Thwarting Fault Attacks using the Internal Redundancy Countermeasure (IRC)

Benjamin Lac[1,5], Anne Canteaut[2], Jacques J.A. Fournier[3], and Renaud Sirdey[4]

[1] CEA-Tech, Gardanne, France, [2] Inria, Paris, France,
[3] CEA-Leti, Grenoble, France, [4] CEA-List, Saclay, France,
[5] ENSM-SE, Saint-Étienne, France,
`{benjamin.lac, jacques.fournier, renaud.sirdey}@cea.fr`,
`anne.canteaut@inria.fr`

Abstract. A growing number of connected objects, with their high performance and low-resources constraints, are embedding lightweight ciphers for protecting the confidentiality of the data they manipulate or store. Since those objects are easily accessible, they are prone to a whole range of physical attacks, one of which are fault attacks against for which countermeasures are usually expensive to implement, especially on off-the-shelf devices. For such devices, we propose a new generic software countermeasure, called the Internal Redundancy Countermeasure (IRC), to thwart most fault attacks while preserving the performances of the targeted cipher. We report practical experiments showing that IRC successfully thwarts fault attacks on the block cipher PRIDE and on the stream cipher TRIVIUM for which we protect both the initialization and the keystream generation.

Keywords: IRC · Physical attacks · Fault attacks · SIMD instructions · Software countermeasure · Lightweight cryptography · IoT.

1 Introduction

The expansion of the Internet of Things (IoT) brings many benefits but also raises a number of issues with respect to security and privacy. Lightweight cryptography (LWC) is investigated in order to address IoT security issues while seeking the best trade-off between security, power consumption, performance and footprint. During the last few years, several lightweight block and stream ciphers have been proposed, like for example KLEIN [23], PRESENT [14], PRINCE [16], PRIDE [5], SIMON [8], SPECK [8], Grain [25], MICKEY [6] or TRIVIUM [17]. These ciphers are mainly designed to resist black-box mathematical attacks. However, since they are used in IoT devices in pervasive environments, implementation-related attacks must also be considered.

Resistance against side channel attacks [30] is now considered as a valuable property which should be taken into consideration when designing lightweight ciphers as seen in ciphers like FIDES [12], PICARO [40], Zorro [22] and the LS-designs family [24]. Another kind of physical attacks, based on fault injections, must also be considered [9]. Many such attacks have been introduced [11], [37], and the proposed countermeasures have significant impacts on the cryptographic implementations' performances and sizes, especially for off-the-shelf devices with no particular hardware mechanism to thwart such attacks.

In this paper, we introduce a new paradigm, called the *Internal Redundancy Countermeasure (IRC)*, for using spatial redundancies to thwart fault attacks. First, we describe the concept of IRC based on the use of SIMD (Single Instruction Multiple Data) instructions, which are increasingly available in off-the-shelf IoT devices: for 32-bit architectures, we work on 4 bytes in parallel. Then, we introduce a method for implementing this countermeasure in a completely generic way, i.e. independently of the cipher. Finally, we report practical experiments that show that IRC successfully thwarts real fault injections on the block cipher PRIDE and on the stream cipher TRIVIUM before discussing about the efficiency of this approach and concluding on some future work.

2 Fault Attacks

Usually, ciphers are constructed to resist black box mathematical cryptanalysis. However, most of them do not take into account implementation-related issues like vulnerabilities to physical attacks, which can be divided into three categories: invasive like reverse engineering, non-invasive like side-channel analysis and semi-invasive like fault attacks. The first one involves specific hardware-related phenomena, very much related to the way the integrated circuits running the ciphers are implemented. The second one is a particularly high threat to the way cryptographic algorithms are implemented and efficient countermeasures, with limited impact on performance, have been proposed (techniques based on masking for e.g). The third one has been the trickiest so far, as detailed below, due to the complexity of the different fault attack routes and the expensive countermeasures. For this reason, we decided first to focus on fault attacks.

2.1 Effects & Exploitation

Fault attacks consist in disturbing the behaviour of the circuit in order to alter the correct execution of the cipher. The faults are injected into the device by various means such as light pulses [44], laser [43], clock glitches [3], spikes on the voltage supply [13] or electromagnetic (EM) perturbations [19]. They can have different effects on the encryption (or decryption) process like an instruction-skip or an n-bit set, reset or flip - commonly $n = 1$, 8 or 32 according to the chosen injection means and to the targeted implementation. An instruction-skip can for example allow to bypass the last key addition layer, common to many ciphers, and thus to retrieve the key which is equal to the difference between the correct and the faulty ciphertexts. A bit set, reset or flip can allow to make a safe-error analysis [28]. It consists in disturbing the content of a conditional loop dependent on a key bit in order to retrieve its value by comparing the obtained ciphertext with the correct one: in case of equality, the condition is false, otherwise it is true. A bit set, reset or flip can also allow to perform differential fault analysis (DFA). DFA, originally described in [11], [15], consists in retrieving a secret key by comparing correct ciphertexts with faulty ones. DFA techniques have been described and applied to most publicly known ciphers going from symmetric-key ciphers like DES [11] or AES [42] to asymmetric ones like RSA [15] or even more complex schemes like Pairings [33]. In the particular field of lightweight cryptography, DFA have been proposed against ciphers like PRESENT [48], SPECK [46], PRINCE [45], PRIDE [31] or TRIVIUM [37].

Yet, these references are not exhaustive in terms of fault effects and in terms of their exploitation in real-life attacks. Thwarting fault attacks (at reasonable cost) is of the utmost importance. But it is not straightforward to devise countermeasures against such attacks because of the diversity of the possible injection methods and because the usually deployed countermeasures (like redundancy, error-correcting codes etc) have a serious impact on performances. In that respect our work focuses on the implementation of an efficient (in terms of computation time, code size and/or power consumption) countermeasure for IoT devices.

