recover the private key while in possess of a faulty signature.

The new one, henceforth named e-th root extraction attack, aims at decrypting an RSA message under a known ciphertext only assumption. The only requirement is to have a faulty and a correct encryption of the same unknown message. The attack

Function 4.4: `GetDifferential`$(c^{(r-1)}, k^{(r)})$

Input : $c^{(r-1)}$, fault-free last-but-one round output; $k^{(r)}$, last round key
Output: (w, j), w: one word difference between faulty and faulty free state after the SubByte of the last-but-one round; $j \in \{0, 1, 2, 3\}$: position of the only non-zero word in the aforementioned difference

```
1  repeat
2      Record a new faulty ciphertext c̃^(r)
3      c̃^(r−1) ← InvSubByte(InvShiftRow(c̃^(r) ⊕ k^(r)))
       /* last-but-one MixColumn                          */
4      δ ← InvShiftRow(InvMixColumn(c̃^(r−1) ⊕ c^(r−1)))
5  until
   δ ∈ { ⟨w0, w1, w2, w3⟩ | ∃ ! j ∈ {0,1,2,3}, w ≠ 0 }
6  return (w_j, j)
```

technique is not specifically tailored for our fault model and fits reasonably well even multi-bit fault events.

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 [8] 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 [20], the modulus factorisation is also computable using only the message m and one faulty computation of the signature $\widetilde{s}$, through calculating

$$p = \gcd(\widetilde{s}^e - m, n)$$

Algorithm IV.5: FULL AES DIFFERENTIAL ATTACK

Input: $\Delta = \{\ \langle \delta_0, 0, 0, 0\rangle,\ \langle 0, \delta_1, 0, 0\rangle,\ \langle 0, 0, \delta_2, 0\rangle,\ \langle 0, 0, 0, \delta_3\rangle\ \}, \forall\ u \in \{0, 1, 2, 3\}, \delta_u \in \{1, \ldots, 255\}$, $|\Delta| = 255 \times 4$, $\Delta' = \{\, d \mid d \leftarrow \textsc{MixColumn}(\delta),\ \forall \delta \in \Delta\,\}$
Output: $(k^{(r-1)}, k^{(r)})$, last two round keys

```
1  begin
2      L0 ← ∅, L1 ← ∅, L2 ← ∅, L3 ← ∅
3      Record a fault-free ciphertext c^(r)
4      Apply Algorithm IV.3 and retrieve the last round key
       k^(r)
5      c^(r−1) ← InvSubByte(InvShiftRow(c^(r) ⊕ k^(r)))
6      repeat
           /* m ∈ {0,1,2,3}, round key word
           index                                           */
7          (ϖ, m) ← GetDifferential(c^(r−1), k^(r))
8          foreach w ∈ {0, ..., 2^32 − 1} do
9              w̃ ← w ⊕ ϖ
10             ζ ← ⟨ζ0, ζ1, ζ2, ζ3⟩ ← ⟨0,0,0,0⟩
11             ζm ← InvSubByte(w̃) ⊕ InvSubByte(w)
               /* last-but-two MixColumn                   */
12             if InvMixColumn(ζ) ∈ Δ' then
13                 Lm ← Lm ∪ { w }
14     until ∀m, Lm ≠ ∅
15     while ∀m, |Lm| > 1 do
16         (ϖ, n) ← GetDifferential(c^(r−1), k^(r))
17         foreach w ∈ Ln do
18             w̃ ← w ⊕ ϖ
19             ζ ← ⟨ζ0, ζ1, ζ2, ζ3⟩ ← ⟨0,0,0,0⟩
20             ζm ← InvSubByte(w̃) ⊕ InvSubByte(w)
               /* last-but-two MixColumn                   */
21             if InvMixColumn(ζ) ∉ Δ' then
22                 Ln ← Ln \ { w }
       /*  L0 = { w̄0 }, L1 = { w̄1 }                        */
       /*  L2 = { w̄2 }, L3 = { w̄3 }                        */
23     s ← ⟨ w̄0, w̄1, w̄2, w̄3 ⟩
       /* last-but-two MixColumn                           */
24     σ ← MixColumn(ShiftRow(s))
25     k^(r−1) ← c^(r−1) ⊕ σ
26     return (k^(r−1), k^(r))
27 end
```

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, which nicely fits our fault model.

