POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Comparison between Differential and Correlation Power Analysis Attacks on Embedded Systems

Author: Maurizio Di Lorenzo

Supervisor: Paolo Ernesto PRINETTO

December, 2021

Acknowledgements

I would like to express all my gratitude to my family, who patiently endured me during my study weekends and provided me with unconditional support.

Special thanks to Gianluca Roascio for his help. He was a guide for me, and he has always been available, even in these difficult times.

Abstract

Today, embedded electronic systems are everywhere, controlling every aspect of everyday life both in professional and in private environments. Most of them manage private information or sensitive data and implement some cryptographic algorithms with the aim of protecting private information from stealing. Even if the algorithms themselves can be considered secure, they can be broken by physical observation of certain properties of the electronic devices, such as the current absorbed or the time taken to execute the algorithms. This is how *Side-Channel Attacks* take place.

These kinds of exploits are very effective ways to gain access to secret information hidden in the embedded systems. It relies on the information channels not intended to be used and, in general, underestimated at the development stage. The general principle has been applied a lot before the advent of embedded system devices: in fact, it can be applied to a mechanical system or even to humans without the need for complex measurement systems, such as a thief than opens a safe using a stethoscope, or simply his ear, listening for some TICs that reveal a right combination digit.

Recently, the spread of the embedded devices hosting private or sensitive information, for example in the Internet of Things (IoT) domain, pushes companies to increase their focus on security, spending time and device hardware resources in secure ciphers based on standard algorithms that enable the device to communicate with the external world.

In this work, the power absorption side channel is investigated, since it does not require costly instrumentation and it is easily accessible. Using side-channel techniques, an attacker can gain insights into working data or execution path to get access to some secret information-related behavior, reducing the needed complexity to discover the secret information (i.e., a secret key of an advanced encryption algorithm). Side channels rarely give direct access to secret information, but most of the time they enormously reduce the number of attempts the hacker has to do to get a secret.

The thesis work analyzes two commonly-used techniques to hack the AES cryptographic algorithm: Differential Power Analysis (DPA) and Correlation Power Analysis (CPA), using a low-cost acquisition system, *ChipWhisperer*. Such a platform is equipped with all the required components to execute experimental tests: synchronous acquisition system, target victim processor, and software libraries.

Some thousands of traces have been acquired from AES encryption execution over the platform to gather enough amount of data to test and compare the two methodologies. Comparison results with respect to the target technology are presented.

Finally, an overview of the possible countermeasures commonly adopted is presented, together with a list of known methods to make them ineffective.

Contents

Chapter 1

Introduction

An *embedded system* can be defined as a computing device that has a dedicated function within a larger system. It is embedded as part of a complete device often including electrical or electronic hardware and, possibly, mechanical parts.

Embedded systems are made of central processing units, memories and I/O devices similar to a personal computer, and sometimes they are personal computers eventually rugged, devoted to run special software to control their main system. Usually, they run the device control software under real-time computing constraints [1]. This definition covers a wide range of devices, used in many different applications, included the so-called Internet of Things (IoT).

The focus of this work is on Side Channel attacks. A not exhaustive list of their direct objectives is described here. The result of these attacks can range from the loss of insignificant personal information, economic losses up to compromising strategic infrastructure as analyzed in [2], perhaps as an indirect result of the attack. Most commonly the goal is of an economic nature.

In the following, a list of the Embedded systems applications and the possible direct causes if their security is violated.

Automotive: nowadays every car has a big amount of electronic control units (ECU from now on), in the range from 50 to 100, each of which is an embedded system. These are in general connected to one or more local networks and sometimes connected to the external world through the ECU devoted to multimedia and navigation system [3].

Recently, the advent of autonomous driving capability increased the number of ECUs both to manage the sensors and the control itself.
Also the use of hybrid motors and pure electric motors had an impact on the complexity and, in most cases, the numbers of ECUs. As shown in Figure 1.1, the car's software complexity is even higher than the aircraft one.

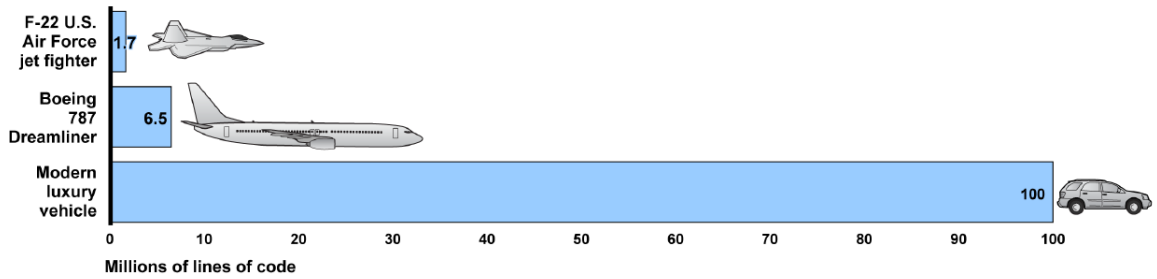

Figure 1.1: Average Lines of Software Code in Modern Luxury Vehicle Compared to Types of Aircraft [4].

Attacks objectives:

- *Change some working parameters*: to increase engine power, modify the emission profiles, etc.
- *Cause vehicle malfunctioning*: can cause injury to people or damage the reputation of the car manufacturer, etc.
- *Get physical access to the vehicle*: to steal the vehicle or the goods present inside of it, etc.

Other transport sectors (aerospace, naval, train...): these types of vehicles are generally equipped with numerous control units devoted to many different functionalities and, in many cases, to achieve the required safety level, mainly due to redundancy.

These are not easily attachable with methodologies that require physical access to the devices, such as Side Channel and Fault Injections, because of poor accessibility. Anyway, there is information about tests put in place by DHS (the US Department of Homeland Security) on remote attacks with positive results (as reported in the article in [5]). This kind of information is kept confidential so no further details are available.

Attacks objectives:

- *To cause vehicle malfunctioning*: can cause injury to people (terrorist attacks).
- *To damage the reputation of the manufacturer*: to cause heavy financial impacts.

Traffic control and infrastructures: traffic control systems cover the supervision of all the possible vehicles typologies, from ground (automotive, trains), sea (open sea and harbor) and air. Also in these cases, the control systems rely on sparse sensors networks and/or on GPS signals prone to be hacked as like the control ECUs placed on board and in control base systems.

There are already different security levels defined for different applications: a secondary road traffic light managing system is less sensitive than a harbor or an airport traffic management system.

Attacks objectives:

- *To cause traffic chaos*: can cause financial losses to both people and/or companies that rely on a defined transfer time
- *To cause people injury or death*: in very sensible cases a malfunctioning can cause severe accidents.

Figure 1.2: Electronic GAS consumption meter.

Consumption meters: consumption meters (or smart meters) are used by the supplying company to charge their customers the cost of consumption measured over a defined period of time.

Today almost all the consumption meters have been converted to smart connected devices able to send out the readings automatically, not involving the customers or external operators to save costs; a sample of present smart meters is shown in Figure 1.2.

Some evaluations of attack-ability have been put in place, one example is reported in [3] where authors demonstrate the effectiveness of cyber attacks.

Attacks objectives:

- *To steel a certain amount of not paid goods*: the main reason to hack such systems is to report a lower (or no) consumption in order to reduce the cost of used energy.
- *To cause financial damage*: the hackers can cause excessive consumption measurements to damage the customer (higher cost) or the supplying company reputation.

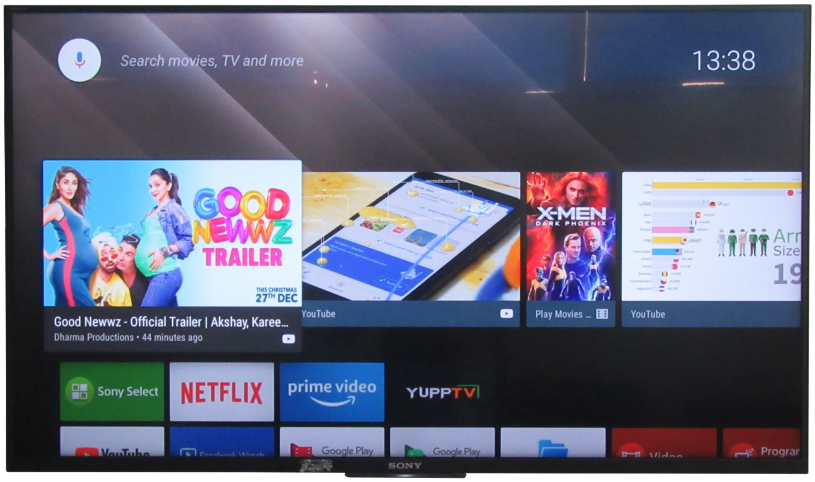

Figure 1.3: Sony Bravia Android TV [6].

Smart entertainment systems: Most home entertainment systems (smart TV as shown in Figure 1.3, Radio, etc) are connected devices with a lot of functionalities and capabilities to browse or play content from the internet, subscribe to the content platform, and manage payment, collect audio and video for communication tools such as Skype, Meet and others.

Attacks objectives:

- *To cause financial damage*: stealing money of subscribed contents.
- *To cause malfunction*: causing the impossibility to get the desired contents.
- *To steal private information*: listening private conversation or recording videos.

House appliances and security systems: Many House Appliances have been developed as connected IoT devices. It has been done mainly to increase the comfort, a cyber butler can manage some annoying stuff for us.

These new features allow us to better manage the house heaters or coolers taking care of switching them on or off based on the presence of people, to use the available electrical power is a smarter way (starting the washing machine after the dishes cleaner machine finished the most power sinking phases)

or in a more economical way, to keep under control the fridge food contents (what is missing and the due-date of some foods), to control the lights and shutters, etc.

Attacks objectives:

- *To create misbehavior*: cause annoying problems due to IoT devices Denial of services (DoS)
- *To cause financial damage*: unwanted power consumption or goods damage
- *To cause issues on power grid*: as depicted in [7] use of cyber attacks on a wide number of high wattage devices to switch them on simultaneously can cause an issue to the power grid.

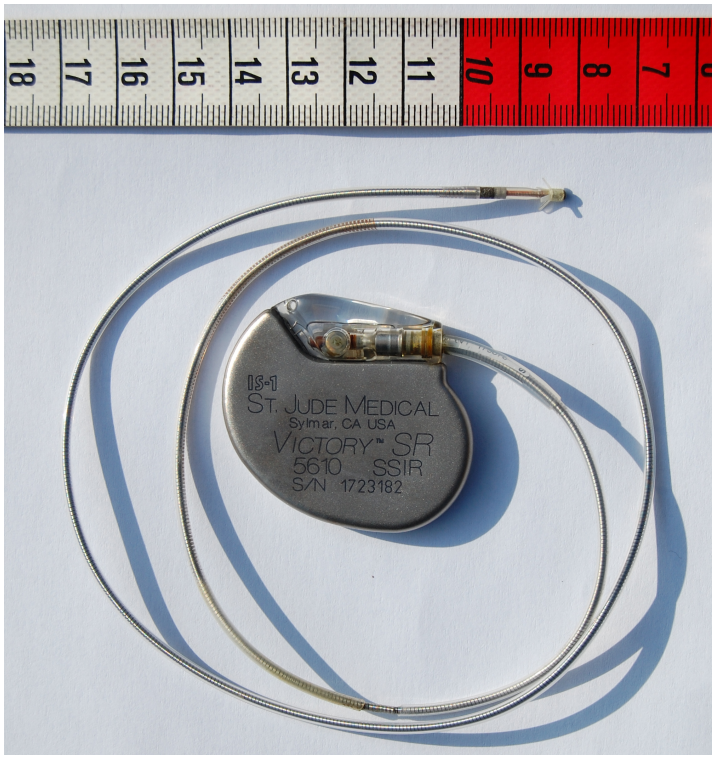

Figure 1.4: PaceMaker [8].

Medical: Almost all the medical devices are based on embedded systems, they can be used to monitor patients (and eventually raise an alert if something goes wrong) or to act on a patient directly such as the insulin auto-feeder or pacemakers (see Figure 1.4). These can be both implanted in the patient body or used as external devices.
Patients' remote assistance, mainly to increase their comfort or to avoid hospitals related infections, is widely used and so the vulnerability of the devices can become a severe problem.

As reported in [2] the companies are already working to mitigate their product vulnerabilities, an example is the ICS Advisory (ICSMA-17-241-01) where Abbott Laboratories claims a Pacemaker vulnerability and recall the patients for a software update.

Attacks objectives:

- *To create misbehavior*: can harm or kill patients
- *To cause financial damage*: to the company that produces or sells the medical devices

PC peripherals: all the PC peripherals are based on embedded systems to manage the communication, configure its critical parameters and perform their function. Most of the times they have been hacked to have privileged access to the PC they are connected to or to cause some annoying problem.

Attacks objectives:

- *To create misbehavior*: cause annoying problems such as devices Denial of services (DoS)
- *To cause product damage*: Product or PC can be damaged
- *To steal personal data*: The PC activity, personal data (passwords, etc.) can be stolen via a virus injected through a peripheral device.

Industrial and agriculture: Industry and Agriculture have similar requirements in terms of sensors (that have to be placed on a large area) and actuators together with the needing for centralized control.

These requirements are satisfied by SCADA systems. Another common requirement of those devices is a long life and the compatibility with old systems in order to keep the plants working simply substituting the damaged devices without the need to redesign the entire network system; for this reason, a wide range of different connectivity is in general available and not always assuring the required security level. Attacks have been done to large production plants gaining access to a SCADA network and sometimes passing through the small IOT sensors or actuators to gather the right to access the network. For further details refer to [2] and [9].

Attacks objectives:

- *To create misbehavior*: cause annoying problems to delay product production
- *To cause product damage*: the produced component (or vegatable) is defective or damaged
- *To cause production system damage*: changing the working parameters some devices can cause damages (rotational speed, robot arm trajectories etc.)
- *To cause operator injuries*: changing the working parameters of some devices can cause damages (rotational speed, robot arm trajectories, etc.) or misbehavior that compromise the operator safety.
- *To steal proprietary data*: automatic working center program contains the locally managed workflow, production rates, etc.

Chapter 2

State of the Art

A good classification of the hardware cyber-attack types by objective and methodology can be found in referenced document [10]. According to it, the attacks methods analyzed in the following can be classified as Passive Non-Invasive and Active Non-Invasive.

These attack methods rely on both software and hardware weaknesses. Software weaknesses are usually caused by the research of performances and can be mitigated by using some best practices available from suppliers operating in security assurance (i.e. RAMBUS or RADWARE) or through related scientific publications. Hardware weaknesses are inherently related to the used technology and internal structure of the processors; in this case, it is harder to mitigate them because any improvement may require a redesign.

2.1 Side-Channel Attacks

Side channel attacks allow us to reveal information not intentionally made available through a non-standard channel; these "channels" are physical observable quantities that are not intended to be used by the system under attack as communication channels.

The use of side channel attacks began well before the electronic era and have been exploited using the capability of electrical and electronic systems to easier access this information.
An example of human side channels are the eye movements, breath or heart frequency, involuntary muscle contractions, etc.; such signals, if observed by an expert eye, can reveal if someone is lying.