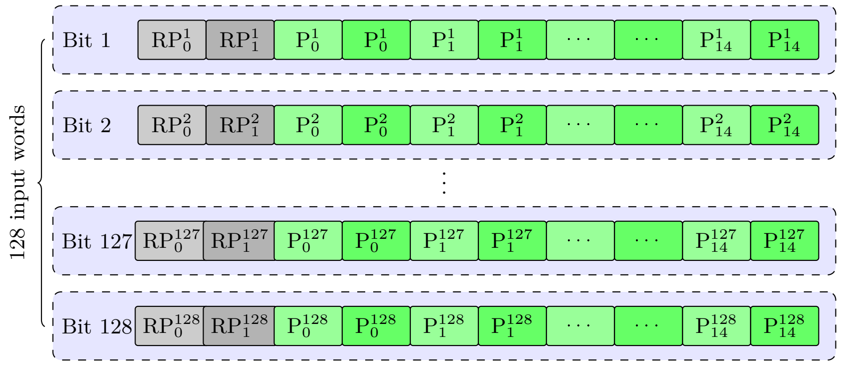
2.2 Countermeasures

Fault attacks can be circumvented using hardware or software countermeasures [26]. Regarding hardware countermeasures, passive shields, which are metal layers over the chip, allows to prevent optical fault injections [47]. However, it is possible to remove passive shields using chemical means and fault injections using EM pulses cannot be blocked by such shields neither. Active shields, which consist of wire meshes that run signals over the chip's surface and detect any interruption on a wire, are thus a very effective means to thwart many fault attacks. Furthermore, the use of light sensors [21] allows to detect anomalies in the circuit's behavior. The main drawback of such hardware countermeasures is their cost. The use of asynchronous design techniques as described in [38] can provide coding techniques that can detect some types of fault injections. Another proposed hardware solution is based on the insertion of parity checks on the data paths [10,27] in order to detect errors introduced by malicious fault injections. A different way of handling this security problem in hardware is to analyze the circuit at design time and adapt the countermeasures until all attacks fail at least at simulation time. This design-time security analysis is proposed in [34]. The problem with hardware countermeasures is that even though simulations may show that they are efficient, there is no absolute guarantee that this will be the case on the final chip where other considerations like final place & route, manufacturing processes etc. may have a huge impact on the countermeasures' efficiency. And by the time the final chips are obtained, if a security flaw is identified, it would be expensive to have another iteration of the design cycle to "patch" the design in hardware (in some cases metal fixes may be used but this would not apply to any part of the circuit). It is hence highly recommended to use these hardware countermeasures with software ones.

One of the basic principle behind most software countermeasures is to make sure that all the calculations' timings are independent from the data or key being manipulated, and to 'hide' the internal calculations of the cryptographic algorithm so that the attacker has no mastery of the data being manipulated and no means to understand what is happening. In the case of Public Key Algorithms like RSA (or ECC), techniques like *message blinding* and *exponent blinding* have been proposed [29,35]. Similar blinding techniques called *data randomisation* or *data masking* [4] have been proposed for Secret Key Algorithms. More generally, masking has been shown to offer protection against some fault attacks like DFA [32]. However, such an approach cannot thwart fault attacks like statistical fault attacks [20] or safe-error analyses. Redundancy is thus a very efficient means to thwart this latter kind of attacks. It consists in computing the same operations on one or several copies of the data, providing either a spatial or a temporal redundancy, and then in comparing the obtained results. Hence, to perform a fault attack, an attacker must obtain the same fault on both computations. It is thus possible to only detect the fault by trapping the system when all the copies do not all lead to the same end result or to correct the fault by applying a majority vote (by returning the one which appears the most) using 3 or more data copies. Thereby, redundancy allows to prevent all fault attacks without considering any fault model since it directly prevents the attacker from getting the fault result. However, each spatial (resp. temporal) copy costs a memory (resp. time) overhead equal to that of an additional operations [36]. Therefore, more and more research focus on trying to perform such redundancies at lower costs.

2.3 Intra-Instruction Redundancy

Recently, a countermeasure based on Intra-Instruction Redundancy [39] was proposed to thwart fault attacks. It consists in using a bit-sliced implementation of a given cipher applied on 32 input blocks. The aim is to exploit a 32-bit architecture - which is the most widely used architecture in IoT devices - taking as input 15 blocks of data interleaved with 15 blocks of redundancy and 2 reference blocks. The reference blocks are constant inputs (plaintexts and keys) for which the corresponding ciphertexts are known. Figure 1 shows an example of the composition of the input words protected by IIR applied on the 128-bit plaintexts $P_i = P_i^1 \cdots P_i^{128}$ with $0 \leqslant i \leqslant 14$ and using two 128-bit reference plaintexts $RP_0 = RP_0^1 \cdots RP_0^{128}$ and $RP_1 = RP_1^1 \cdots RP_1^{128}$.



128 input words
Bit 1
RP_0^1 RP_1^1 P_0^1 P_0^1 P_1^1 P_1^1 ⋯ ⋯ P_{14}^1 P_{14}^1
Bit 2
RP_0^2 RP_1^2 P_0^2 P_0^2 P_1^2 P_1^2 ⋯ ⋯ P_{14}^2 P_{14}^2
Bit 127
RP_0^{127} RP_1^{127} P_0^{127} P_0^{127} P_1^{127} P_1^{127} ⋯ ⋯ P_{14}^{127} P_{14}^{127}
Bit 128
RP_0^{128} RP_1^{128} P_0^{128} P_0^{128} P_1^{128} P_1^{128} ⋯ ⋯ P_{14}^{128} P_{14}^{128}

Figure 1: Bit-slicing with Intra-Instruction Redundancy

IIR principally allows to thwart mono-bit fault models thanks to the redundancy and also to thwart instruction skip thanks to the reference blocks. Indeed, an instruction skip will have the effect of changing the value of the reference block and will therefore be detected at the end of the encryption or decryption. Unfortunately, multi-bit fault models can still be effective: for example, a two-bit fault on the two copies of a data block will be undetectable. Moreover, IIR imposes to use, in most cases, a less efficient implementation of the cipher due to the Boolean circuit transformation overhead necessary for bit-slicing [41], to take as input 15 blocks of data per encryption and to use n words in order to store and manipulate an n-bit input. However, using reference blocks as part of a countermeasure is very effective against instruction skip. Thereby, we investigated the possibility of keeping this property while using a conventional (i.e. non-bitscliced) implementation of a cipher. Moreover, we also looked at the possibility to start from an efficient 8-bit implementation - which is usually the preferred option for lightweight ciphers - on a 32-bit architecture. Hence, we propose the following Internal Redundancy Countermeasure (IRC).