B. e-th Root Extraction Attack

In order to attack the RSA cryptosystem we propose a new algorithm to extract the e-th root of a number modulo n in polynomial time exploiting the knowledge of another power of the same number. The target of this attack is to retrieve the

Algorithm V.1: e-TH ROOT EXTRACTION

Input: $e_1, e_2 \in \{1, \ldots, \varphi(n) - 1\}$, $e_1 \geq e_2$,
$c_1 = m^{e_1} \bmod n$, $c_2 = m^{e_2} \bmod n$

Output: (m, n): either $(m, \perp)$ if the e-th root may be extracted, (p, q) if the modulus can be factored or $(\perp, \perp)$ otherwise

```
1  begin
2      if τ ≠ 1 then
3          return (τ, n/τ)
4      τ ← gcd(c_2, n)
5      if τ ≠ 1 then
6          return (τ, n/τ)
7      if gcd(e_1, e_2) ≠ 1 then
8          return (⊥, ⊥)
9      τ ← gcd(c_1, n)
10     γ_1, γ_2 ← c_1, c_2
11     ε_1, ε_2 ← e_1, e_2
       /* Integer division */
12     θ ← ⌊ε_1/ε_2⌋,  ρ ← ε_1 mod ε_2
       /* Cost: 1 modular multiplication, 1 modular inversion and 1 modular exponentiation */
13     γ_3 ← γ_1 γ_2^{-θ} mod n
14     while ρ ≠ 0 do
15         γ_1, γ_2 ← γ_2, γ_3
16         ε_1, ε_2 ← ε_2, ε_1 − θε_2
           /* Integer division */
17         θ ← ⌊ε_1/ε_2⌋,  ρ ← ε_1 mod ε_2
           /* Cost: 1 modular multiplication, 1 modular inversion and 1 modular exponentiation */
18         γ_3 ← γ_1 γ_2^{-θ} mod n
19     return (γ_2, ⊥)
20 end
```

input message encrypted through RSA using a correct and a faulty encryption of the same message.

This hypothesis is analogous to being able to decipher a message assuming the knowledge of two encryptions done with two public keys sharing the same modulus n. Whilst this does not happen due to an incorrect generation of two public-private keypairs (otherwise the two keyholders would be able to mutually read the other's messages), the encryption of a same message through exponentiation by two different public exponents e_1, e_2 may happen if a message is re-encrypted and a fault hits the exponent during the second encryption.

A practical applicative scenario could be the retrieval of the session key during an RSA-KEM [28] handshake. This assumes that the party in charge to choose the session key re-encrypts the same value in the case a faulty encapsulation occurred. To the best of the author's knowledge this technique has not yet been used in order to mount an attack.

Algorithm V.1 describes a method to retrieve the plaintext of an RSA encryption using Euclid's Greatest Common Divisor Algorithm as a pivot to perform operations on the two known ciphertexts.

In the case either of the ciphertexts shares a nontrivial factor with the modulus n, which would in turn imply that the ciphertext value is a zero divider over $(\mathbb{Z}_n, \cdot)$, it is possible to employ it to factor n by simply computing their greatest common divisor. However, the chances of this happening in a real world scenario are extremely slim: in fact the ratio of unitary elements in $(\mathbb{Z}_n, \cdot)$ is exactly $\varphi(n)/n$, which is very close to one when n is the product of two large primes.

The algorithm properly extracts the e-th root only when the two values e_1, e_2 are coprime. A well known result in number theory [17] states that, provided that the two numbers are randomly chosen from a large enough range, the probability of them being coprime approaches $\frac{6}{\pi^2}$, that is roughly 61%.

The algorithm computes the value of $gcd(e_1, e_2)$ following the classic Euclid's Algorithm and computing for each step the value of $m^{e_1 \bmod e_2} \bmod n$ employing the values $c_1 = m^{e_1} \bmod n$ and $c_2 = m^{e_2} \bmod n$ (line 13 and line 18).

Assuming $e_1 \geq e_2$, the number of steps that Euclid's Algorithm needs to perform is in $\mathcal{O}(\log(e_1))$, therefore at most in $\mathcal{O}(\log \varphi(n))$ (*Lamé's Theorem* [19]).