Figure 2.1: Polygraph [11].

The "lie detector" (shown in Figure 2.1) helps to measure the human side channels to facilitate and objectify their observation.

Another example is the way a safe with a mechanical combination lock can be opened without knowing its pin. The thief listens to the safe from outside using a stethoscope to hear the "clicks" that reveal the correct combination numbers (see Figure 2.2).
In both cases, the use of side channels allows gaining access to information that should have remained secret.

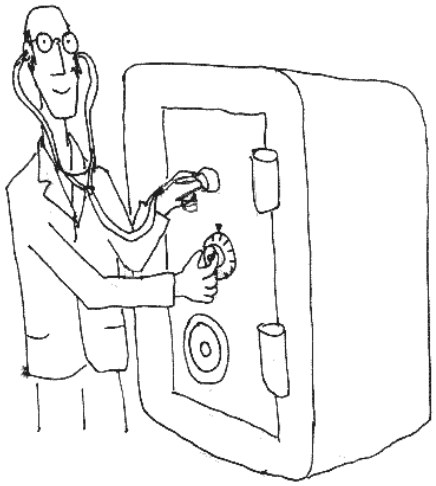

Figure 2.2: Side channel Safe cracking, from [12].

The discovery of side channels and the beginning of their use on electric/electronic devices (mainly ciphers used for military secret communications) can be dated at the end of 2nd World War, even if they have been studied and applied later on, during the Cold War, from both sides of the Iron Curtain [13].

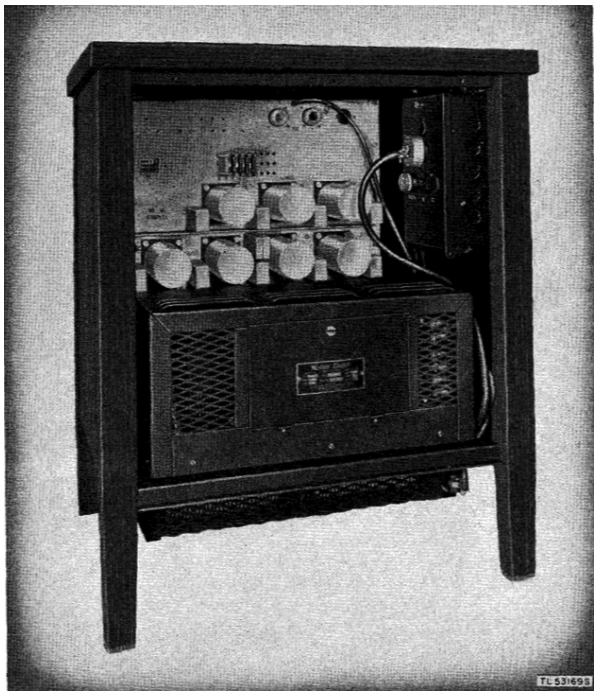

Figure 2.3: TTY mixer 131B2 TM-11-2222, from [14].

At the end of the 2nd World War, the development of cipher algorithms using random key generation and automatic electro-mechanical cipher/decipher devices made almost impossible to decipher the messages without some information about its key. In 1943 some tests at Bell Laboratories on the teletypewriter SET 131-B2, part of Sigtot system, showed the leak of information related to the deciphered (plain text) message via electromagnetic emissions (in Figure 2.3 a component of the Sigtot system).

At that time this discovery just lead to the definition of a security zone around the cipher/decipher installation site to be kept clear from enemies spying antennas to avoid any information leaks.
After this discovery the TEMPEST project was opened: in the first stage, its goal was to define how to avoid information leaks and, later on, how to leverage it to steal the enemies (or even allied) secret communications contents.

To exploit Side Channel attacks, the attacker must gather some knowledge about how the device to be attacked works and gain access to the physical channel he wants to measure.

Nowadays it is easier than in the past to get the device's knowledge:
Form *Hardware* perspective: the device is usually made of commercial off the shelf components (such as the CPU or some devices connected to it) and the necessary information can be freely found over the internet, in case the data is not available some experimental test have to be put in place, also in this case, the relatively low price of the devices to be attacked (for example the electronic keys of smart appliances or the access key to a banking system) helps to get a good cost / possible income ratio to let it attractive.
From *Software* perspective: most of the security related algorithms are defined in standards (sometimes the source code is available in a freely accessible library).

A list of the most common physical quantities used as side channels on embedded systems is given below:

- *Time*: time measurement is the most used. Easy to measure does not require in general any expensive equipment. Can be used alone (in case of attacks made on communication channels only) or with other side channels to observe the time relation between defined events.

- *Sinked Current or Power*: another commonly used side channel. It requires some additional instrumentation in general commonly accessible. Any switching electronic device, such as the CPU registers or data buses, exhibits a different current absorption for a defined state or, in most cases, for a state change; in CMOS devices the bit flip operation causes sunk current peaks somehow proportional to the number of bits flipped.

- *EM radiation*: it requires more complex instrumentation. It is based on the same principle of current or power measurement, when there is fast current change there is a generation of EM. It could be more precise than power measurement because the EM probe can give some information also on the physical position of the EM emissions and allow better signal isolation from other EM sources present in the board or even in another area inside the same DIE.

- *Light emission* in a trivial way some information can be captured through device LEDs light emission observation, some times it can reveal some information (think at the case of a serial communication led directly driven by a line voltage level).
  More complex cases are related to the single transistor observation: when it switches few photons are emitted and observable on a depacked device [15]; it requires the use of fast light detectors and is, in general, more complex and costly than the methods listed before.

- *Sound emission* also in this case there are two approaches: the first one is a passive listening of human-machine interaction to convert the sound emission back into the sequence of operation (i.e. key pressed in the right sequence on a keyboard).
  The more complex one leverage the sound emission due to electronic components vibrations changes, as explained in [16], due to the change of power requirements during various computational phases. Capacitors and coils are the bigger contributors to this phenomenon.

- *Temperature* the passive side channel attack measuring the die temperature requires the device to be depacked and instrumented to measure its temperature. It is similar to the power side channel because the used power has a direct impact on the DIE temperature (see [17]). This approach is more complex and requires, when possible, a deep analysis to be able to reveal the wanted hidden information.

Figure 2.4: Side channel analysis instrument bench [18].

As said before, to exploit most of the side channels attacks, the attacker should have physical access to the target device and some instrumentation as the one shown in Figure 2.4.

Chapter 3

A Real Study-Case: CWNANO

In this Chapter, an experiment to identify the key of an AES algorithm is described.

3.1 ChipWhisperer NANO

To ease the measures acquisition phase a standard off-the-shelf product has been used: the ChipWhisperer NANO[1] (CWNANO). A picture of this tiny board is shown in Figure 3.1. This is the smaller and lower cost platform from ChipWhisperer, anyway its capabilities are enough to gather the data needed to perform a complete Side Channel Power Analysis to get to test different methodologies and get the expected results.

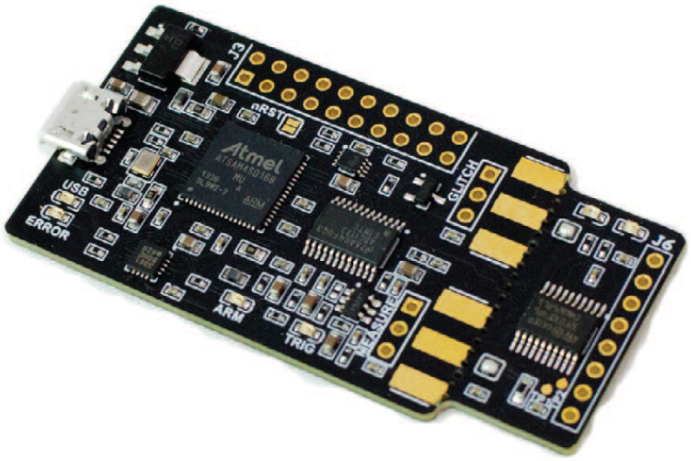

Figure 3.1: ChipWhisperer NANO [19].

In short, the product specification from NewAE CWNANO website [19] and its datasheet are shown in Table 3.1. This board embeds the victim board equipped with an STM32F303F4P6 described in the following Chapter.

The target board clock source is supplied by the CW-NANO acquisition controller, this allows to precisely synchronize the analog sampling point referred to the Target CPU clock edges. The victim absorbed current measurement is made through a shunt resistor placed on the positive power supply pin PCB track (see target description), the signal is then managed by an analogical front end to convert it to a suitable voltage level for the high-speed 8bit parallel analog to digital converter ADC1173, in Figure 3.2 the electrical diagram of the analog input section.

[1] https://rtfm.newae.com/Capture/ChipWhisperer-Nano/

Feature	Notes/Range
ADC Specs	8-bit 20MS/s
ADC Clock Source	Internally generated, external input
Analog Input	AC-Coupled, fixed gain of 10dB
Sample Buffer Size	50 000 samples
ADC Decimation	No
ADC Offset Adjustment	No
ADC Trigger	Rising-edge
Presampling	No
Phase Adjustment	No
Capture Streaming	No
Clock Generation Range	60MHz, divisible by 1, 2, 4, 8, or 16
Clock Output	Regular only

Table 3.1: CWNANO specs.

The circuit performs an AC decoupling and applies an offset to get a zero current level at the mid of the analog to digital converter measurement range (half of Vdd). The complete output function of this circuit is described in Equation 3.1 where I_T is the current flowing into the shunt resistor present on victim board R_{12} and V_u is the signal named AIN in the Figure 3.2 schematics.

$$V_u = \frac{Vdd}{2} - \left((Vdd - R_{12} \cdot I_T) * \frac{s \cdot C_{25} \cdot R_{25}}{s \cdot C_{25} \cdot R_{24} + 1} \right) \tag{3.1}$$

From dynamic perspective, there is a zero at $0\frac{rad}{s}$ and a pole at $370 \cdot 10^3 \frac{rad}{s}$ so around 58 kHz after that point the circuit gain magnitude will be of about $4.5\frac{V}{mA}$.

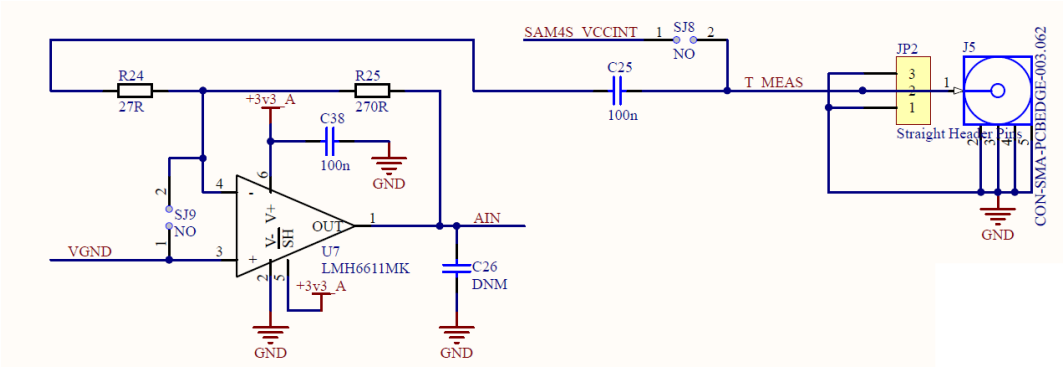

Figure 3.2: ChipWhisperer NANO analog input section, from [19].

The higher is the absorbed current the higher is output measured voltage, so the converted value, so we will have a direct relationship between the microprocessor absorbed current and the sampled value. ChipWhisperer NANO is not only an instrument to gather target side channels info, it includes the capability to inject faults on the power line and to program and control the target victim (at least if it is the one supplied with the board).

The software API is freely available on ChipWhisperer website. The supplied installation package uses a Jupyter Notebook environment to apply the try and test methodology, anyway the ChipWhisperer library can be used in a standard Python program to put in place custom automated procedures.

3.2 Target STM32F0

The victim board supplied with this board is equipped with an STM32F030F4P6. The processor is manufactured by ST and it is based on a 32bit CORTEX M0 IP from ARM Ltd, a RISC processor with a three stage pipeline, with a lot of on-chip peripherals to manage clock, memory, communications, special functions management with various I/O configurations selectable for each pin.
The victim board uses in a minimal I/O configuration, with some connection for both communication to NANO board and control a couple of LEDs; its electrical diagram is in Figure 3.3.

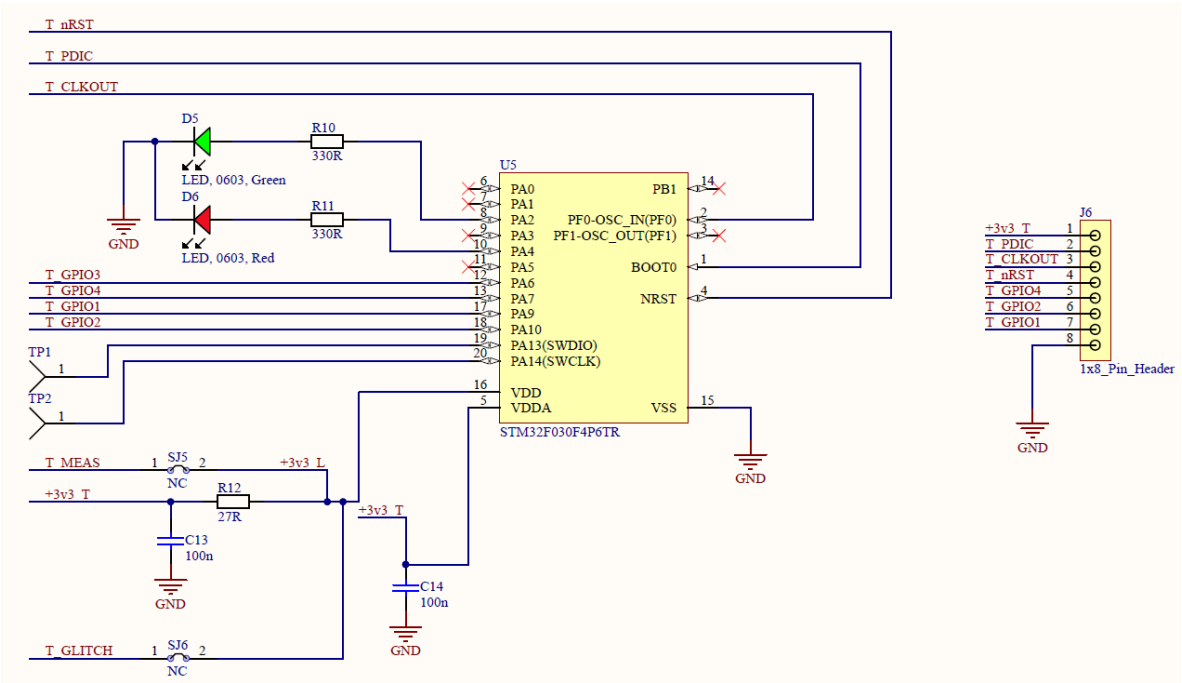

Figure 3.3: ChipWhisperer NANO STM32F030 Target, from [19].