3 Internal Redundancy Countermeasure

3.1 General Principle

It is common to use a 32-bit implementation of a cipher on a 32-bit architecture in order to fully exploit the architecture's capabilities. However, the use of spatial redundancy in this case requires a larger memory overhead. In order to decrease it, we propose to use an efficient 8-bit implementation of the cipher simultaneously applied on 4 blocks on a 32-bit word. Indeed, we replace each 8-bit operator by means of a single stream of 32-bit instructions corresponding to the same operation performed independently on each byte in a SIMD fashion. This has a timing overhead since it generally requires more instructions than the original 32-bit implementation would but it highly decreases the required memory overhead since it uses a single stream of instructions instead of 4 parallel ones (which is not always possible according to the architecture). IRC is based on this concept but also uses reference blocks to increase the countermeasure's efficiency. The manipulated words are thus composed of one data byte interleaved with the corresponding byte of the reference block and two copies depending on the used cipher. Figure 2 shows a typical example of a 32-bit word as used in IRC.

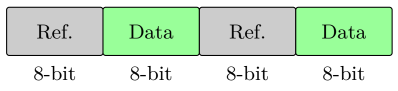


Figure 2: IRC's 32-bit word structure

Then, IRC executes the cipher by means of a single stream of 32-bit instructions operating independently on each byte. Finally, at the end of encryption or decryption, IRC makes comparisons involving the different copies and the stored reference ciphertext. IRC expects all corresponding copies to be equal to each other and each obtained reference ciphertext to be in turn equal to the stored reference ciphertext. Therefore, to perform a fault injection, an attacker must obtain the same fault on each copy of the data without affecting the reference block. This case is extremely difficult to control for the attacker in practice, especially when the reference block is interleaved between copies of the data. Moreover, it is possible to apply an 8-bit rotation to each word as described in Figure 2 at one or more temporal positions throughout the encryption or decryption since the operations are performed independently on each byte. It allows to make the positions of the data blocks unpredictable to the attacker. It is then possible to retrieve the data at the end of encryption or decryption by comparing the obtained bytes with the stored reference block. Now we will describe the different ways of using IRC depending on the type of cipher targeted.

3.2 IRC on Block Ciphers

The construction of the words depends on the required security level. Generally, there are two possibilities to prevent the same fault on k spatial copies of blocks:

i. *Fault detection:* use $k+1$ copies of the data in each word and trap the system when they do not all lead to the same end result. Note that, in this case, an attacker can make a safe-error analysis since she must only know that a fault has been injected to perform this kind of attacks.
ii. *Fault correction:* use $2k+1$ copies of the data in each word and return the one which appears the most, by applying a majority vote among them. It provides in this case an additional security against the safe-error attacks.

IRC offers the possibility of having either one of these two strategies. For fault detection, a representation as the one given in Figure 2 can be used. In fault correction mode, IRC can use a single reference block split into two nibbles (4-bit words). Then, each nibble is arranged between two copies of the data as depicted on Figure 3.

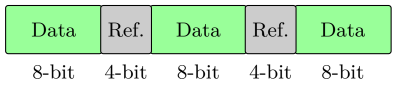
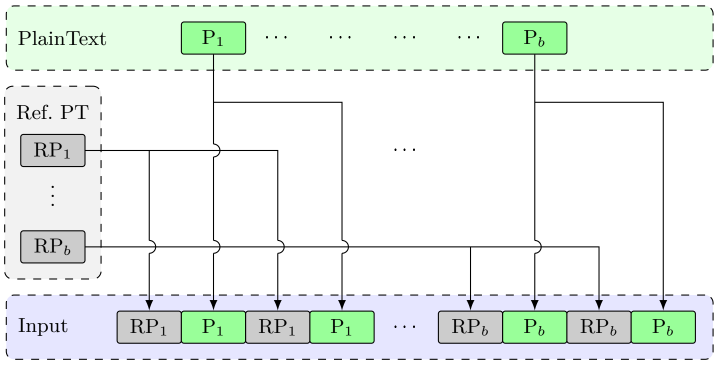


Figure 3: IRC's 32-bit word for a majority vote

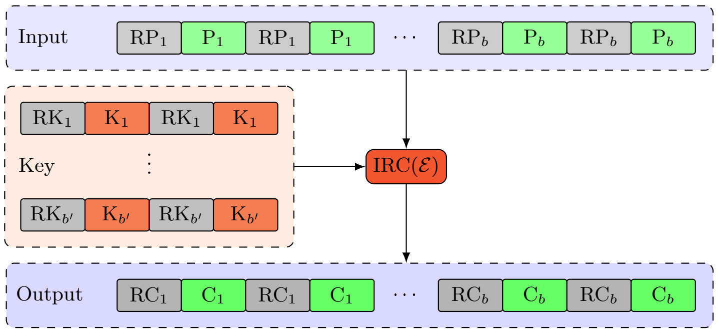
The counterpart of this latter method is that the nonlinear operators are more complex to implement, generally they are more expensive since IRC cannot use SIMD instructions. Now we will detail the case of the fault detection. Let $\mathcal{E}$ be an 8-bit implementation of a block cipher which takes a b-byte plaintext $\mathrm{P} = \mathrm{P}_1 \cdots \mathrm{P}_b$ as input, uses a b'-byte key $\mathrm{K} = \mathrm{K}_1 \cdots \mathrm{K}_{b'}$ and produces a b-byte ciphertext $\mathrm{C} = \mathrm{C}_1 \cdots \mathrm{C}_b$. IRC uses a b-byte reference plaintext $\mathrm{RP} = \mathrm{RP}_1 \cdots \mathrm{RP}_b$, a b'-byte reference key $\mathrm{RK} = \mathrm{RK}_1 \cdots \mathrm{RK}_{b'}$ and a b-byte reference ciphertext $\mathrm{RC} = \mathrm{RC}_1 \cdots \mathrm{RC}_b$. First, for each $i \in \{1, \cdots, b\}$, IRC stores in a 32-bit word the byte P_i concatenated with RP_i, P_i and RP_i as illustrated in Figure 4.