For each step, the integer division between the exponents has complexity in $\mathcal{O}(\log^2 \varphi(n))$, which is dominated by the complexity of the additional modular operations required to compute the intermediate value γ_3 (line 18) using the two ciphertexts. In fact, the complexity of performing a modular multiplication, a modular exponentiation and a modular inversion is in $\mathcal{O}(\log^3 n)$. Thus, the complexity of the whole algorithm is in $\mathcal{O}(\log^4 n)$, that is in P and therefore treatable even for large values of n. In particular, given the common sizes of n in RSA modules the computation is largely feasible even with limited computational resources.

In order to employ the Algorithm V.1 in a fault attack scenario, the values of e_1 and e_2 must be known: this is equivalent to a very precise fault hypothesis where both the number of erroneous bits of the exponent and their positions are known.

We assume, coherently with the error model presented in Section II, the hypothesis of a single faulty bit of the exponent, whose position is known up to a small number of possible ones. Express a single bit faulty exponent e_2 as $e_1 - 2^{\xi}$ for some values of $\xi \in \Xi = \{0, \ldots \lceil \log_2 \varphi(n) \rceil - 1\}$; in order to retrieve the correct plaintext m we need to run the Algorithm V.1 for each possible value of e_2, and check through re-encryption if the computed value is the one sought.

In the worst case, for a single bit fault, the number of hypotheses amount exactly to the bit size of $\varphi(n)$. On the other hand, if the position of the faulty bit is fixed w.r.t. the width w of the computing device word (as in Section II-E), the amounts of the hypothesis set Ξ is reduced to $\frac{\lceil \log_2 \varphi(n) \rceil}{w}$, which typically is between one and two orders of magnitude smaller than $\lceil \log_2 \varphi(n) \rceil$.

VI. Experimental Results

We now provide experimental evidence of the practicality of the algorithmic techniques exposed in Section IV and in Section V, and reporting the results of conducting them on an ARM9 CPU. We report figures of merit for both the attack strategy proposed in [23] and our original contribution which allows us to attack any number of rounds of any AES cipher. Subsequently, we discuss the results of the experimental campaign conducted in order to assess the practical feasibility of the Bellcore and e-th root extraction attacks addressed to the RSA cryptosystem.

The attacked platform was running a vanilla Linux 2.6.15 kernel (DENX distribution) during all the fault collection campaigns and the programs performing encryption were compiled into regular ELF binaries which were run from the shell. Both the instruction and the data caches of the CPU were *enabled* during the experiments and the frequency set to the maximum one supported, thus providing an unsimplified real world working condition.

A. Experimental Evaluation of the Attacks to AES

Since all the attacks on AES are based on the successful injection of one byte faults in a specific word of a specific round of the algorithm, the first step to ascertain the practical feasibility of the attack is understanding the distribution of the faults over the states of the cipher.

We considered three different implementations of AES according to the strategies described in Section IV-A: the first implementation uses 4 T-boxes and is the one used in OpenSSL [13], while the other two respectively use a single T-box and the reference S-box in order to achieve a smaller memory footprint. The choice of evaluating implementations of AES differing by the computation-memory tradeoff was made in light of the fact that the data caching policies of the ARM9 could have a sensible impact on the performances of the attacks, since the CPU caches have been shown in Section II to have a mitigating effect on faults. The first explorative campaign was directed at understanding the fault distribution w.r.t. the rounds of the cipher. Figure 8 depicts the faults spread on the first 10 rounds of the AES-128 algorithm, obtained through collecting 100k faults and classifying them by the round they hit. This was done through inverting the faulty ciphertexts with the known key and calculating the differences between each state of the correct and the erroneous runs until the single byte difference was found. The depicted data were collected using the 4 T-box implementation of AES, but all the other fault distribution differ for less than 0.1% for each value from the reported ones. As the figure shows, the fault are almost equally distributed on the first $r-1$ rounds of the cipher except for the last one which has a sensibly lower probability to be hit. The fault distribution over the rounds for the AES-192 and AES-256 algorithms are analogous to the reported one except for the larger number of rounds.