The clock source is supplied by the NANO board (the target firmware shall be configured to use the external clock instead of the internal one to assure the synchronization of the sampling system) and the absorbed current is sensed via a shunt resistor (R12 in Figure 3.3 schematic) placed between the bypass capacitor and the IC input power pin, in this way the effect of the capacitor will not impact the current measures. There is also the possibility to apply some fault injection via the T_GLITCH signal on VDD.

3.3 AES Algorithm Description

All the information reported has been extracted from [20] and [21], see these referenced documents for a deeper description.

AES is a symmetric encryption algorithm that processes data blocks of 128 bits at a time, this length is normally referred to as 4 words of 32 bits and identified as **Nb = 4**. There are 3 different key lengths defined in the standard: 128, 192 or 256 bit, also in this case the key length is expressed as the number of words **Nk = 4, 6, or 8**.

The processing flow, for both the cipher and decipher phase, works around a state of 128 bits organized in 4 x 4 bytes matrix or 4 words columns: **Nb = 4**. The initial values of the state are the starting data to be ciphered or deciphered and modified through different rounds to get the final ciphered text or plain text. The number of rounds to be executed depend on the length of the key: 10 rounds for 128 bits, 12 for 192 bits, and 14 for 256 bits, it is identified with **Nr = 10, 12 or 14**.

The pseudo-code in Figure 3.4 shows the sequence of operations for the Cipher algorithm.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 3.4: AES Cipher pseudo-code [21].

The inverse cipher has a similar approach (almost reverse respect to the one used in the cipher), its description is not useful for the purpose of this work so it will not be explicitly described, refer to [21] for a detailed explanation.

After the state loading, an *AddRoundKey()* is executed then the rounds are repeated until the end. All the rounds of cipher algorithm following the first *AddRoundKey()* are made of the same functions execution sequence (each one described in the following) except the last one where the *MixColoumns()* is missing.

The following transformations are made using Galois Field $\mathbf{GF(2^8)}$ operations, see [20] and [21] for detailed description, for simplicity in this work they will be treated as logic functions and look-up tables as per their code implementation.

3.3.1 SubBytes

SubBytes() is a non linear transformation operated at byte level (see Figure 3.5) using a substitution table named **S-box** showed in Figure 3.6.

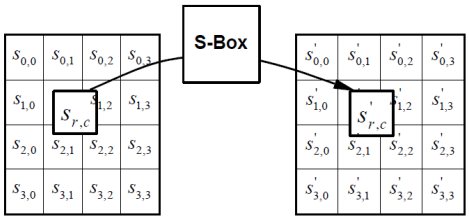

Figure 3.5: S-box application on each byte of the State from [21].

The table is invertible (so each input value gives a unique output value that is not repeated).

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.6: S-box values from [21].

3.3.2 ShiftRows

ShiftRows() transformation operate a cyclical shift (or rotation) in left direction of each state row, made of 4 bytes, of a defined number of bytes, respectively 0, 1, 2, 3.

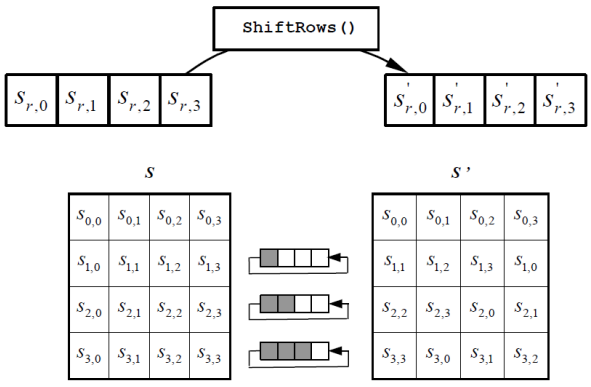

Figure 3.7: ShiftRows from [21].

As depicted in Figure 3.7 while moving the row content towards left, the byte that drops out of the row is placed at the rightmost position (as like a rotation on a circle), this behavior is repeated at each shift.

If we have to shift three times the last row the result of the transformation will be the one in Equation 3.2.

$$S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3} \rightarrow S_{3,3}, S_{3,0}, S_{3,1}, S_{3,2} \tag{3.2}$$

3.3.3 MixColumns

MixColumns() transformation operates on the state column by column, applying a polynomial transformation. Schematically it could be showed as Figure 3.8.

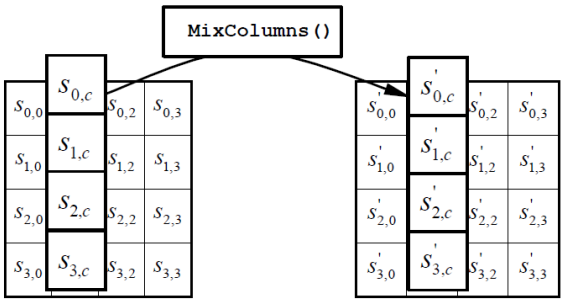

Figure 3.8: MixColumns from [21].

This transformation applies the formulas in Equation 3.3, note that the dot multiplication is a finite field multiplication (see [21]).

$$\begin{aligned}
S'_{0,c} &= (\{02\} \bullet S_{0,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\
S'_{1,c} &= S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c} \\
S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c}) \\
S'_{3,c} &= (\{03\} \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c})
\end{aligned} \tag{3.3}$$

The finite field multiplications are the only operations that can't be easily mapped to any C code operation. These multiplications anyway are only by 02 or 03, note that the polynomial form of this numbers are: $\{02\} = x$ and $\{03\} = x + 1$.

A special function to easily obtain the $\mathbf{GF(2^8)}$ multiplication time x can be used: $x \cdot 2 = x << 1$ conditionally EXORed with $\{1b\}$ if a carry over the byte size occurs; this function is called $xtime()$. So the multiplications in Equation 3.3 can substituted with Equation 3.4 where $xtime()$ is easily implementable in C.

$$(\{02\} \bullet A) \oplus (\{03\} \bullet B) = xtime(A) \oplus xtime(B) \oplus B = xtime(A \oplus B) \oplus B \tag{3.4}$$

The new transformation became the one in Equation 3.5

$$\begin{aligned}
S'_{0,c} &= xtime(S_{0,c} \oplus S_{1,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus S_{3,c} \\
S'_{1,c} &= xtime(S_{1,c} \oplus S_{2,c}) \oplus S_{2,c} \oplus S_{3,c} \oplus S_{0,c} \\
S'_{2,c} &= xtime(S_{2,c} \oplus S_{3,c}) \oplus S_{3,c} \oplus S_{0,c} \oplus S_{1,c} \\
S'_{3,c} &= xtime(S_{3,c} \oplus S_{0,c}) \oplus S_{0,c} \oplus S_{1,c} \oplus S_{2,c}
\end{aligned} \tag{3.5}$$

3.3.4 AddRoundKey

The *AddRoundKey()* transformation operates at word level, applying a binary XOR between each state word and key schedule word as visible in Figure 3.9.

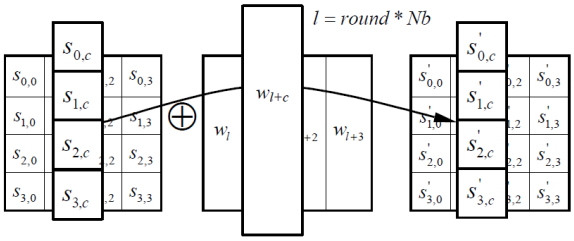

Figure 3.9: AddRoundKey from [21].

This operation can be done at byte level with the same results and it is sometimes more suitable if implemented together with the SubBytes to avoid to save its results into the state.

In Equation 3.6 the *AddRoundKey()* function description; it is repeated for all the state columns referred as c with values that vary from 0 to 3. The key is expanded and rn represent the round number to select the relevant key section.

$$\begin{aligned} S'_{0,c} &= S'_{0,c} \oplus K_{0,c,rn} \\ S'_{1,c} &= S'_{1,c} \oplus K_{0,c,rn} \\ S'_{2,c} &= S'_{2,c} \oplus K_{0,c,rn} \\ S'_{3,c} &= S'_{3,c} \oplus K_{0,c,rn} \end{aligned} \tag{3.6}$$

The expanded key is made of a transformation of the known key in order to have 128 key bits available for each round. The expanded key computation, known as key schedule, is the result of *KeyExpansion()* function.

3.3.5 Key Expansion

The *Key Expansion* algorithm gives the number key words needed to complete the required rounds **Nr** defined by the length of the key **Nk**. The key expansion algorithm pseudo-code is shown in Figure 3.10.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Figure 3.10: Key Expansion algorithm pesudo-code from [21].

The final key schedule is made, for the first **Nk** words, of the registered secret key, the remaining part is computed in a cycle using the following functions:

- *SubWord()* applies the **S-Box** to replace the single bytes content of a four bytes group.
- *RotWord()* rotates left a group of four bytes: $a_0, a_1, a_2, a_3 \rightarrow a_3, a_0, a_1, a_2$.
- *Rcon[i]* is a group of four bytes a_0, a_1, a_2, a_3 where $a_1 = a_2 = a_3 = 0$ and a_0 has a value depending by i defined as the polynomial x^{i-1} evaluated in $\mathbf{GF(2^8)}$. Practically its values are selected from the following bytes value (written in hexadecimal notation): $[00, 01, 02, 04, 08, 10, 20, 40, 80, 1b, 36]$.

3.4 CW-Nano Relevant API Description

The ChipWhisperer APIs documentation can be found in [22], here a short description of the used classes and related parameters usefully to understand the way they have been used for this work, see the online documentation for configuration details.
To use its objects and functions the library *chipwhisperer* shall be imported, usually a shorter name like *cw* is assigned:

```
import cw as chipwhisperer
```

3.4.1 Object Scope

The object ***scope*** create and configure the data acquisition system on ChipWhisperer.

To configure and enable it instantiate this object as a standard class *scope()*. It can accept 2 optional parameters: *type, sn* both with default value set to *None*. These parameters, if not specified, are automatically detected through the ChipWhisperer hardware connected to the PC. In case more than one ChipWhisperers are connected to the PC the desired board can be addressed wuth either the type (if they are of different types) or the serial number that allows selecting the physical board to be addressed. The *scope* instantiated object has different sub-modules, methods and attributes based on the detected hardware, for CWNANO we have the following sub-modules:

- *scope.**adc***: give access to ADC configuration.
  - *scope.adc.**samples***: read/set the number of samples to be stored.
  - *scope.adc.**clk_src***: read/set the ADC clock source: in can be *'int'* or *'ext'*.
  - *scope.adc.**clk_freq*** : read/set the ADC sample frequency, the value is rounded to the closest possible integer value.
- *scope.**io***: acquisition module *GPIO* settings to manage the target communication, program and measure triggering. These are in general left as per default configuration except for target program procedure where the CW NANO board requires a target reset cycle. The available sub-modules are:
  - *scope.io.**tio1***: *T_GPIO1* connected to STM32F0 pin *PA9*,
  - *scope.io.**tio2***: on schematic *T_GPIO2* connected to STM32F0 pin *PA10*,
  - *scope.io.**tio3***: on schematic *T_GPIO3* connected to STM32F0 pin *PA6*,
  - *scope.io.**tio4***: on schematic *T_GPIO4* connected to STM32F0 pin *PA7*,
  - *scope.io.**pdid***: on schematic connected to the 20 pins connector to be used with external target *T_PIDD pin 20*, not connected to STM32F0,
  - *scope.io.**pdic***: on schematic *T_PIDC* connected to STM32F0 pin *BOOT0*,
  - *scope.io.**nrst***: on schematic *T_nRST* connected to STM32F0 pin *NRST*,
  - *scope.io.**clokout***: on schematic *T_CLKOUT* connected to STM32F0 pin *PF0-OSC_IN*,
  - *scope.io.**cdc_settings***: to set the way the USART parameters can be changed with the USB CDC,
- *scope.**glitch***: used to configure the target power line glitches in term of duration and time offset from trigger.
  - *scope.glitch.**ext_offset***: offset form trigger rising edge,
  - *scope.glitch.**repeat***: width of glitch in cycles.
- *scope.**default_setup()*** configure the scope object default values, for CW-Nano they are:

```
ChipWhisperer Nano Device
fw_version =
    major = 0
    minor = 30
    debug = 0
io =
    tio1          = None
    tio2          = None
    tio3          = None
    tio4          = None
    pdid          = True
    pdic          = False
    nrst          = True
    clkout        = 7500000.0
    cdc_settings = array('B', [1, 1])
adc =
    clk_src   = int
    clk_freq = 7500000.0
    samples   = 5000
glitch =
    repeat     = 0
    ext_offset = 0
```

- *scope.**con()*** connects to attached CW Nano, an optional parameter with the CW-Nano serial number can be used.
- *scope.**dis()*** disconnects the scope object from CW-Nano.
- *scope.**arm()*** arm the ADC, the trigger will be GPIO4 rising edge (fixed trigger)
- *scope.**capture()*** captures a new trace.
- *scope.**get_last_trace()*** rerturns an array with last captured trace samples.
- *scope.**get_serial_ports()*** get the CDC serial ports associated with this scope.

3.4.2 Object Target

The object ***target*** gives the interface to the target device (for CW-Nano the on-board STM32F0). The dafault target interface is the *Simple Serial Target*. It requires only one parameter: the scope object instance described above, and allows to configure the target type with optional parameters, some of them are allowed if required by the target firmware. A basic usage example is:

```
import chipwhisperer as cw
scope = cw.scope()
target = cw.target(scope)
```

- *target.**baud***: manage the serial communication baud rate,
- *target.**write(data)***: write *data* passed as parameter of *string* to the target.
- *target.**read(num_char=0, timeout=250)***: read data from target. The two optional parameters define how to manage the communication:
  num_char is the number of characters to be read, the default value 0 set the read of all the available characters;
  timeout is the time to wait before returning if no char is available, the default value is 250ms.
- *target.**in_waiting()***: returns the number of characters available to be read in the serial buffer.

- *target.**in_waiting_tx()***: returns the number of characters waiting to be sent in the ChipWhisperer serial output buffer.
- *target.**simpleserial_wait_ack(timeout=500)***: wait for an ack from target for *timeout* ms, if the optional parameter is not present it waits for 500ms.
- *target.**simpleserial_write(cmd, num, end = '\n')***: write a SimpleSerial command *cmd* (usually a string of 1 char) with data *num* (in byte-array format), *end* is an optional parameter that define the end of string value with default value *'\n'* (Ascii newline character).

```
target.simpleserial_write('p', text)
```

- *target.**simpleserial_read(cmd, pay_len, end='\n', timeout=250, ack=True)***: reads data from target related to command *cmd* (usually a string of 1 char), *pay_len* is the amount of bytes to be received, the other parameters are optional to define the end of string, the recevive maximum timeout and the needing of an *ack* at the end of the command.

```
response = target.simpleserial_read('r', 16)
```

- *target.**simpleserial_read_witherrors(cmd, pay_len, end='\n', timeout=250, glitch_timeout=8000, ack=True)***: to be used in case of glitch test when the expected results can include errors in data contents of format. See ChipWhisperer documentation online for further details.
- *target.**set_key(key, ack=True, timeout=250)***: same function of

```
target.simpleserial_write('k',key)
```

- *target.**close()***: close target.
- *target.**con(scope=None, **kwargs)***: connects to target.