PlainText
P_1
$\cdots$
P_b
Ref. PT
RP_1
$\vdots$
RP_b
Input
RP_1 P_1 RP_1 P_1 $\cdots$ RP_b P_b RP_b P_b

Figure 4: IRC on block ciphers - composition of words

IRC also stores, for each $i \in \{1, \cdots, b'\}$, the byte K_i concatenated with RK_i, K_i and RK_i. Then, it executes the cipher by means of a single stream of 32-bit instructions operating independently on each byte, denoted by $IRC(\mathcal{E})$, to obtain, for each $i \in \{1, \cdots, b\}$, the byte C_i concatenated with RC_i, C_i and RC_i. Figure 5 shows the execution of $IRC(\mathcal{E})$.



Input
RP_1 P_1 RP_1 P_1 $\cdots$ RP_b P_b RP_b P_b
RK_1 K_1 RK_1 K_1
Key
$\vdots$
$RK_{b'}$ $K_{b'}$ $RK_{b'}$ $K_{b'}$
$IRC(\mathcal{E})$
Output
RC_1 C_1 RC_1 C_1 $\cdots$ RC_b C_b RC_b C_b

Figure 5: IRC on block ciphers - execution of $IRC(\mathcal{E})$

Finally, it makes comparisons involving each copy and the stored reference ciphertext. It then returns the ciphertext only if all the copies lead to the same result as illustrated in Figure 6.

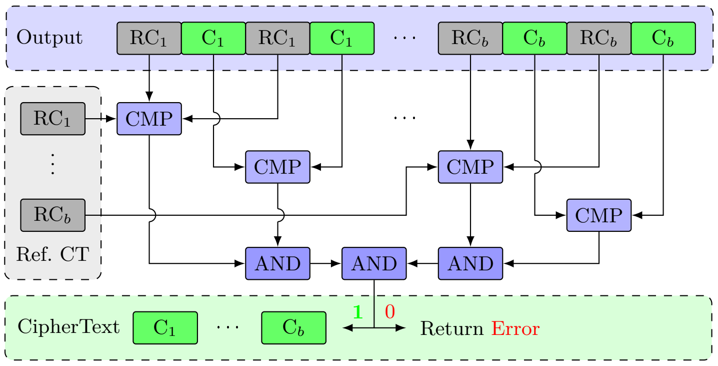


Figure 6: IRC on block ciphers - comparisons

Note that this last stage can be done with only 2 comparisons: first by comparing the last 16 bits of the word with the first 16 bits, then by comparing the stored reference byte with the obtained one.

3.3 IRC on Stream Ciphers

Modern stream ciphers are generally composed of two parts:

i. The first one is an initialization step: a function maps the secret key and a public initialization vector to an internal state S_0. Then, another function $\mathcal{E}$ is applied to S_0 to produce an pre-keystream-generation internal state S_1.

ii. The second one is the keystream generation: a function $\mathcal{I}$ is applied to S_1 which modifies its value and generates a byte of keystream (in the case of an 8-bit implementation). It produces as many keystream bytes as necessary to add to the plaintext in order to produce the ciphertext.

In this context, IRC consists in first applying to the initialization step the same method as previously described on block ciphers. The initial internal state $\text{IRC}(\text{RS}_0,\text{S}_0)$ is composed of the internal state S_0, a reference internal state RS_0 and their respective copies. RS_0 is obtained from a reference key and iv, which can be themselves easily generated (rather than stored). IRC applies $\mathcal{E}$ to $\text{IRC}(\text{RS}_0,\text{S}_0)$ by means of a single stream of instructions operating independently on each byte, operation that we shall denote by $\text{IRC}(\mathcal{E})$. It obtains an internal state $\text{IRC}(\text{RS}_1,\text{S}_1)$ composed of S_1, RS_1 and their copies. Figure 7 shows the initialization step protected by IRC in the case of a fault detection, where $S_{n,i}$ (resp. $\text{RS}_{n,i}$) denotes the i-th byte of the internal state S_n (resp. of RS_n). Moreover, each operation in Figure 7 is made for all $i \in \{1, \cdots, b\}$ with b the number of bytes of the internal state.

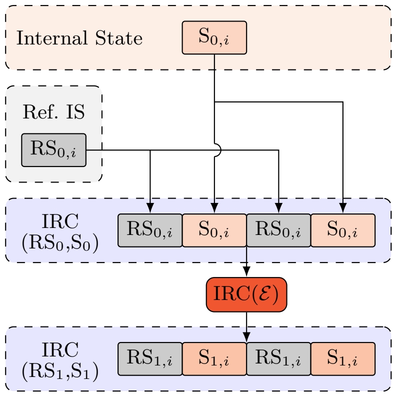


Figure 7: IRC on stream ciphers - initialization step

Then, each byte of the obtained internal state S_1 is stored in temporary registers and the function $\mathcal{I}$ is applied to $\text{IRC}(\text{RS}_1,\text{S}_1)$ also by means of a single stream of instructions operating independently on each byte, denoted by $\text{IRC}(\mathcal{I})$, to generate the first keystream word $\text{IRC}(\text{RK},\text{K}_1)$ and a new internal state

IRC(RS_2,S_2). IRC(RK,K_1) is composed of the first keystream byte K_1, the reference keystream byte RK and their copies. IRC compares the different copies of the keystream and compares the obtained reference keystream bytes with the stored one to ensure the correct value of K_1. It then stores its value instead of the reference keystream. Figure 8 shows the first keystream generation protected by IRC with the same notation as previously.