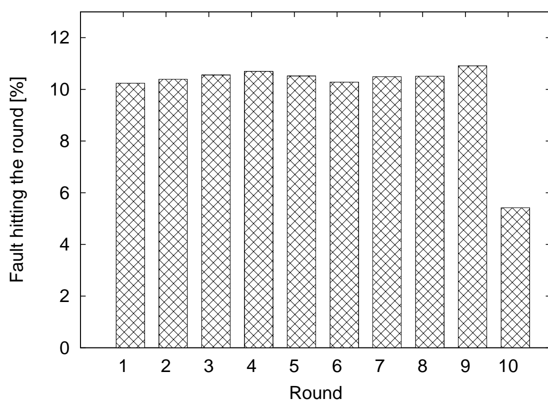

Figure 8. Distribution of the faults over the rounds of the AES algorithm

Table II
Percentages of faults hitting each column over 50k injected faults – AES four T-boxes

State Word	Faults hitting a column [%]			
	O0	O1	O2	O3
First	40.30	25.08	24.48	24.63
Second	19.15	24.73	25.05	24.14
Third	19.36	24.62	25.13	25.93
Fourth	21.17	25.56	25.32	25.28

Table III
Percentages of faults hitting each column over 50k injected faults – AES one T-box

State Word	Faults hitting a column [%]			
	O0	O1	O2	O3
First	25.16	25.00	25.06	25.52
Second	23.85	25.67	25.03	25.12
Third	25.81	23.99	24.35	24.75
Fourth	25.26	25.33	25.54	24.60

Table IV
Percentages of faults hitting each column over 50k injected faults – AES Reference Implementation

State Word	Faults hitting a column [%]			
	O0	O1	O2	O3
First	24.45	24.83	24.38	20.76
Second	22.94	25.12	25.81	19.23
Third	25.10	24.88	26.19	20.59
Fourth	27.48	25.17	23.60	39.40

Willing to ascertain the fault distribution over the state of a single round, it is necessary to take into account the effect of the optimisation strategies employed by the compiler. This is mandated by the fact that aggressive optimisation may employ the coalesced instructions of the ARMv5TE architecture which may alter the fault spread over the words of the state.

Table II, Table III and Table IV report the fault spread over the words of a state, averaged over 50k faults for each implementation, and sorted by increasing optimisation level to which the GCC compiler was set.

The reported results depict an uniform spread of the faults over all the four words of the inner state of the cipher, regardless of the implementation or the optimisation grade of the binary.

The collected results ascertain an uniform spread in the fault locations both in the space and time domain within the execution of the cipher. Thus ensuring the possibility of obtaining useful faults to lead the attacks, provided a statistically significant number of faults is collected.

After collecting a 50k faulty ciphertexts from 2000 different plaintexts, we moved on to process them offline through Algorithm IV.3 [23] and our new Algorithm IV.5.

All the offline computations have been conducted on an Intel Core i7 920 clocked at 4.0GHz and running Ubuntu Linux 9.04 AMD64. All the algorithms have been implemented in `C++` using POSIX standard threads in order to split the load on the four cores of the machine.

Among all the collected faulty ciphertexts properly formed to be exploited in Piret's attack (Algorithm IV.3), some are unsuitable to recover the key, according to theoretical claims of [23] (roughly 2% of them). Table V reports the percentage of practically useful faults to be employed in Piret's Attack: the reported figures confirm the aforementioned theoretical observation, reporting an average usefulness of the faults above 95%. Table VI reports the figures of performance of

Table V
USEFUL FAULTS TO LEAD PIRET'S ATTACK

Compiler Optimization	**Useful Faults per Implementation [%]**		
	Reference	**One T-box**	**Four T-boxes**
O0	97.31	94.11	83.01
O1	97.42	97.52	99.13
O2	98.07	95.75	92.69
O3	97.01	93.48	99.23

the two attacks to all key length of AES, regardless of the implementation parameters, since they do not influence the effectiveness of the offline analysis. To evaluate the performances of Piret's attack we collected a statistically significant number of faulty ciphertexts from the encryption of 2000 different plaintexts, together with the correct ciphertexts.

Piret's algorithm needs 8 faults hitting the last-but-one round (two for each word of the state) of the cipher in order to successfully recover the last round key. The first row of Table VI reports that, on average, 84 collected faulty ciphertexts are enough to obtain the 8 correct ones and to recover the cipher key. Indeed, since one fault over ten is a correct one, the expected number of faults to be gathered amounts to 80. The measured CPU time, employed to run Algorithm IV.3, amounts to one minute with a memory footprint of 480kB including all the data involved, thus well within computability with a common desktop.

To evaluate the performances of the generalized attack, we generated two sets of faulty ciphertexts, one with AES-192 and one with AES-256, employing 2000 different plaintext for each algorithm.