3.4.3 Useful Functions

Chipwhisperer API have also some useful functions to be used to aggregate commands sequence to record traces, function to program the embedded target device and objects to perform different kinds of analysis included into *chipwhisperer.analyzer* library (not used in this work).

In this work the ***program_target(scope, prog_type, fw_path, **kwargs)*** function is used, it allows to program the target without the needingd to use external tools.
Parameters description:

- ***scope***: the scope class instance to be used for the target connection.
- ***prog_type***: the type of target we want to program, the supported ones are:
  - *programmers.STM32FProgrammer*
  - *programmers.XMEGAProgrammer*
  - *programmers.AVRProgrammer*
- ***fw_path***: path to hex file to program including file name and extension.

As example the command used to program the device with the target code described later is:

```
cw.program_target(scope, cw.programmers.STM32FProgrammer, "./
    simpleserial-aes-CWNANO.hex")
```

3.5 Target Code Description

To test the effectiveness of the two Side Channel Power Attacks object of this work, the AES test software available in CWNANO firmware package has been used. It implements a basic AES algorithm without the use of the encryption HW accelerator.

As visible in below description the source code uses a lot of nested functions to achieve the desired behavior; this software structure, even is not easily readable, has been defined by NewAE people to obtain an easy customization of the algorithm and/or the hardware platform keeping the common part of source code used in all the managed platforms.

The selection of target Hardware, algorithms end special configurations is done through some variables configuration at compile time (usually done in a Jupyter environment using PYTHON variables passed though a structured makefile).

Only a few commands are relevant for this work: ***'k'***, ***'p'*** and ***'r'***.

- Command ***'k'*** is used in the *target.set_key()* function and sets the secret key to be used by the AES algorithm.

  The command executes the *AES128_ECB_indp_setkey()* function where the *Key* pointer is initialized with the address of the received key data buffer then executes the key expansion function.

```
static uint8_t* Key;

...

void AES128_ECB_indp_setkey(uint8_t* key)
{
    Key = key;
    KeyExpansion();
}
```

  KeyExpansion() is the routine devoted to create the complete key array *RoundKey[]* to be used in though the different rounds of AES algorithm.
  Only the first part of the function is relevant for this work, to verify that the first 16 *RoundKey[]* bytes are filled with the received key values, the rest of the function computes the rest of the key expansion used in later cipher rounds.

```
static uint8_t RoundKey[176];

...

// This function produces Nb(Nr+1) round keys. The round keys
    are used in each round to decrypt the states.
static void KeyExpansion(void)
{
   uint32_t i, j, k;
   uint8_t tempa[4]; // Used for the column/row operations

   // The first round key is the key itself.
   for(i = 0; i < Nk; ++i)
   {
        RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
        RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
        RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
        RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
   }
```

```
// All other round keys are found from the previous round
   keys.
for(; (i < (Nb * (Nr + 1))); ++i)
{
    for(j = 0; j < 4; ++j)
    {
        tempa[j]=RoundKey[(i-1) * 4 + j];
    }
    if (i % Nk == 0)
    {
        // This function rotates the 4 bytes in a word to
           the left once.
        // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

        // Function RotWord()
        {
            k = tempa[0];
            tempa[0] = tempa[1];
            tempa[1] = tempa[2];
            tempa[2] = tempa[3];
            tempa[3] = k;
        }

        // SubWord() is a function that takes a four-byte
           input word and
        // applies the S-box to each of the four bytes to
           produce an output word.

        // Function Subword()
        {
            tempa[0] = getSBoxValue(tempa[0]);
            tempa[1] = getSBoxValue(tempa[1]);
            tempa[2] = getSBoxValue(tempa[2]);
            tempa[3] = getSBoxValue(tempa[3]);
        }

        tempa[0] =  tempa[0] ^ Rcon[i/Nk];
    }
    else if (Nk > 6 && i % Nk == 4)
    {
        // Function Subword()
        {
            tempa[0] = getSBoxValue(tempa[0]);
            tempa[1] = getSBoxValue(tempa[1]);
            tempa[2] = getSBoxValue(tempa[2]);
            tempa[3] = getSBoxValue(tempa[3]);
        }
    }
    RoundKey[i * 4 + 0] = RoundKey[(i - Nk) * 4 + 0] ^
       tempa[0];
    RoundKey[i * 4 + 1] = RoundKey[(i - Nk) * 4 + 1] ^
       tempa[1];
    RoundKey[i * 4 + 2] = RoundKey[(i - Nk) * 4 + 2] ^
       tempa[2];
    RoundKey[i * 4 + 3] = RoundKey[(i - Nk) * 4 + 3] ^
```

```
                tempa[3];
        }
    }
```

- Command *'**p**'* supplies 16 bytes (128 bits) of plain text data and encrypt it.

  Before to start with the encryption it rise the trigger signal (**GPIO4**) to synchronize the start of data acquisition, this is useful because CVNANO may contain a relatively small amount of data before being transferred to the host computer so is important that we can acquire only the relevant part of the current absorption trace.

```
uint8_t get_pt(uint8_t* pt, uint8_t len)
{
    trigger_high();
    aes_indep_enc(pt); /* encrypting the data block */
    trigger_low();
    simpleserial_put('r', 16, pt);
    return 0x00;
}
```

  The function *aes_indep_enc(pt)* is the actual cipher and it is wrapped to *AES128_ECB_indp_crypto(uint8_t* input)*

```
void AES128_ECB_indp_crypto(uint8_t* input)
{
    state = (state_t*)input;
    BlockCopy(input_save, input);
    Cipher();
}
```

  The plain-text is copied byte by byte (with *BlockCopy()* function) into the *state* variable to be modified in the following AES cipher algorithm *Cipher()*:

```
// Cipher is the main function that encrypts the PlainText.
static void Cipher(void)
{
    uint8_t round = 0;

    // Add the First round key to the state before starting
       the rounds.
    AddRoundKey(0);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr-1 rounds are executed in the loop below.

    for(round = 1; round < Nr; ++round)
    {
        SubBytes();
        ShiftRows();
        MixColumns();
        AddRoundKey(round);
    }

    // The last round is given below.
    // The MixColumns function is not here in the last round.
```

```
        SubBytes();
        ShiftRows();
        AddRoundKey(Nr);
    }
```

- Command *'r'* send back the ciphered text on host request.

3.6 Data Acquisition and Analysis

3.6.1 Leak Model

A simple model of the leak exploited in power analysis from [23] consider a single data line driven by two MOSFETs (one on the UP side and one on LOW side) and a load capacitor from the data bus to the ground as depicted in the Figure 3.11.

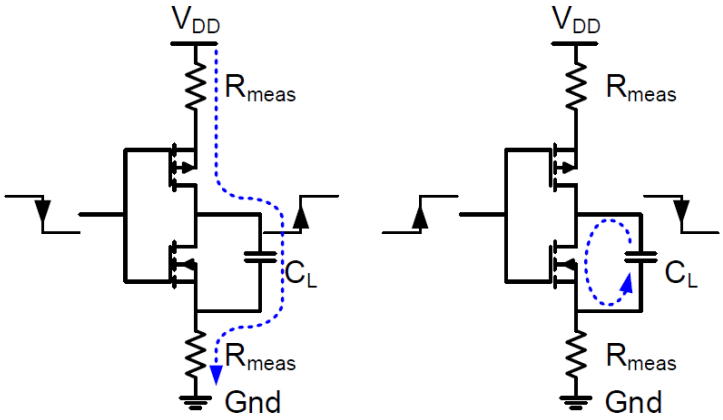

Figure 3.11: Charge and discharge of a CMOS inverter from [23].

An output LOW to HIGH transition causes a temporary current flowing from Vdd through the shunt resistor to charge C_L, the HIGH to LOW transition discharge C_L through the LOW side MOSFET. As stated above the absorbed current of a processor changes as a function of the executed instructions. In the case of CMOS devices big current variations happen when a parallel bus changes the number of active (of HIGH logic status) bits as it happens during a write memory.

The number of non-zero bits of a byte or a word is known as ***Hamming Weight*** (HW from now on) and, according to the leak model, it should be directly related to the absorbed current. The ***Hamming Distance*** is the count of different bits between two states, this is often used to predict the amount of current needed during a transition between two states since only the bits that flips produce some differences in current abortion. The model selection is usually done analyzing the hardware characteristics or, in most cases, both are tested tand the best preforming is selected.

To verify the HW model several traces, about one thousand, have been acquired supplying random plain text then grouped per HW predicting both the AddRoundKey(0) and the SubBytes() results and averaged. The sample point has been identified (in both cases) subtracting the relevant group HW 0 to the average of all traces searching an absolute max value in the sample range defined for the relevant function.

This allows to depict the below Figures 3.12 and 3.13: as expected the current vs HW has almost a linear relationship and increases when HW increases. In AddRoundKey(0) case (Figure 3.12) the flat top is caused by the input ADC saturation that happens each time the recorded track value reach 0,5 (positive saturation) or -0,5 (negative saturation).

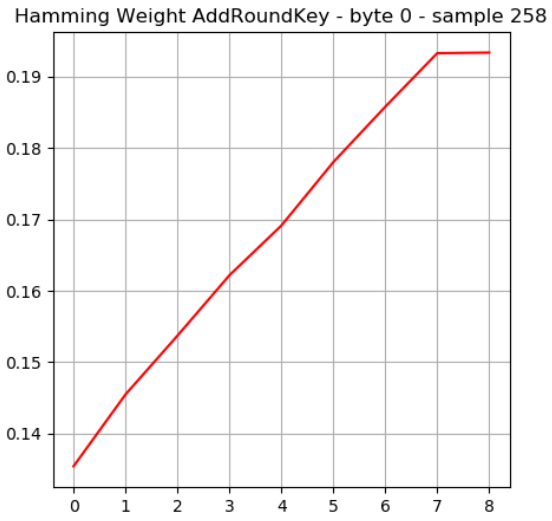

Figure 3.12: Hamming Weight measured at AddRoundKey(0) on byte 0.

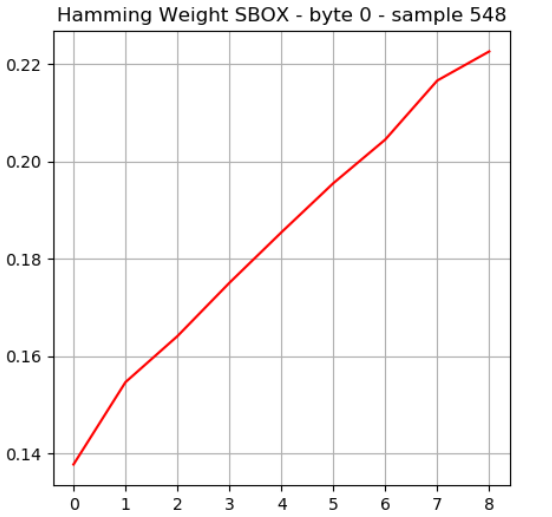

Figure 3.13: Hamming Weight measured at SBOX on byte 0.

3.6.2 Code Execution Timings

The code execution timings have been mapped using an instrumented code, since the used target executes only the AES algorithm no time variations are expected for traces acquired at different times.

In Table 3.2 the list of the execution duration time of the above described AES functions mapped to acquired amount of samples, this has been used to identify the Functions sample starting point.

Event	Duration [us]	Duration [smp]	Absolute [smp]
Trigger	0	0	0
BlockCopy()	28.0	210	0 - 210
AddRoundKey(0)	40.0	300	210 - 510
SubBytes()	33.6	252	510 - 762
ShiftRows()	12.0	90	762 - 852
MixColoums()	67.6	508	852 - 1360
AddRoundKey(1)	40.0	300	1360 - 1660
...	...	...	...

Table 3.2: AES algorithm timings (smp stand for samples).

The data present in Table 3.2 is used to tag a sampled current trace shown in Figure 3.14.
A first visual analysis of Figure 3.14 shows that each function's code execution absorbed current shape becomes clearly identifiable and, zooming on the relevant part of the trace, also the iterations within the single function became easily identifiable (see Figure 3.15).

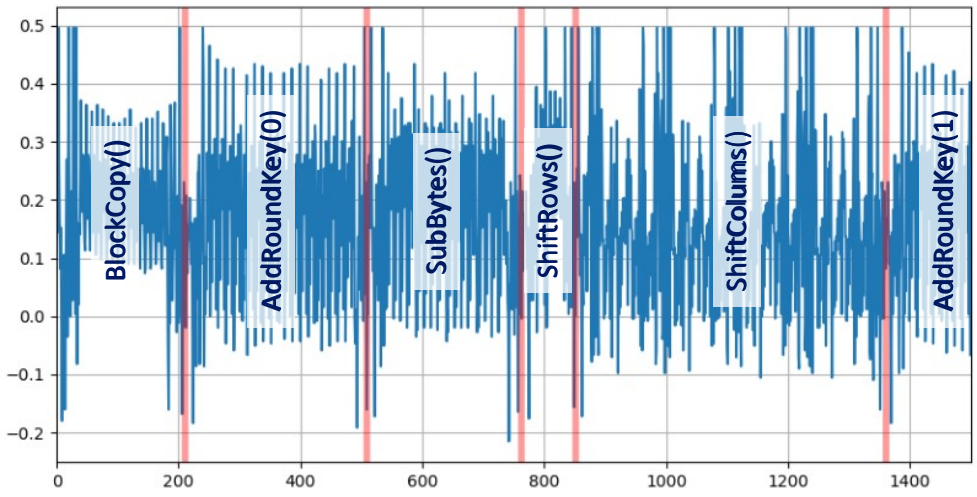

Figure 3.14: First round sampled current tagged.

This approach is known as ***SPA*** (Simple Power Analysis), generally, it is applied to gain some insight of the algorithm execution [24].

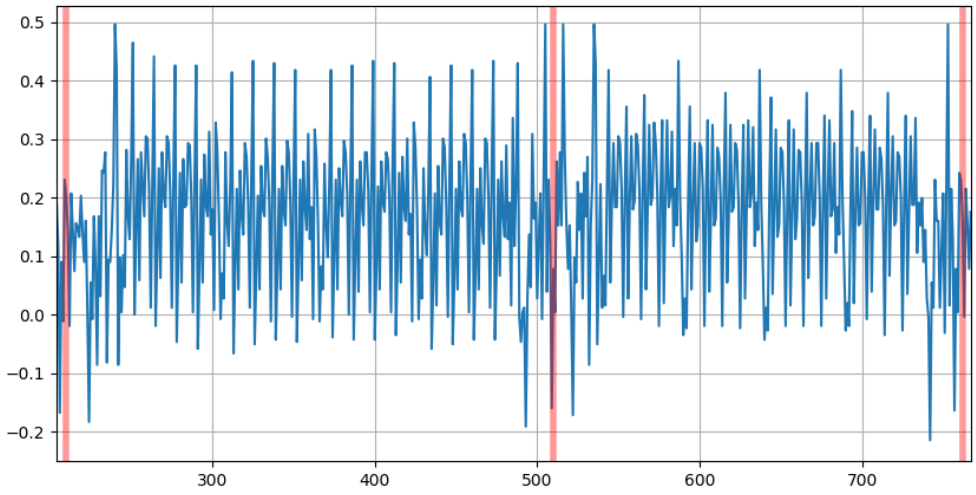

Figure 3.15: AddRoundKey(0) and SubBytes() details of first round sampled current.