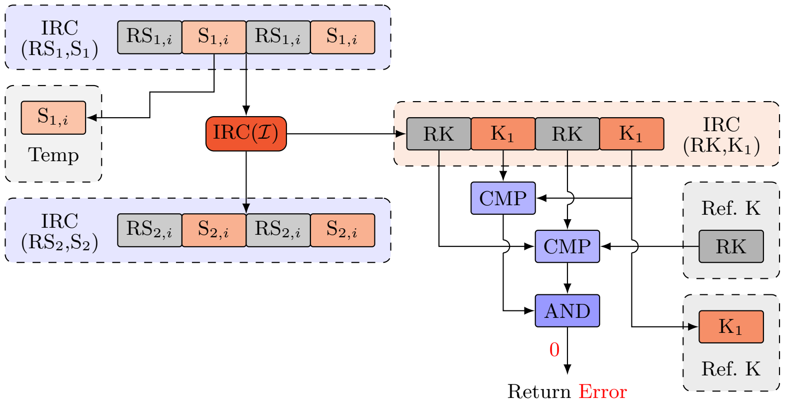


Figure 8: IRC on stream ciphers - first keystream byte generation

However, although the keystream byte is correct, it is possible that IRC(RS_2,S_2) contains a fault since only a part of the internal state is generally used to produce the keystream. Consequently, IRC makes also comparisons on IRC(RS_2,S_2) to ensure its correct value. Then, it replaces in each word the obtained reference internal state RS_2 by S_1 in order to protect each new keystream byte by the previous one as illustrated in Figure 9.

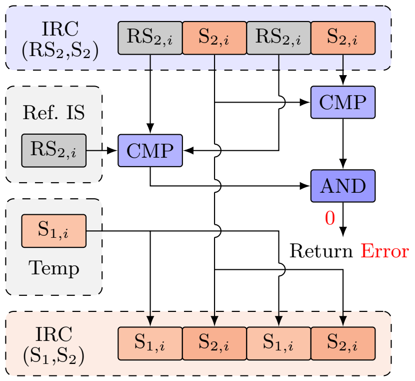


Figure 9: IRC on stream ciphers - comparisons

Each generated keystream word is hence composed of the actual keystream byte, the previous one and their respective copies. IRC can therefore make comparisons between the copies of the keystream bytes and with the previously stored keystream byte. Figure 10 shows the generation of the following keystream bytes protected by IRC also with the same notation as previously.

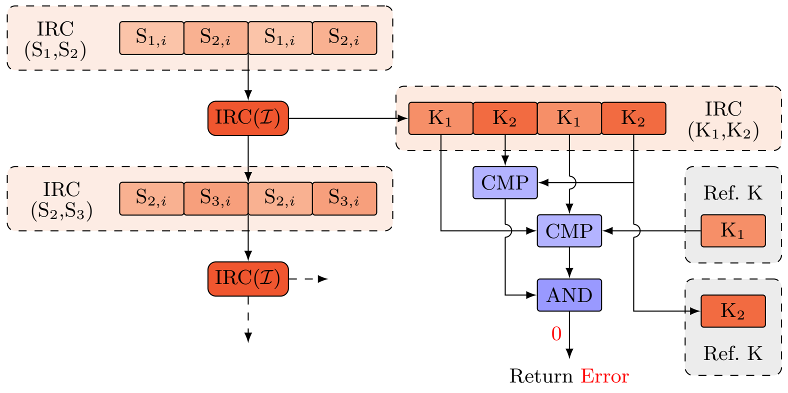


Figure 10: IRC on stream ciphers - generation of the following keystream bytes

To determine when the first byte of keystream K_1 is returned, two options are possible depending on the required security level:

i. Return K_1 after the comparisons in IRC(RK,K_1): we obtain a security similar to the case previously described for block ciphers.

ii. Return K_1 after the comparisons in IRC(K_1,K_2): we obtain an additional temporal redundancy since IRC generates K_1 twice consecutively and compares the generated values. Such an option has an overhead of one use of the iteration $\mathcal{I}$, which has generally a low cost.

It is also possible to use two temporal redundancies by replacing in each word of IRC(S_2,S_3) the first byte S_2 by S_1. In this case, IRC returns K_1 after the comparisons in IRC(K_2,K_3). Such an option has an overhead of two uses of the iteration $\mathcal{I}$ and needs an additional temporary register to store K_2 after the comparisons in IRC(K_1,K_2), which is not a high cost to pay for two additional temporal redundancies.

3.4 IRC Requirements

First, the reference block must be chosen such that each of its bytes is never, or as little as possible, equal to 0xFF (resp. 0x00) throughout the encryption or decryption, otherwise a set (resp. reset) of the whole state allows the attacker to obtain an undetected fault since it does not affect the reference block. It must also be chosen such that each of its bytes is always, or as much as possible, affected by the operations that the cipher performs, otherwise an instruction skip allows the attacker to obtain an undetected fault. Both properties must also be satisfied for each of its nibbles in the case of a majority vote. It is possible to obtain such properties by trying several random keys and plaintexts until satisfying both properties throughout the encryption or decryption, i.e. using the following code after each instruction applied to a byte B:

```
uint8_t temp;
temp =B;
...
/* Instruction applied to B */
...
if(B==0x00||B==0xff||B==temp){
    return 0;}
```

Note it is also possible to use two different reference blocks to satisfy both properties more easily. Then, we have to exclude ciphers which use conditional instructions depending on the value of an intermediate variable like the irregular clocking in MICKEY [6]. It is difficult to deploy IRC in this case since the reference block can have a different conditional value. It is nevertheless possible to replace the conditional instructions by making them dependent on both values. However, it is quite expensive since it requires additional operators, but such instructions are not recommended since they can allow to make safe-error attacks because of the conditional loop.