The generalised attack to AES requires 16 faults (two for each word of both the last-but-two and the last-but-one round states) in order to retrieve both the last and the last-but-one round keys and thus retrieving either the AES-192 or the AES-256 cipher key. The second row of Table VI reports that, on average 106 and 252 faulty ciphertexts are respectively enough to obtain the 16 correct ones required to retrieve either the AES-192 or the AES-256 cipher key. The measured CPU time, employed to run Algorithm IV.5, amounts to two minute with a memory footprint of 500kB for the AES-192 and to two minute and 21 seconds with a memory footprint of 605kB for the AES-256.

It is thus possible to successfully break the AES cipher through collecting on average 2016kB of faulty ciphertext corresponding to different plaintexts (used to invert the last round) and 2016kB of faulty ciphertext corresponding to the same plaintext (to invert the next to last one). This is widely feasible since there is a high redundancy in the common data encrypted with AES (protocol or file headers, for instance, are always the same).

B. Experimental Evaluation of the Attacks to RSA

Following the track of Section VI-A, in this section we provide an analogous experimental validation of the attack techniques presented in Section V demonstrating the practical feasibility breaking of the RSA cryptosystem through the injection of faults.

1) Bellcore Attack Evaluation: Table VII reports the results of the fault collection campaign on a C implementation of the RSA signature primitive, using the CRT and Montgomery arithmetic. For each modulus size an compiler optimization, 2k faulty signatures were collected and subsequently used to factor the modulus according to the technique described in Section V-A on an Intel Core 2 Quad E6600 clocked at 2.4GHz. Since the Bellcore attack comes down to the computation of a $\gcd$ as far as the computations go, the running times and memory footprint needed to perform the attack are negligible.

Table VII reports success rates for the attack ranging from 83.6% to 42.0 % depending on the modulus size and the optimization grade of the binary. The increase in the success rate of the attack when moving up from 1024 to 2048 bit sized moduli 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.

Table VII
NUMBER OF FAULTS EXPLOITED TO FACTOR THE MODULUS

Modulus Size [bit]	**Exploitable Faults [%]**		
	O0	**O1**	**O2**
512	83.6	70.0	78.3
1024	63.1	42.0	56.6
2048	65.9	63.6	58.9

In order to evaluate a well known and widespread open

Table VI
ATTACK PERFORMANCES WHILE PROCESSING 50K FAULTS

Algorithm	**Key Size [bit]**	**Running Time**	**Memory Footprint [kB]**	**Number of Correct Faults**	**Average Number of Faults Collected**
Piret's Attack	**128**	$1'$	480	8	84
Generalised AES Attack	**192**	$2'$	500	16	106
	256	$2'\ 21''$	605	16	252

source implementation of RSA, we decided to mount the last voltage underfeeding attack to RSA-CRT using OpenSSL 0.9.1i [13] compiled with relase grade compiler optimizations enabled (-O2). In the attacked implementation both message blinding and signature verification attack countermeasures were disabled. The significant difference between the results in Table VIII and the previous ones lies in the fact that the OpenSSL library uses large amount of function pointers thus distrupting the caching strategies employed by the data cache.

Table VIII
PERCENTAGE OF EXPLOITABLE FAULTS OVER 2000 INJECTED FAULTS

Modulus Size	**Exploitable Faults[%]**
512	93.2
1024	77.4
2048	79.4

2) 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 V-B. The platform used for the experiment was the same employed for the experiment of the previous section, that is a C-code implementation of RSA based on Montgomery Multiplication. 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.

Considering modulus sizes of 512, 1024, and 2048 bits respectively, collected 2000 results from faulty runs of the RSA encryption primitive.

For each faulty ciphertext collected, we needed to iterate the plaintext retrieval algorithm (Algorithm IV.1) for all the H possible fault hypothesis and check through re-exponentiation if the retrieved message was correct. The number of possible fault hypotheses H amounts to $\lceil \log_2 e \rceil / w$ where e is the public exponent and w is the word length of the microprocessor (i.e. 32 bit).

Table IX shows in the first column the percentage of exploitable faulty ciphertexts out of 2k 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 2.4GHz.

In the light of the results shown in Table IX, the e-th root extraction attack is validated as a practical methodology to exploit one bit low voltage induced fault.

The worst case recovery time does not exceed 2 minutes

Table IX
ROOT EXTRACTION SUCCESS RATE OVER 10000 INJECTED FAULTS

Modulus [bit]	**Exploitable Faults [%]**	**Single Check and Retrieval Time [s]**
512	92.2	0.263
1024	20.2	3.9845
2048	34.2	101.112

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.