3.6.3 Points of Attack

Analyzing both the code and the sampled traces we need to identify the possible *points of attacks*:

- *BlockCopy()*: here ve have the copy of the plain text to state memory, it is clearly visible but of no interest because it contains only known information.
- *AddRoundKey(0)*: the plain text is XORed with the secret KEY and stored to the state memory, this is an interesting point because it hosts the secret data we want to reveal; anyway it could be easily shielded with small code modifications to embed the EXOR function in the call of *SubBytes()* at a byte level. Anyway the firmware we analyzed doesn't apply this optimization.
- *SubBytes()*: this is a good point of attack because it involves the secret KEY and two known operations.
- The other functions adds additional transformation that makes the job of secret KEY identification harder, mainly because the single KEY byte influences more than one memory write.

The Side Channel Power Analysis attack is now applied at the two identified **points of attacks**: *AddRoundKey(0)* and *SubBytes()* whose trace is depicted in detail in Figure 3.15.

3.6.4 Differential Power Analysis

To attack the devices leveraging the defined leak model the ***DPA*** Differential Power Analysis [25] can be used.

DPA relies on the statistical relevance of power absorption differences caused by different data content or execution path. This is a statistical approach, it requires multiple acquisitions to be divided into two groups to evaluate the means difference with a guessed secret data and compare the expected results and acquired tracks. This method can't be applied to single tracks because of the acquisition noise, caused mainly by other circuit components working together with the observed device (just think at the pipeline or other microprocessor IP causing unexpected power absorption peaks).

To gather good results every group of tracks should be composed by random data where only one parameter, selected via the guessed secret value, is common (let say the value of a bit during a memory write operation).

If there is a deterministic correlation the two tacks groups mean values difference highlights it with identifiable peak (either positive or negative) otherwise it will be close to zero (just think at the difference of two random sequences average value: if the amount of elements is big enough the result will be close to zero), see Figure 3.16.

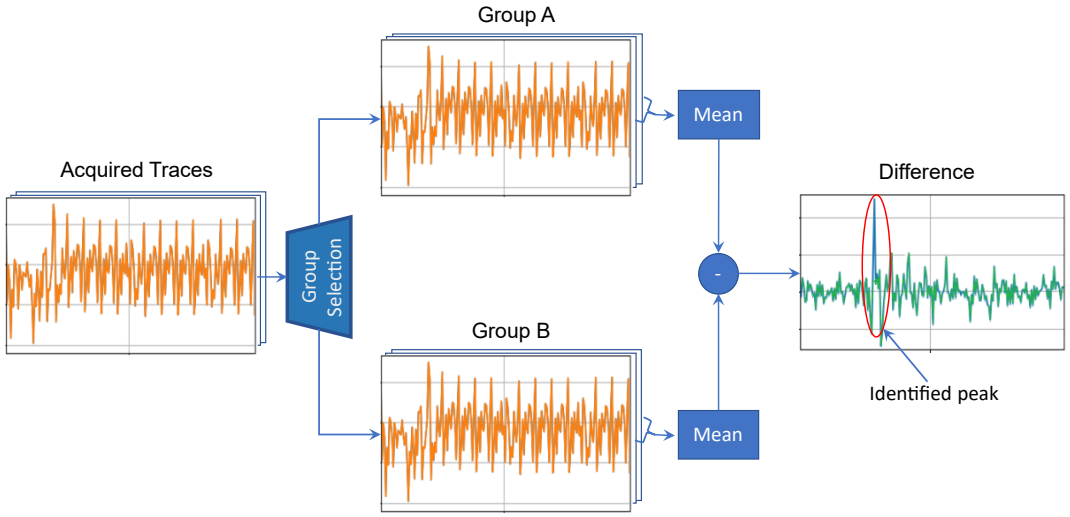

Figure 3.16: Differential Power Analysis schema.

As stated before the traces grouping can be made using different selection criteria, in any case, it starts trying to guess the secret values we want to identify and classify the acquired tracks according to data value expected using the guess applied to the supplied plain text.

The expected result is defined through a offline execution of the relevant part of the AES algorithm on the known plain text supplied for each current trace acquisition. Let's start at the first *Point of Attack* identified for AES algorithm *AddRoundKey*, it simply applies the Equation 3.7:

$$OutValue[x] = PlainText[x] \oplus KEY[x] \tag{3.7}$$

We have now an expected value for each state byte where the guessed KEY byte has been applied, and it is a different result for each trace we have been acquired.
To group the results we work at a single byte level and we identify a selection criterion, the easiest and most common is based on the value of a specific bit in the relevant *OutValue*.

As example the result of DPA on three guesses of *KEY[0]* values with group criteria on *OutValue[0]* bit 0 is shown in Figure 3.17. The green and orange traces are completely superposed, this is the reason why only the green trace is visible since it is the last one plotted. It is clearly visible that the average trace differences give results value in the magnitude of 0,005 not visible if we simply compare the average traces where the amplitude of the variations are of about 0,6 (see Figure 3.15).

One of the most common ways used to identify the correct guess is to record the max track peak value associated with every guess in a set of guesses (in the above case 256 possible values of *KEY[0]*) and search for the maximum value to identify the right one.

It works most of times. Anyway sometimes the identified peak is not the one we are looking for, so it results in a wrong value identification. These unexpected peaks are called ***Ghost Peaks*** caused by other high HW data writing. In Figure 3.17 only three guessed values resulting averages differences traces are shown to avoid a too crowded graph, anyway, multiple positive peaks are visible: the two bigger peaks are at sample number 41 (guess 0x00) and at sample number 258 (guess 0x2b and 0xff).

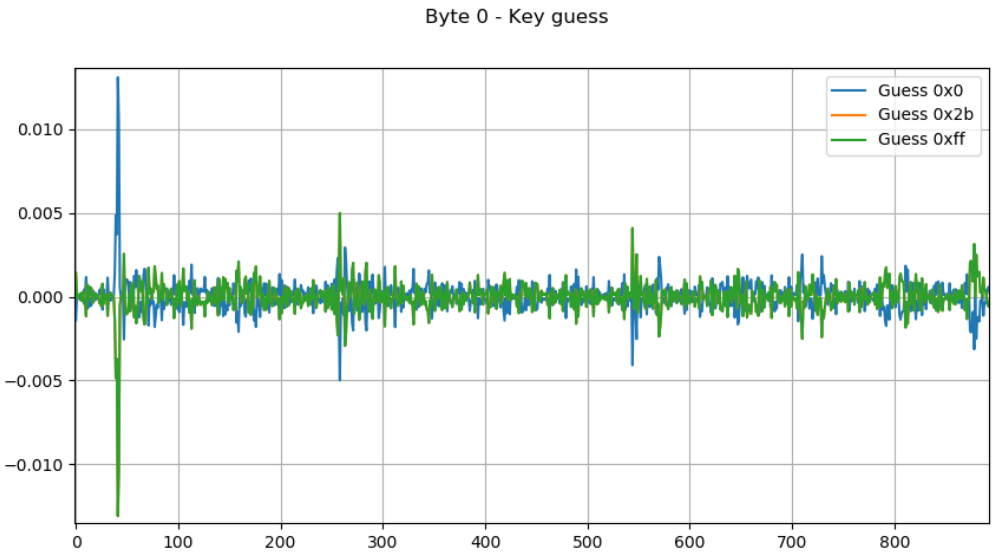

Figure 3.17: DPA on guess KEY[0].bit0 = 1.

The first peak is clearly ***Ghost Peak*** because it falls in the *BlockCopy()* window outside of the *AddRoundKey(0)* where we expect to see the wanted result. To avoid this wrong identification the max peak search will be reduced on a subset of trace samples (called a window) on the AddRoundKey(0) region: from sample 210 to 510 as depicted in Figure 3.18.

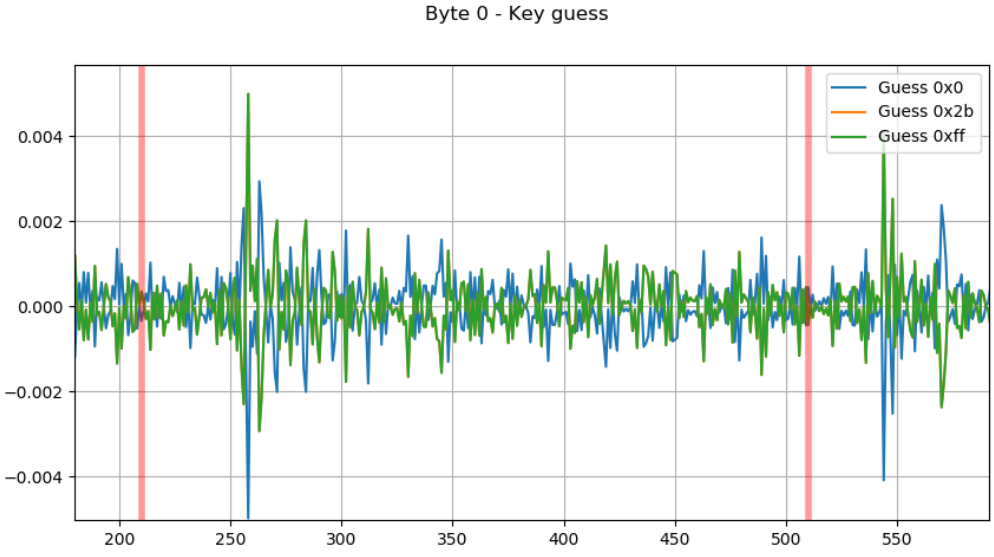

Figure 3.18: DPA on guess KEY[0].bit0 = 1 detail of *AddRoundKey(0)* window.

Running it for 256 single KEY byte values (0 to 255) and for all the KEY bytes give us the results reported in Figure 3.19.

```
SubKey n:00 guessed value:  0xff - real value 0x2b at position 258
SubKey n:01 guessed value:  0x00 - real value 0x7e at position 270
SubKey n:02 guessed value:  0xff - real value 0x15 at position 284
SubKey n:03 guessed value:  0x00 - real value 0x16 at position 296
SubKey n:04 guessed value:  0x00 - real value 0x28 at position 318
SubKey n:05 guessed value:  0x00 - real value 0xae at position 331
SubKey n:06 guessed value:  0x00 - real value 0xd2 at position 344
SubKey n:07 guessed value:  0x00 - real value 0xa6 at position 357
SubKey n:08 guessed value:  0xff - real value 0xab at position 380
SubKey n:09 guessed value:  0xff - real value 0xf7 at position 393
SubKey n:10 guessed value:  0xff - real value 0x15 at position 406
SubKey n:11 guessed value:  0x00 - real value 0x88 at position 418
SubKey n:12 guessed value:  0xff - real value 0x09 at position 441
SubKey n:13 guessed value:  0xff - real value 0xcf at position 454
SubKey n:14 guessed value:  0xff - real value 0x4f at position 467
SubKey n:15 guessed value:  0x00 - real value 0x3c at position 479
```

Figure 3.19: DPA on guess KEY[0].bit0 = 1 results with window restriction, red lines are errors.

Knowing the stored secret *KEY* values we can clearly see that all the results are wrong. This happens because of the bit-wise selection of *EXOR*, since the group has been selected on bit 0 the two group average values are superposed for all the KEYs with the same bit0 value of the secret KEY. As example the first byte correct value is **0x2b** $\rightarrow$ **0010 1011**$_\mathbf{b}$ identified as **0xff** $\rightarrow$ **1111 1111**$_\mathbf{b}$.

One possible solution is to iterate it for every bit testing only the relevant bits of guessed values 0x00 and 0xff are used to build the final value as a sequence of the single-bit values. In this case the results are more precise but some bit values identification are still wrong as shown in Figure 3.20.

```
 SubKey n:00 bit 00 guessed value:  1 at position 258
 SubKey n:00 bit 01 guessed value:  1 at position 258
 SubKey n:00 bit 02 guessed value:  0 at position 258
 SubKey n:00 bit 03 guessed value:  0 at position 254
 SubKey n:00 bit 04 guessed value:  0 at position 257
 SubKey n:00 bit 05 guessed value:  1 at position 258
 SubKey n:00 bit 06 guessed value:  0 at position 257
 SubKey n:00 bit 07 guessed value:  0 at position 257
--> SubKey n:00 guessed value:  0x23 - real value 0x2b
 SubKey n:01 bit 00 guessed value:  0 at position 270
 SubKey n:01 bit 01 guessed value:  1 at position 271
 SubKey n:01 bit 02 guessed value:  1 at position 271
 SubKey n:01 bit 03 guessed value:  1 at position 270
 SubKey n:01 bit 04 guessed value:  1 at position 271
 SubKey n:01 bit 05 guessed value:  1 at position 271
 SubKey n:01 bit 06 guessed value:  0 at position 269
 SubKey n:01 bit 07 guessed value:  0 at position 270
--> SubKey n:01 guessed value:  0x3e - real value 0x7e
```

Figure 3.20: DPA on guess KEY with: check for all bits results and window restriction, red lines are errors.

The first two bytes results, showed above, highlight a couple of wrong bit identification: byte 0 bit n.3 and in byte 1 bit n.6; in both cases the wrong bit identification sample position is a little different than the one used for the other bits in the same byte, this is clearly a sign of the mistake since the byte write instruction is done in parallel in the same clock cycle. It is due to the *Ghost Peak* visible in Figure 3.21.

Also this problem can be solved by windowing the traces to a narrow area, anyway the position of the *Ghost Peak* and the searched Peak are very close so other solutions are preferable.

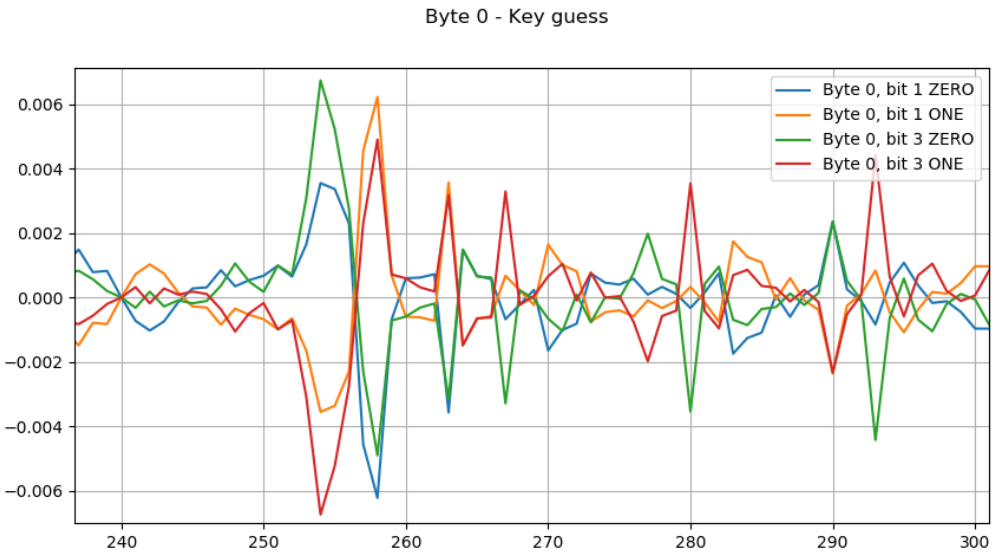

Figure 3.21: DPA bitwise on guess KEY[0].bit0 and bit 3.