3.5 Implementation

Now we will describe how operators usually working on bytes can be translated to work on SIMD words. Since most lightweight ciphers are designed to have efficient 8-bit implementations, they mainly use bitwise operators like logical AND, OR, XOR, shift, rotation etc. and also nonlinear operators like modular addition and multiplication, etc. on bytes. Bitwise operators being intrinsically SIMD, they are straightforward to use for IRC. In the absence of a suitable SIMD instruction in the ISA (Instruction Set Architecture), IRC needs masks to implement nonlinear operators using few additional instructions, systematically operating as a whole on all the bytes of a 32-bit word in order to ensure the unicity of the instruction stream. Note that several lightweight block ciphers use S-boxes as nonlinear building-blocks. IRC thus uses the algebraic normal forms of the coordinates of the S-boxes which allow to implement them using only bitwise operators. Lastly, from a software engineering point of view, it is possible to benefit from the operator overloading capabilities of a number of languages (like C++) in order to protect a reference implementation without modifying its code. An example of the code in language C++ of some nonlinear operators for a 32-bit architecture is given in Appendix A. Therefore, we can deploy IRC directly from a parametric reference code of a cipher:

```
template<typename byte>
void Cipher(byte *state, byte *key, ···) { ··· }
```

without having to write a dedicated compiler. Indeed, we can simply instantiate the unprotected cipher from the instruction:

```
Cipher<uint8_t>(state, key, ···);
```

and the cipher protected by IRC from the instruction:

```
Cipher<IRC>(state, key, ···);
```

Additionally, most modern compilers (like GCC) provide (non standard) vector extensions which allow to write portable code and to automatically benefit from the SIMD instructions available in the targeted ISA when the appropriate compiler options are used.

4 Practical Implementations & Tests

In order to test IRC, we deployed it in the "fault detection mode" as previously described on two different 32-bit architectures: an ARM Cortex-M3 micro-controller on which we tested our own overloaded operators and an ARM Cortex-M4 micro-controller in order to exploit the SIMD instructions it provides. We used these two micro-controllers since they are quite representative of the off-the-shelf devices used for IoT. In both cases, we implemented and executed implementations of one representative lightweight block cipher PRIDE and of one representative stream cipher TRIVIUM with and without IRC. Then, we compared the resulting performances and we analyzed the resistance of IRC against fault attacks in practice. Note that in both cases no other countermeasure was implemented.

4.1 IRC on PRIDE

PRIDE is a block cipher introduced by Albrecht et al. [5] in 2014. It is one of the most efficient lightweight block cipher in terms of software implementation as shown by the performance comparisons given in [5,7]. The specifications of PRIDE are given in [5]. In Table 1 we compare the performances and footprints of the 8-bit reference implementation of PRIDE given in [2] and the same implementation protected by means of IRC with the reference plaintext 0x19cb6e3cc15d254f, the reference key 0xb8f653fa05f4f9c39889ce4bb9015865 and the corresponding reference ciphertext 0x8b8cc44779935cf2. The used reference block allows us to comply with the requirements (bytes never equal to 0x00 or 0xFF and always affected by the operations executed during the ciphering) on 97% of the implementation (with a full protection on the first and the last two rounds which generally are the areas targeted for DFA). It is possible to obtain a complete protection on the full implementation using in addition the reference plaintext 0x32c46c37168a7248, the reference key 0x485a895ac53577e7ffbd140564f5ca45 and the corresponding reference ciphertext 0x5b2569f55b45e69c. To be fair, we also include the performances of the 32-bit optimized (hence different) implementation of that algorithm given in [31] which of course achieves higher throughput on a 32-bit platform since it fully exploits the architecture.

Table 1: Performance comparisons for encrypting a 64-bit plaintext

	ARM Cortex-M3		ARM Cortex-M4	
	Time (cycles)	Size (bytes)	Time (cycles)	Size (bytes)
32-bit implementation	2852	464	2804	416
8-bit implementation	4370	558	4347	558
IRC implementation	6304	886	5108	636

PRIDE performs 4 additions on bytes per round. Thereby, the ARM Cortex-M4 micro-controller allows to obtain better performances thanks to the SIMD UADD8 ARM instruction as we can see in Table 1. However, PRIDE also performs several shifts on bytes, which need some masks and cannot be replaced by SIMD instructions. With this approach, we have as much as 4 executions of the cipher done in parallel (spatial redundancy) with two copies of the actual ciphered block and two copies of the reference block on one single core. If we were to achieve the same level of protection but without this approach, it would cost the same overhead as with 4 classical spatial copies of the data, i.e. it would require 4 times more memory. Thereby, on the one hand IRC allows to use spatial redundancy on this kind of device, and on the other hand even if we could use the basic redundancy methods, IRC would remain better since it allows to significantly decrease the required size, yet to the detriment of the execution time. Now we will provide the results we have obtained by testing some physical attacks on the ARM Cortex-M3 micro-controller implementing PRIDE protected by IRC.

Fault Attacks on IRC-protected PRIDE: In order to test IRC, we injected faults into the chip at different temporal locations using EM pulses as in [31] because with this approach we did not need to decapsulate the chip and we were able to inject faults at precise enough instants. The EM pulse used had a duration of 200ns, the applied voltage across the loop was varied by steps of 1 V between 180V and 219V and we injected 250 pulses by targeting the middle of the die. There are various possibilities to detect faults from IRC in a fault detection mode: thanks to the spatial redundancy, to the first reference block or to the second one. Thus, for each fault, we examined the value of the faulty ciphertext and how the fault has been detected. In total, we obtained 4823 faults from 10,000 EM injections. Among these faults 3107 were UART communication faults and 1716 were not. Then, IRC detected all the faults: 3444 were detected thanks to the spatial redundancy, 4805 thanks to the first reference block and all the 4823 thanks to the second one. Finally, we detected more faults thanks to the reference blocks: these faults actually made instruction skips and have thereby not been detected by the spatial redundancy. However, some faults have been only detected by the spatial redundancy: they have only affected one byte of the data. Therefore, IRC allowed us to fully thwart such a fault injection and the use of the spatial redundancy and of the reference block was necessary to detect all the faults.