The result shown in this section prove that it is realistically possible to exploit the two proposed attack techniques against the RSA cryptosystem using a reasonable number of induced faults on a complex and widely used platform such as the Linux operating system running on ARM9 microprocessors.

VII. CONCLUSION

In this paper we presented a characterization of a new fault model: the technique chosen to induce the faults was constantly underfeeding the general purpose CPU involved in the computations.

We employed with success the induced faults in order to lead attacks against industry grade implementations of the RSA and the AES cryptosystems. Moreover we devised two new attack techniques, one for each cryptosystem and have been able to validate their practical effectiveness with a thorough experimental campaign. We were able to successfully break the AES cipher employing only 4kB of faulty ciphertext, to retrieve an RSA encrypted plaintext using at most 5 faulty ciphertexts regardless of the size of the modulus and to factor the RSA modulus employing at most two faulty signatures. After conducting the whole experimental campaign no signs of tampering were left on the attacked device, thus proving that the employed technique is not invasive and does not alter the further functioning of the device. The attack technique is fully realizable with low cost off-the-shelf instruments which is a significant strong asset of the proposed attack technique.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Agilent Technologies. 34420A NanoVolt, Micro-Ohm Meter Datasheet, July 2009.

[2] Agilent Technologies. 6633B 100 Watt System Power Supply Datasheet, July 2009.
[3] Agilent Technologies. E3631A 80W Triple Output Power Supply Datasheet, July 2009.
[4] Frederic Amiel, Christophe Clavier, and Michael Tunstall. Fault analysis of dpa-resistant algorithms. In Breveglieri et al. [11], pages 223–236.
[5] ARM. ARM9 Family of General-Purpose Microprocessors, ARM926EJ-S Technical Reference Manual.
[6] B. Callaghan, B. Pawlowski, P. Staubach. RFC 1813 : NFS Version 3 Protocol Specification, June 1995.
[7] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low Voltage Fault Attacks on the RSA Cryptosystem. *FDTC*, In press, 2009.
[8] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT*, pages 37–51, 1997.
[9] Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors. *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*. IEEE Computer Society, 2007.
[10] Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors. *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*. IEEE Computer Society, 2008.
[11] Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors. *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*. Springer, 2006.
[12] Codesourcery. GNU Toolchain for ARM Processors, July 2009.
[13] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie. The OpenSSL Project, May 2009.
[14] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher square. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.
[15] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
[16] Wolfgang Denk et al. Das U-boot Bootloader, May 2009.
[17] G.H. Hardy. *An Introduction to the Theory of Numbers*. Oxford Science Publications. Oxford Press, fifth edition, 1979.
[18] Karen Sollins. RFC 1350 : The TFTP Protocol, July 1992.
[19] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition) (Art of Computer Programming Volume 2)*. Addison-Wesley Professional, 3 edition, November 1997.
[20] Arjen K. Lenstra. Memo on RSA Signature Generation in the Presence of Faults, September 1996.
[21] National Institute of Standards and Technology (NIST). FIPS-46-3: Data Encryption Standard (DES). http://www.itl.nist.gov/fipspubs/, May 1999.
[22] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard. http://www.itl.nist.gov/fipspubs/, November 2001.
[23] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.
[24] Jörn-Marc Schmidt and Christoph Herbst. A Practical Fault Attack on Square and Multiply. *FDTC*, 0:53–58, 2008.
[25] Jörn-Marc Schmidt and Michael Hutter. Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results. In Johannes Wolkerstorfer Karl C. Posch, editor, *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*, pages 61 – 67. Verlag der Technischen Universität Graz, 2007.
[26] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. Optical fault attacks on aes: A threat in violet. In David Naccache and Elisabeth Oswald, editors, *6th Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2009*. IEEE-CS Press, 2009. in press.
[27] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical Setup Time Violation Attacks on AES. In *EDCC-7 '08: Proceedings of the 2008 Seventh European Dependable Computing Conference*, pages 91–96, Washington, DC, USA, 2008. IEEE Computer Society.
[28] Victor Shoup. A proposal for an ISO-Standard for Public Key Encryption (version 2.1), manuscript, December 2001.
[29] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.
[30] STMicroelectronics. SPEAr Head200, ARM926, 200k Customizable eASIC Gates, Large IP Portfolio SoC, May 2009.