A way to improve the DPA approach results is to change the grouping criteria involving more than 1 bit increasing the HW distance (see [26]), this should increase the distance of the two groups mean value. With this approach, we must go back to full *KEY* guesses list grouping on the expected results with HW >= 4 (i.e. at least 4 bits = 1) that gives us the results in Figure 3.22 where all secret *KEY* values have been correctly identified!.

```
SubKey n:00 guessed value:  0x2b - real value 0x2b at position 258
SubKey n:01 guessed value:  0x7e - real value 0x7e at position 271
SubKey n:02 guessed value:  0x15 - real value 0x15 at position 284
SubKey n:03 guessed value:  0x16 - real value 0x16 at position 296
SubKey n:04 guessed value:  0x28 - real value 0x28 at position 318
SubKey n:05 guessed value:  0xae - real value 0xae at position 332
SubKey n:06 guessed value:  0xd2 - real value 0xd2 at position 345
SubKey n:07 guessed value:  0xa6 - real value 0xa6 at position 358
SubKey n:08 guessed value:  0xab - real value 0xab at position 380
SubKey n:09 guessed value:  0xf7 - real value 0xf7 at position 393
SubKey n:10 guessed value:  0x15 - real value 0x15 at position 406
SubKey n:11 guessed value:  0x88 - real value 0x88 at position 418
SubKey n:12 guessed value:  0x09 - real value 0x09 at position 440
SubKey n:13 guessed value:  0xcf - real value 0xcf at position 454
SubKey n:14 guessed value:  0x4f - real value 0x4f at position 467
SubKey n:15 guessed value:  0x3c - real value 0x3c at position 480
```

Figure 3.22: DPA on guess KEY: group selection HW=4 and window restriction.

In Figure 3.23 the detail on first *KEY* byte correct identification (picture on the right) versus all the others (picrture on the left), the maximum peak in *AddRoundKey(0)* window among all the guesses is at sample number 258 with a value of about 0,025 while all the others are below 0,015.

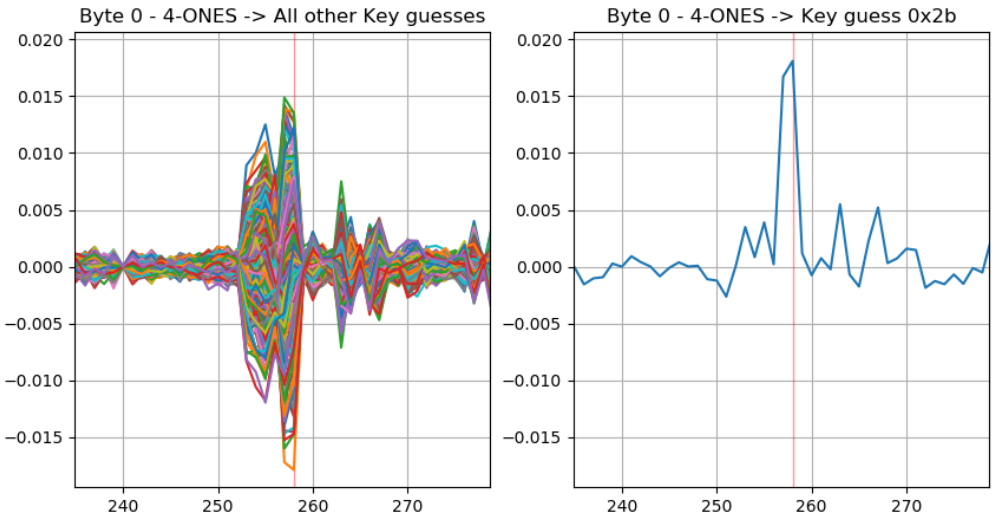

Figure 3.23: DPA AddRoundKey(0) HW=4, byte 0 identification.

The same approach can be applied to the *SubBytes()* window applying a Equation 3.8 to identify the expected values.

$$OutValue[x] = SubBytes(PlainText[x] \oplus KEY[x]) \tag{3.8}$$

In this case, we can try again a simpler group classification, as like the bit 0 value, because the two groups' bits values are not correlated to the used KEY, so the average can lead to better results as shown in Figure 3.24.

```
SubKey n:00 guessed value:  0x2b - real value 0x2b at position 548
SubKey n:01 guessed value:  0x7e - real value 0x7e at position 598
SubKey n:02 guessed value:  0x15 - real value 0x15 at position 648
SubKey n:03 guessed value:  0x16 - real value 0x16 at position 698
SubKey n:04 guessed value:  0x28 - real value 0x28 at position 559
SubKey n:05 guessed value:  0xae - real value 0xae at position 609
SubKey n:06 guessed value:  0xd2 - real value 0xd2 at position 659
SubKey n:07 guessed value:  0xa6 - real value 0xa6 at position 710
SubKey n:08 guessed value:  0xab - real value 0xab at position 570
SubKey n:09 guessed value:  0xf7 - real value 0xf7 at position 620
SubKey n:10 guessed value:  0x15 - real value 0x15 at position 670
SubKey n:11 guessed value:  0x88 - real value 0x88 at position 721
SubKey n:12 guessed value:  0x09 - real value 0x09 at position 581
SubKey n:13 guessed value:  0xcf - real value 0xcf at position 631
SubKey n:15 guessed value:  0x3c - real value 0x3c at position 731
```

Figure 3.24: DPA on SubBytes() attack results.

In Figure 3.25 the detail on first secret *KEY* byte correct identification (picture on the right) versus all the others (picture on the left), the maximum peak in *SubBytes(0)* window among all the guesses is at sample number 548 with a value of about 0,0094 while all the others are below 0,0056.

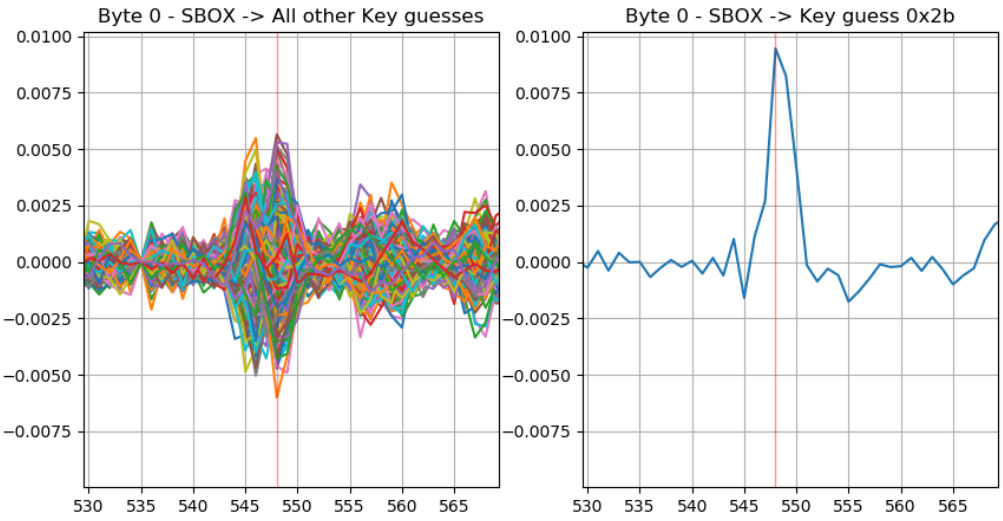

Figure 3.25: DPA SubBytes(0), byte 0 identification.

Both these approaches can lead to the correct identification of the secret KEY using some insight into how the algorithm is implemented.

A way to classify the efficiency of the used method is to see how many traces are needed to get to the desired results. To get it the acquired data traces has been shuffled to get three independent arrays of randomly distributed series of acquired data coupled with supplied plain text and evaluated increasing the amount of traces to be evaluated in term of number of correctly identified secret KEY bytes, to obtain a stable value the average of the results obtained on the three shuffled data sets is used as correctness indicator and shown in the graphs.

Figure 3.26 shows the required traces number using the *AddRoundKey()* with group selection on $HW >= 4$; about 320 traces are needed in this case.

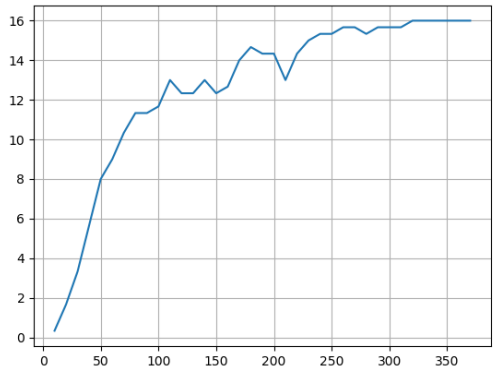

Figure 3.26: DPA *AddRoundKey(0)*,with group selection on $HW >= 4$, traces needed to get correct secret KEY identification.

Figure 3.27 shows the required traces number using the *SubBytes()* with group selection on *bit 0 value*, about 600 traces are needed in this case.

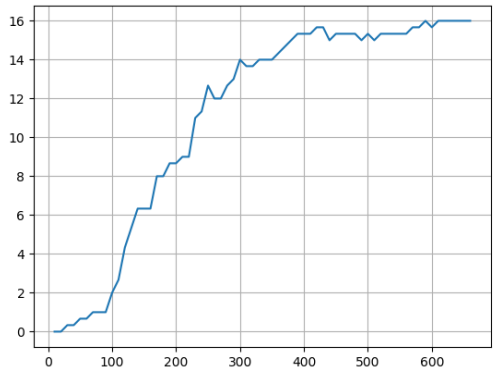

Figure 3.27: DPA *SubBytes(0)* with group selection on *bit 0 value*, traces needed to get correct secret KEY identification.

Figure 3.28 shows the required traces number using the *SubBytes()* with group selection on *HW* $>=$ *4*, about 80 traces are needed in this case.

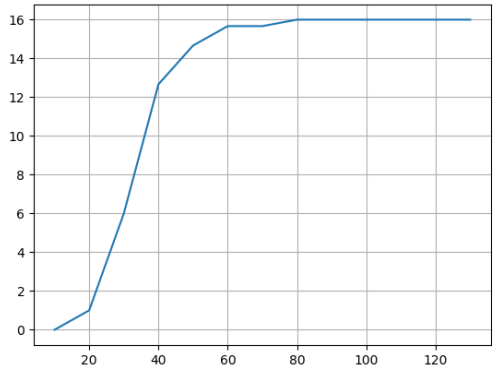

Figure 3.28: DPA *SubBytes(0)* with group selection on *HW* $>=$ *4*, traces needed to get correct secret KEY identification.

So the *HW* criteria used to group the traces greatly improve the detection effectiveness reducing the needed amount of acquired traces.

3.6.5 Correlation Power Analysis

A further improvement can be achieved using the *Correlation Power Analysis* algorithm (*CPA* form here on) [27].

It works applying the *Pearson Correlation Coefficient* (*PCC* from now on), shown in Equation 3.9, to select the most probable secret KEY value among a relatively small amount of traces.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \cdot \sigma_Y} \tag{3.9}$$

The above Equation can be written for discrete samples sequences x and y resulting in the Equation 3.10 where $\bar{x}$ and $\bar{y}$ are the mean value of x and y respectively.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{3.10}$$

The *PCC* gives us a number that shows how much two sets of data are linearly correlated; the result is a number in the range [-1,1] where the magnitude says how well the two data series are linearly correlated: 1 is perfect correlation while 0 is uncorrelated, and the sign gives us the slope of correlation.

CPA uses the same leak model used for *DPA* where power absorption is a linear function if the *Hamming Weight* of the data to be written in memory added to some noise, see Equation 3.11.
With this assumption if a and b are constant the *PCC* magnitude will be 1 with a sign equal to the sign of a. Note that we don't need to evaluate the values of a and b, we just assume they will have small variations during the complete data acquisition procedure.

$$Pa = a \cdot HW(Ex) + b \tag{3.11}$$

One of the arrays to be used is the expected power (let's call it y): one element for each acquired trace is calculated for every secret KEY guess and the trace relevant plain text byte as depicted in Figure 3.29.

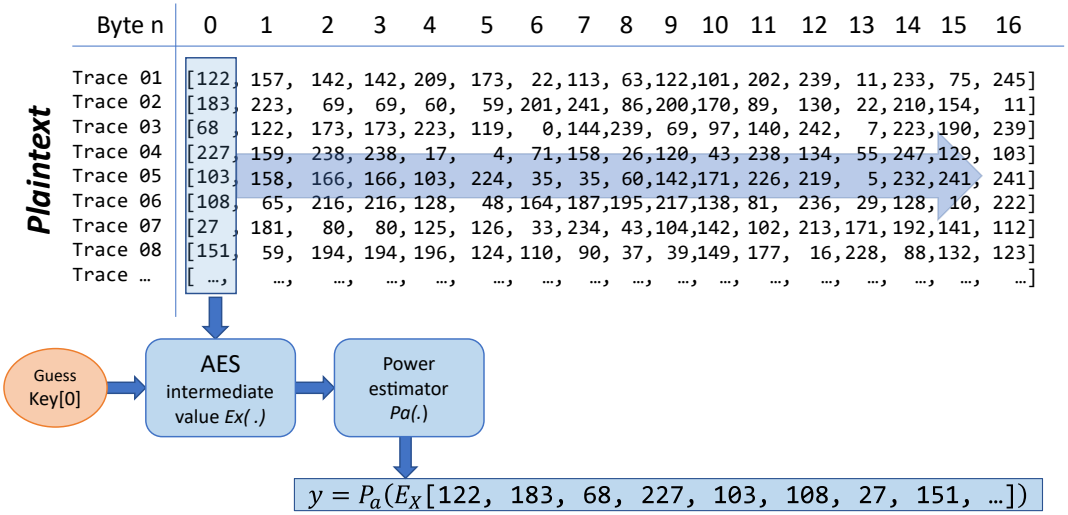

Figure 3.29: Power array estimator for guesses on Key[0].

The second array (let's call it x) is made of the samples of all acquired trace at the same relative time (i.e. the sample number 0 of 20 traces is an array of 20 elements) as depicted in Figure 3.30.

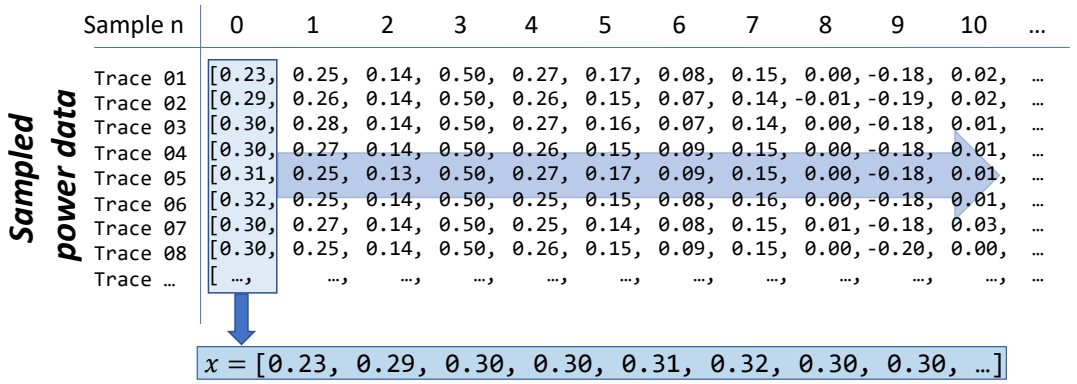

Figure 3.30: Sample array from acquired traces at sample n. 0.