4.2 IRC on TRIVIUM

TRIVIUM is a stream cipher introduced by De Cannière and Preneel [17], [18] in 2006. It belongs to the eSTREAM portfolio of recommended stream ciphers and has been specified as an ISO standard [1]. The specifications of TRIVIUM are given in [17]. In Table 2, we compare the performances and footprints of an 8-bit implementation of TRIVIUM detailed in Appendix B and the same implementation protected by means of IRC with the reference iv 0x73ad56fe566b227847f8, the reference key 0x4b752b672d363d93e7a3 and the reference internal state 0xd8b252aa20ecb9afb36cf7f4a42d1b1839fd86e63b68491fc3925 97c9477f22cd19562de. The used reference block allows us to comply with the requirements on 90% of the implementation. It is possible to obtain a complete protection on the full implementation using in addition the reference iv 0xe475605c64c6d25b5f18, the reference key 0x0bcc0de165fa80897046 and the reference internal state 0xc606f18d33cec788fa981538f612d0cc24e440e2c901e50bd380c19 20cb013fc70d05cf8. To be fair, we also include the performances of another 32-bit optimized implementation of that algorithm given in Appendix C.

Table 2: Performance comparisons to generate 32 bits of keystream

	ARM Cortex-M3		ARM Cortex-M4	
	Time (cycles)	Size (bytes)	Time (cycles)	Size (bytes)
32-bit implementation	168	292	166	292
8-bit implementation	852	296	820	296
IRC implementation	1172	448	1144	448

TRIVIUM uses only bitwise operators as well as several shifts which need some masks to be implemented in the case of IRC. Thereby, SIMD instructions are not used on the ARM Cortex-M4 micro-controller which implies that similar performances are obtained on both micro-controllers. Finally, although we cannot use basic redundancy methods, IRC allows to significantly decrease the required size compared to them to the detriment of the execution time which has an overhead larger than in the case of PRIDE since the 32-bit implementation of TRIVIUM is much faster than its 8-bit implementation (it requires 4 times less instructions to produce the same keystream length). Now we will describe as previously the results we obtained by testing some physical attacks on the ARM Cortex-M3 micro-controller implementing TRIVIUM protected by IRC.

Fault Attacks on IRC-protected TRIVIUM: In order to test IRC, we injected faults into the chip at different temporal locations using EM pulses with the same set-up as previously described. After the initialization process, we generated 10 bytes of keystream. In the case of TRIVIUM, there are also various possibilities to detect faults from IRC in a fault detection mode: thanks to the spatial redundancy, to the first temporal redundancy or to the second one. Thus, for each fault, we also examined the value of the faulty ciphertext and how the fault has been detected. In total, we obtained 3703 faults from 10,000 EM injections. Among these faults 3437 were UART faults and 266 were not. IRC detected all the faults: 3690 thanks to the spatial redundancy, 2491 thanks to the first temporal redundancy and 2915 thanks to the second one. This time, we detected more faults thanks to the spatial redundancy: these faults have only affected one byte of the data and have thereby not been detected by the temporal redundancy. We also observed that some faults have only been detected by the temporal redundancy: they have produced instruction skips. Therefore, IRC is highly efficient to thwart such a fault injection and, once again, the use of both spatial and temporal redundancies was necessary to detect all the faults.

4.3 What about side channel attacks?

Although IRC is not intended to provide resistance against side channel attacks, we also tested to perform experimentally such attacks on the ARM Cortex-M3 micro-controller implementing PRIDE protected by IRC.

Electromagnetic radiations analysis. First, to probe further capabilities of IRC, we performed a electromagnetic radiations analysis. The aim is to identify the operations made by the cipher. Figure 11 shows the obtained curves which allow us to identify all the steps of PRIDE despite IRC.

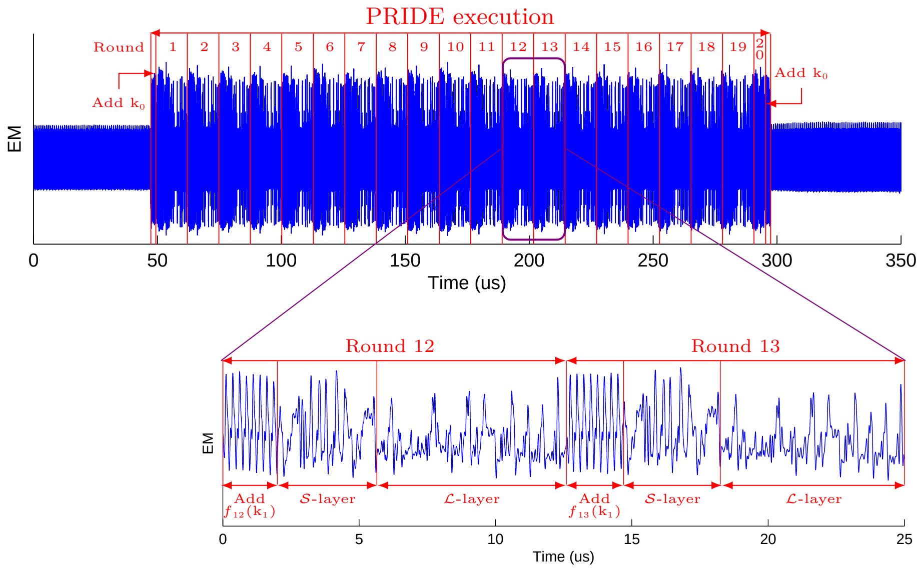


Figure 11: Electromagnetic radiations analysis of PRIDE

Correlation electromagnetic analysis. Then, we implemented the correlation electromagnetic analysis (CEMA) described in [2]. PRIDE was executed for 1000 random plaintexts with the fixed key 0xa371b246f90cf582e417d148e239ca5d. The chosen reference block was the last test vector given in [5]. The last substitution layer was targeted for the data acquisition and EM traces were captured with 7500 points per encryption. Thereafter, the matrix of obtained traces is denoted by T.