The PCC correlation coefficient is computed for each sample using the same y array and the x array related to the sample under test. The result is an array with the same length of a trace made of PCCs, see Figure 3.31. Only the maximum value of this array (and eventually its position) is relevant for the single-byte key identification. In general, it is considered as absolute value because the HW to Absorbed Power relationship can be either positive or negative depending on how the data capture is made, in CWNano case we know it is positive so we can focus on positive values only. This process is then repeated for all the key guesses and for all the key bytes to discover the complete secret key.

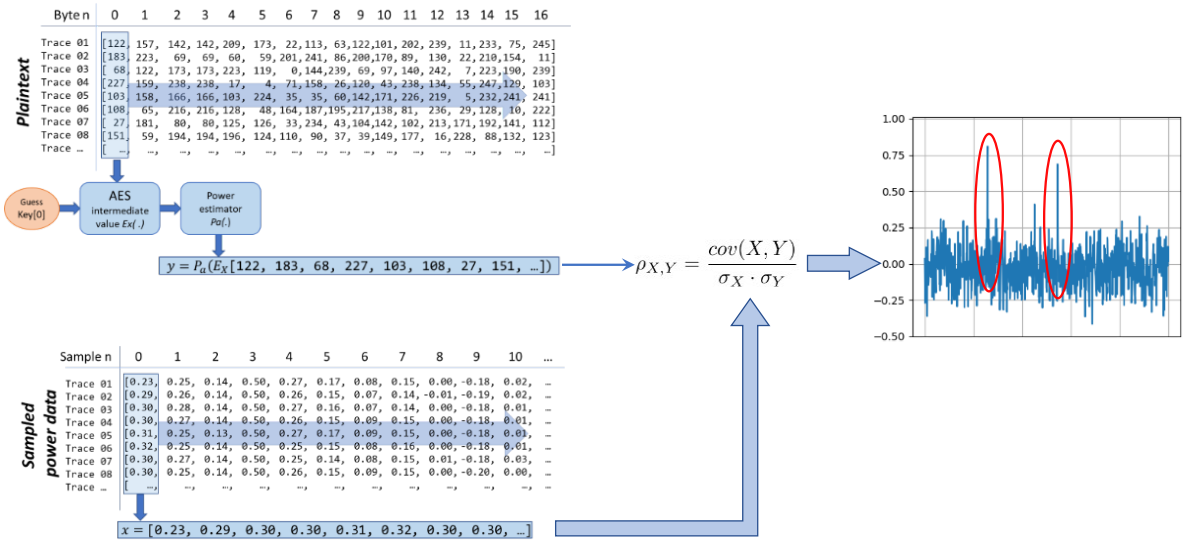

Figure 3.31: Correlation Power Analysis schema.

As first step it is applied to the first *Attack Point*: the output of *AddRoundKey()* function as shown in Figure 3.32.

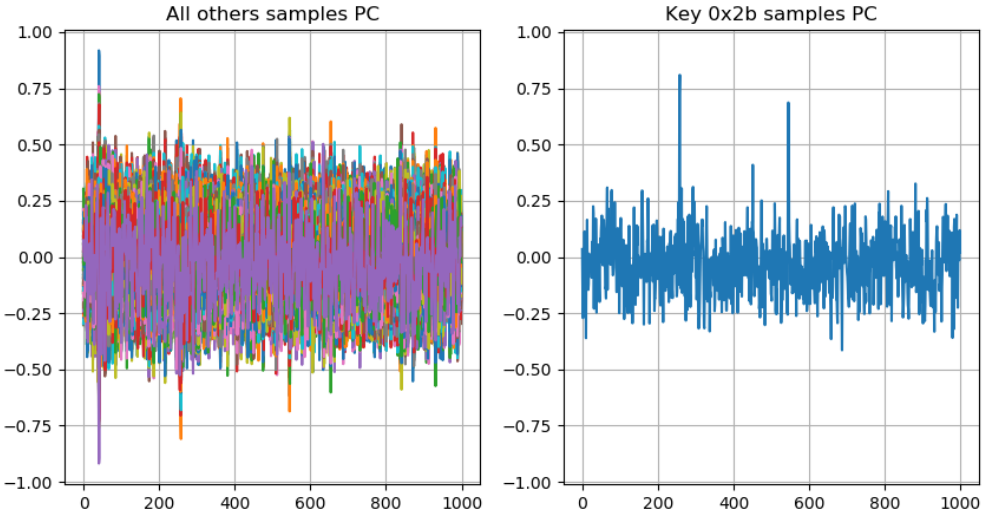

Figure 3.32: *PCC* on *AddRoundKey(0)* first 1000 samples.

Again there is a *Ghost Peak* visible in the *BlockCopy()* function window; this is easily explainable because when the KEY guess is 0 the AddRoundKey() output is exactly the plain text value, that is the data managed by *BlockCopy()* function, the acquired samples details are shown in Figure 3.33.

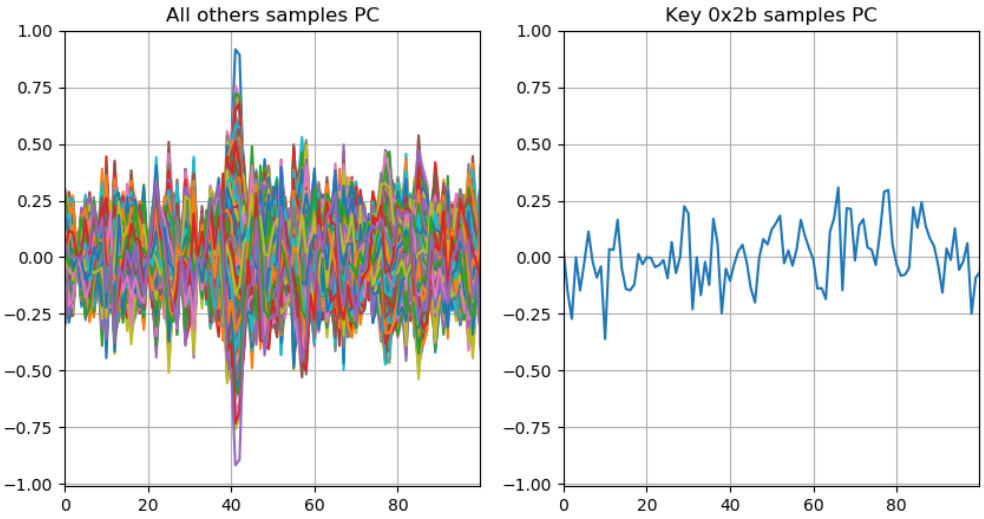

Figure 3.33: *PCC* on *AddRoundKey(0)* Ghost Peak detail in *BlockCopy()* window.

The searched correlation is present later, at higher sample index, as depicted in Figure 3.34.

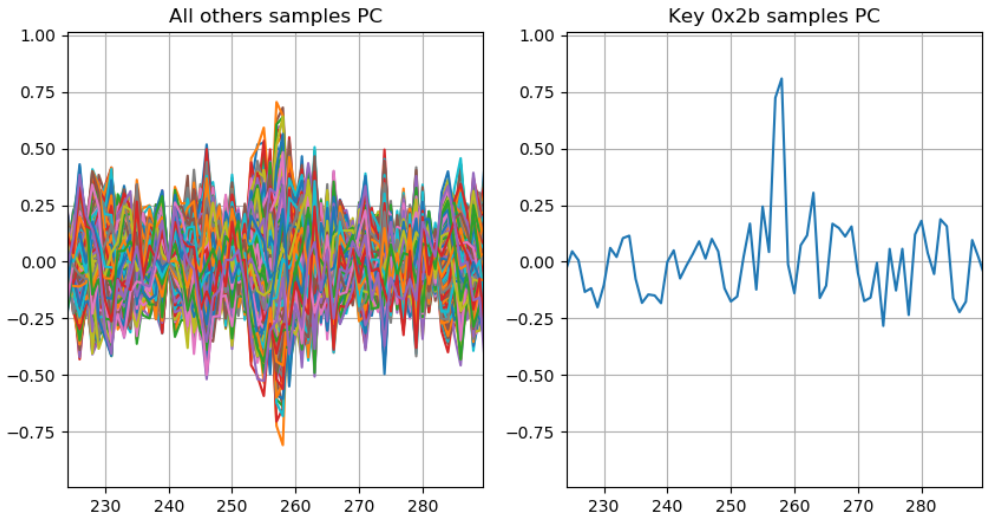

Figure 3.34: *PCC* on *AddRoundKey(0)* Peak detail in *AddRoundKey(0)* window.

The complete process with the max research restricted to *AddRoundKey(0)* window is able to identify the correct secret key, see Figure 3.35.

```
byte:00 - 2b OK - Pos:  258 PCC: 0.8090464806115729
byte:01 - 7e OK - Pos:  271 PCC: 0.8354009375878052
byte:02 - 15 OK - Pos:  283 PCC: 0.8342175527632381
byte:03 - 16 OK - Pos:  297 PCC: 0.7843584960046946
byte:04 - 28 OK - Pos:  318 PCC: 0.8588368993055520
byte:05 - ae OK - Pos:  332 PCC: 0.8537232609962971
byte:06 - d2 OK - Pos:  345 PCC: 0.8791922099081197
byte:08 - ab OK - Pos:  380 PCC: 0.7599447137533424
byte:09 - f7 OK - Pos:  393 PCC: 0.8119241752369173
byte:10 - 15 OK - Pos:  405 PCC: 0.8661184533292429
byte:11 - 88 OK - Pos:  419 PCC: 0.8362670369931144
byte:12 - 09 OK - Pos:  440 PCC: 0.7978349837789444
byte:13 - cf OK - Pos:  454 PCC: 0.6730465814729092
byte:14 - 4f OK - Pos:  467 PCC: 0.8107924650751189
byte:15 - 3c OK - Pos:  480 PCC: 0.7696166996511269
```

Figure 3.35: *PCC* on *AddRoundKey(0)* results restricted to *AddRoundKey(0)* window.

Applying *CPA* to the second *Attack Point*: the output of *SubBytes(0)* function as shown in Figure 3.36.

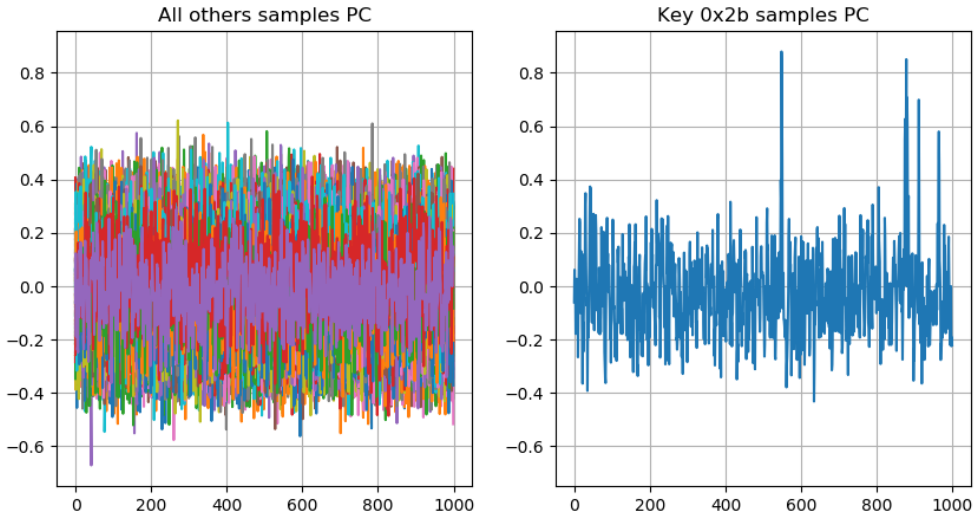

Figure 3.36: *PCC* on *SubBytes(0)* first 1000 samples.

The searched correlation peak detail is visible in Figure 3.37, note that in this case the *Ghost Peak* is not present at the beginning of the trace, it happen because there is no correlation between the searched Power Estimated array and the Plain Text managed by *BlockCopy()* function.

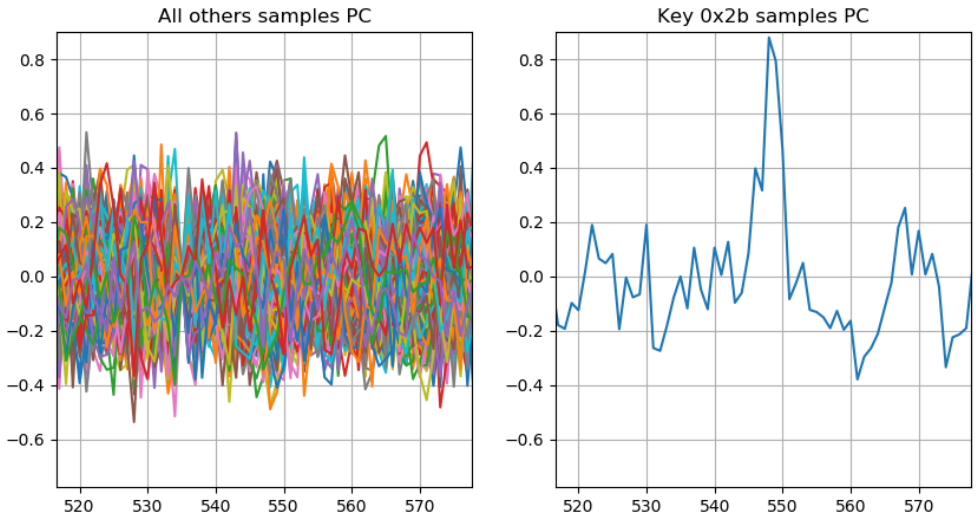

Figure 3.37: *PCC* on *SubBytes(0)* Peak detail.

The complete process applied on *SubBytes(0)* is able to identify the correct secret key without window restriction as shown in Figure 3.38:

```
byte:00 - 2b OK - Pos:  548 PCC: 0.8251359345665976
byte:01 - 7e OK - Pos:  598 PCC: 0.9028572174914622
byte:02 - 15 OK - Pos:  648 PCC: 0.8667232746940737
byte:03 - 16 OK - Pos:  698 PCC: 0.9162618575062188
byte:04 - 28 OK - Pos:  559 PCC: 0.8239385753611774
byte:05 - ae OK - Pos:  609 PCC: 0.8117521284154102
byte:06 - d2 OK - Pos:  660 PCC: 0.8144383121109557
byte:08 - ab OK - Pos:  570 PCC: 0.8639884468362278
byte:09 - f7 OK - Pos:  620 PCC: 0.8695165115842424
byte:10 - 15 OK - Pos:  889 PCC: 0.8920412937657465
byte:11 - 88 OK - Pos:  721 PCC: 0.8353320933038173
byte:12 - 09 OK - Pos:  581 PCC: 0.8968311919205509
byte:13 - cf OK - Pos:  631 PCC: 0.8963570395501116
byte:14 - 4f OK - Pos:  681 PCC: 0.9204068436012930
byte:15 - 3c OK - Pos:  731 PCC: 0.9076983019850139
```

Figure 3.38: *PCC* on *SubBytes(0)* results with no window restriction.

As done before Figure 3.39 shows the required traces number to get to complete secret KEY correct identification over 3 random traces selection in case of *CPA* applied to *AddRoundKey(0)*: about 95 traces are needed.

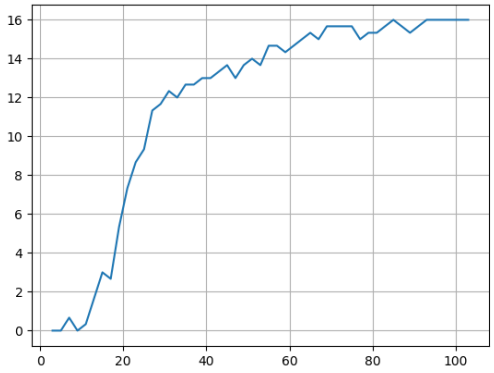

Figure 3.39: *CPA* on *AddRoundKey(0)*, traces needed to get correct KEY identification.

Figure 3.40 shows the required traces number to get to complete secret KEY correct identification over 3 random traces selection in case of *CPA* applied to *SubBytes(0)* : about 30 traces are needed.

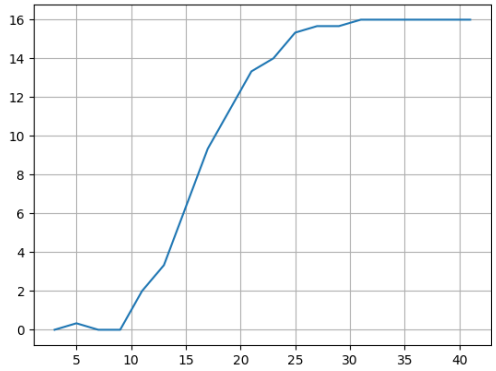

Figure 3.40: *CPA* on *SubBytes(0)*, traces needed to get correct KEY identification.

Chapter 4

Conclusion

4.1 Results Evaluation

The work has been carried out looking at two possible Attack Points and two methods to analyze the acquired data traces.

As regards the Attack Points, it has been highlighted that it is preferable to choose a point in which the data saved in the memory has a poor correlation, at single-bit level, with the plain text supplied to the encryption algorithm. Despite this, it is still possible to choose a non-optimal Attack Point at the price of using a higher number of acquired data traces.

The possibility of attacking an algorithm in two points allows, if necessary, to reconstruct the part of the algorithm between these points if unknown or partially known. The look-up table implemented by the SubBytes() function can be rebuilt after the identification of the secret KEY by exploiting the AddRoundKey() Attack Point and then focusing on the output of SubBytes() using a selected plain text with the aim of identifying the bytes stored in the look-up table. In this case, this activity was useless since the look-up table contents are clearly reported in the AES algorithm described in [21].

As reported in Table 4.1 DPA requires, in general, more data traces than CPA to get a correct KEY identification; furthermore, DPA requires to have a better separation between the groups in terms of Hamming Weight to be able to correctly identify the secret KEY. This means that if a small amount of data traces are available and the plain text has not a good random distribution, it could happen that one group is not big enough to evaluate a good average.

CPA is less expensive in terms of number of needed data acquisition traces, even if it is more complex from a computational perspective.

	Attack Points	
Algorithm	**AddRoundKey()**	**SubBytes()**
DPA (bit 0)	Don't work	600
DPA (HW=4)	320	80
CPA	95	30

Table 4.1: Required traces for a successful attack: results comparison.

The comparison results can be summarized as follow, in term of needed data:

- CPA works better than any DPA (around 3 times better)
- DPA works better in selecting groups with higher hamming weight distance (around 7 times)
- Attacks points where hamming weight shows higher un-correlation with plain text data works better

The computational effort required by both methodologies is small enough to be easily managed by a standard performance PCs, like the ones typically used in a home and/or professional environment.

The algorithms' implementation doesn't require the use of expensive or complicated software environments, they can be easily coded in PYTHON that is a completely free and available on the net, accompanied by some libraries, also free; this is the most chosen way.
The availability of a lot of examples with relative implementation guides makes these methodologies easily accessible even to relatively inexperienced users.

A stumbling block is still the hardware side of the work needed to acquire the desired data traces. Even if it is simple, it still requires adequate tools and skills necessary to modify/make an interface for acquiring the physical quantity under investigation (the absorbed current in the case of this work). The inexpensive tools available today, as like as the ChipWhisperer used for this work, can give access to these methodologies to a to a larger audience of users, anyway the relatively low sample rate and the need to interface it to a real victim still create a sort of barrier to the completely inexpert user.

Some different configurations, using more sophisticated instruments like a PICO Scope or a standard High Performance Digital Acquisition Oscilloscope and a ChipWhisperer used to manage only the communication with the target and the relevant acquisition trigger, are documented in the net, in this case the tools cost considerably increase.

4.2 Known Countermeasures

Attack methodologies undergo continuous evolution as well as countermeasures to make them less effective. The delay in identifying countermeasures and the difficulty of implementing them on devices already released on the market are determining factors in the choice of their use.

Surely all the countermeasures that require hardware changes can be taken into account for new products and by introducing changes on products already in production, raising their level of modification, even if it is much more difficult and expensive. A higher level of difficulty is for the products already delivered to customers. Sometimes, if the product is chip enough, as like a smart card used to access bank accounts, they can be replaced by throwing away the old devices.

For complex and costly products, countermeasures that only impact software are more appropriate. In many cases these products have the possibility of receiving updates remotely, think at a router or a mobile phone, making the update easier and interrupting its operation for a much shorter time.

The countermeasures goal is to hide the relevant side channel information leak leveraging on the leak model used. With reference to [28] and [29] a non-exhaustive list of countermeasures, both hardware and software, that can be adopted today to deal with the attack methodologies described in this work are:

- *Hiding*: this kind of countermeasure is typically implemented in hardware; its goal is to avoid the data dependency on current absorption due to different behavior of HIGH to LOW and LOW to HIGH switching (HW). Many different implementations have been identified, anyway the most referenced one uses memory writes with both the value and its opposite logical value, as in the Dual Rail Logic [30]; less complex solution tries to filter off the high frequency part of the absorbed current sometimes with a simple capacitor placed very close to the relevant power supply pins or more complex solution as [31].
  This method can also be applied in Software, balancing the number of set bits processing in a word containing both the valid data and its complement.
  This is, in theory, a very effective method; anyway the data dependency is not completely removed due to physical differences present in different parts of the device, usually it can be bypassed with more precise measurements and signal processing analysis.

- *Increasing noise*: adding noise, both through Hardware noise generators on chip, executing different instructions in parallel, widening data path or in Software doing things in Parallel, is another way to hide the exploitable power components. Also these solutions do not hide completely the exploitable side channel power absorption differences, anyway can make the attack required work harder.

- *Masking*: or *secret sharing* can be implemented both in Hardware or Software. The goal is to randomize the intermediate values keeping the same input and output as the standard algorithm. In Hardware it is typically achieved through *Masked Logic* elements, working at gate level using a bit mask sequence randomly selected applied to the masked elementary logic function. In Software it could be achieved in a similar way [32], as example by applying (EXOR) the random mask to a certain part of the algorithm and again at the end in a way that doesn't change the algorithm results, also in this case the mask is randomly chosen and changed at every algorithm execution or sooner if possible.
- *Random shuffle*: implemented both in Hardware or Software [33] the goal of this method is to avoid the time correspondence (sample position) of different traces.
- *Random instruction insertion*: implemented both in Hardware [34] or Software the goal of this method is doth to avoid the time correlation (sample position) of different traces and to add some noise due the unexpected instruction execution.

4.3 Future Developments

As in a war the exploitation of a vulnerability linked to a side-channel and the identification of the relative countermeasures follow each other until a definitive solution is identified, perhaps involving several technologies or making it cheaper to direct efforts on another side-channel or methodology.

All the countermeasures listed above are intended to make the Side-Channel based attack much more complicated by reducing its effectiveness in terms of costs/benefits. The natural continuation of this work consists in studying the countermeasures listed in the previous chapter and the related attack methods. Several activities are already present the literature (see [35]) such as higher-order attacks [36], alignment algorithms [37] and acquired data noise removal [38].

One additional possible further evolution is the study of a way to automate the identification of the used countermeasures, one or more, and define the right methodology or set of methodologies that ensure to put in place an effective attack.

Bibliography

[1] S. Heath, "1 - what is an embedded system?," in *Embedded Systems Design (Second Edition)* (S. Heath, ed.), pp. 1 – 14, Oxford: Newnes, second edition ed., 2002.

[2] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, and J. Lopez, "A survey of IoT-enabled cyberattacks: Assessing attack paths to critical infrastructures and services," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3453–3495, 2018.

[3] S. Kumar, H. Kumar, and G. R. Gunnam, "Security integrity of data collection from smart electric meter under a cyber attack," in *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, pp. 9–13, 2019.

[4] "U.s. gao - vehicle cybersecurity: Dot and industry have efforts under way, but dot needs to define its role in responding to a real-world attack." https://www.gao.gov/products/GAO-16-350. (Accessed on 08/31/2020).

[5] C. Biesecker, "Connected aircraft open new cyber threat vectors to commercial aviation, thales usa chief warns." https://www.defensedaily.com/connected-aircraft-open-new-cyber-threat-vectors-commercial-aviation-thales-usa-chief-warns/cyber/, 2018. (Accessed on 01/01/2021).

[6] Kskhh, "A sony bravia smart tv showing the home screen." (CC BY-SA 4.0).

[7] S. Soltan, P. Mittal, and H. V. Poor, "Blackiot: Iot botnet of high wattage devices can disrupt the power grid," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 15–32, USENIX Association, Aug. 2018.

[8] S. Fruitsmaak, "An artificial pacemaker (serial number 1723182) from st. jude medical, with electrode. the body of the device is about 4 centimeters long, and the electrode measures roughly 58 centimeters.." (CC BY 3.0).

[9] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 268–286, IEEE, 2017.

[10] P. Prinetto and G. Roascio, "Hardware security, vulnerabilities, and attacks: a comprehensive taxonomy," CEUR Workshop Proceedings, 2020.

[11] FBI, "Poligraph picture, public domain." (Accessed on 01/12/2021).

[12] T. Hunkin, "Illegal engineering." https://www.timhunkin.com/94_illegal_engineering.htm. (Accessed on 11/06/2021).

[13] D. Easter, "The impact of 'tempest' on anglo-american communications security and intelligence, 1943–1970," *Intelligence and National Security*, pp. 1–16, jul 2020. (Accessed on 08/26/2020).

[14] U. S. G. P. Office, "Teletypewriter set 131b2." http://www.navy-radio.com/manuals/tty/fgq1-tm-11-2209.pdf, 1946. pag 59.

[15] H. Wang, D. Ji, Y. Zhang, K. Chen, J. Chen, and Y. Wang, "Optical side channel attacks on singlechip," in *Proceedings of the 2015 International Conference on Industrial Technology and Management Science*, Atlantis Press, 2015.

[16] D. Genkin, A. Shamir, and E. Tromer, "Acoustic cryptanalysis," *Journal of Cryptology*, vol. 30, pp. 392–443, feb 2016.

[17] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," in *Smart Card Research and Advanced Applications*, pp. 219–235, Springer International Publishing, 2014.

[18] RAMBUS, "Dpa workstation testing platform." https://www.rambus.com/security/dpa-countermeasures/. rev 04.

[19] "Cw1101 chipwhisperer-nano." https://rtfm.newae.com/Capture/ChipWhisperer-Nano/. (Accessed on 14/04/2021).

[20] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[21] "FIPS PUB 197: Advanced encryption standard (AES)," tech. rep., National Institute of Standards and Technology, nov 2001.

[22] "Cw1101 chipwhisperer-nano - api description." https://chipwhisperer.readthedocs.io/en/latest/api.html. (Accessed on 11/06/2021).

[23] F.-X. Standaert, "Introduction to side-channel attacks," in *Integrated Circuits and Systems*, pp. 27–42, Springer US, dec 2009.

[24] M.-L. Akkar, R. Bevan, P. Dischamp, and D. Moyart, "Power analysis, what is now possible...," in *Advances in Cryptology — ASIACRYPT 2000* (T. Okamoto, ed.), (Berlin, Heidelberg), pp. 489–502, Springer Berlin Heidelberg, 2000.

[25] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, pp. 5–27, mar 2011.

[26] Y. HAN, X. ZOU, Z. LIU, and Y. CHEN, "Efficient DPA attacks on AES hardware implementations," *International Journal of Communications, Network and System Sciences*, vol. 01, no. 01, pp. 68–73, 2008.

[27] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Lecture Notes in Computer Science*, pp. 16–29, Springer Berlin Heidelberg, 2004.

[28] L. Zhang, L. Vega, and M. Taylor, "Power side channels in security ics: Hardware countermeasures," May 2016.

[29] D. Das, M. Nath, S. Ghosh, and S. Sen, "Killing EM side-channel leakage at its source," IEEE, aug 2020.

[30] J.-L. Danger, S. Guilley, S. Bhasin, and M. Nassar, "Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors," in *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, IEEE, nov 2009.

[31] M. Kar, A. Singh, S. K. Mathew, A. Rajan, V. De, and S. Mukhopadhyay, "Reducing power side-channel information leakage of AES engines using fully integrated inductive voltage regulator," *IEEE Journal of Solid-State Circuits*, vol. 53, pp. 2399–2414, aug 2018.

[32] S. Bhasin, J.-L. Danger, S. Guilley, and Z. Najm, "A low-entropy first-degree secure provable masking scheme for resource-constrained devices," in *Proceedings of the Workshop on Embedded Systems Security - WESS '13*, ACM Press, 2013.

[33] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *Advances in Cryptology – ASIACRYPT 2012* (X. Wang and K. Sako, eds.), (Berlin, Heidelberg), pp. 740–757, Springer Berlin Heidelberg, 2012.

[34] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "A smart random code injection to mask power analysis based side channel attacks," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis - CODES mathplus ISSS '07*, ACM Press, 2007.

[35] C. Clavier, J.-S. Coron, and N. Dabbous, "Differential power analysis in the presence of hardware countermeasures," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, pp. 252–263, Springer Berlin Heidelberg, 2000.

[36] K. Lemke-Rust and C. Paar, "Gaussian mixture models for higher-order side channel analysis," in *Cryptographic Hardware and Embedded Systems - CHES 2007* (P. Paillier and I. Verbauwhede, eds.), (Berlin, Heidelberg), pp. 14–27, Springer Berlin Heidelberg, 2007.

[37] Q. Tian and S. A. Huss, "A general approach to power trace alignment for the assessment of side-channel resistance of hardened cryptosystems," in *2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, IEEE, jul 2012.

[38] T.-H. Le, J. Clediere, C. Serviere, and J.-L. Lacoume, "Noise reduction in side channel attack using fourth-order cumulant," *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 4, pp. 710–720, 2007.