$$T = \begin{bmatrix} T_0 \\ T_2 \\ \vdots \\ T_{7499} \end{bmatrix} = \begin{bmatrix} t_{0,0} & t_{1,1} & \cdots & t_{0,999} \\ t_{2,0} & t_{2,1} & \cdots & t_{2,999} \\ \vdots & \vdots & \ddots & \vdots \\ t_{7499,1} & t_{7499,2} & \cdots & t_{7499,999} \end{bmatrix}. \quad (1)$$

To recover each byte $\mathcal{P}(k_0)_i$ for $i \in \{0, \cdots, 7\}$, we first derive the estimation matrices E^i from the Hamming weight of each ciphertext C_j for $j \in \{0, \cdots, 999\}$ XORed to each key hypothesis $H_K \in \{0, \cdots, 255\}$.

$$E^i = \begin{bmatrix} E_0^i \\ E_1^i \\ \vdots \\ E_{255}^i \end{bmatrix} = \begin{bmatrix} e_{0,0}^i & e_{0,1}^i & \cdots & e_{0,999}^i \\ e_{1,0}^i & e_{1,1}^i & \cdots & e_{1,999}^i \\ \vdots & \vdots & \ddots & \vdots \\ e_{255,0}^i & e_{255,1}^i & \cdots & e_{255,999}^i \end{bmatrix} \tag{2}$$

where $e^i_{H_K,j} = HW(C_{j,i} \oplus H_K)$. Then, we perform a classical CEMA attack (also called *Vertical*) by computing the correlation coefficient matrices P^i, for $i \in \{0, \cdots, 7\}$, from the Pearson correlation coefficient between E^i and T, namely

$$P^i = \begin{bmatrix} P_0^i \\ P_1^i \\ \vdots \\ P_{n-1}^i \end{bmatrix} = \begin{bmatrix} \rho_{0,0}^i & \rho_{0,1}^i & \cdots & \rho_{0,255}^i \\ \rho_{1,0}^i & \rho_{1,1}^i & \cdots & \rho_{1,255}^i \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n-1,0}^i & \rho_{n-1,1}^i & \cdots & \rho_{n-1,255}^i \end{bmatrix} \tag{3}$$

where $\rho^i_{t,H_K} = \mathsf{Corr}(T_t, E^i_{H_K})$. Figure 12 shows the plot corresponding to P^0 on the interval 1100 to 2100 points in abscissa. From Figure 12, we can clearly distinguish the highest value which was 0xf3. Finally, we retrieve the correct value of $\mathcal{P}(k_0) = $ 0xf3f721cb1c882658 from all P^i.

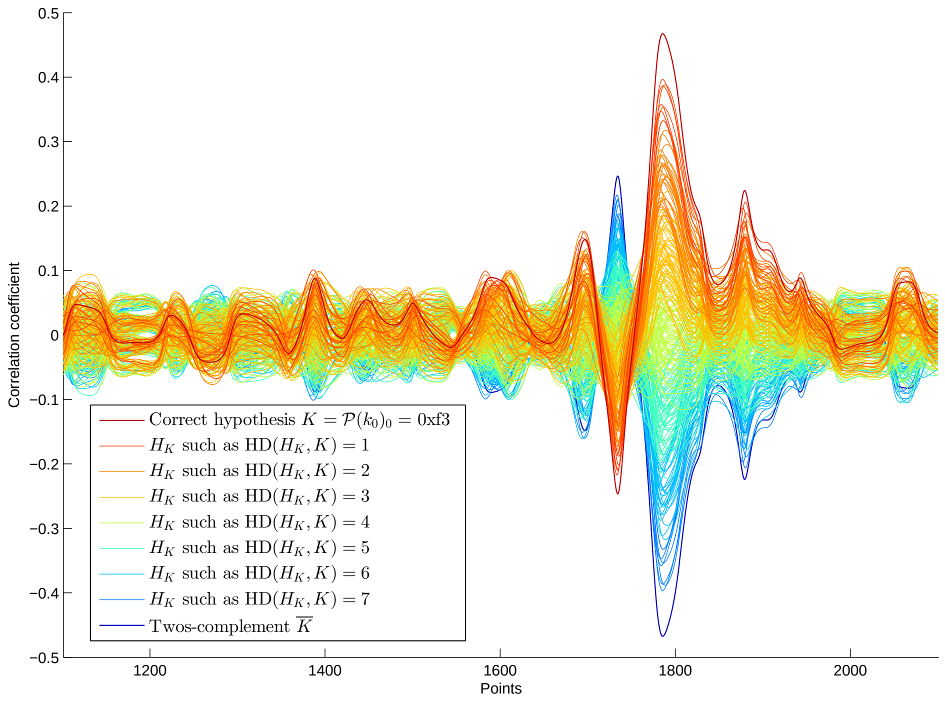


Figure 12: Key recovery of $\mathcal{P}(k_0)_0$

We also conducted the previously CEMA with the same complexity by changing the reference block at each execution without knowledge or assumption about it. It added only a negligible noise which did not prevent the attack.

The described CEMA targets the STRB ARM instruction as explained in [2]. Therefore, according to our results, we believe that this instruction is done byte by byte, which explains why IRC did not increase the complexity to the CEMA. It is possible that IRC increases the complexity of other side-channel attacks but we believe that it does not provide real protection against them. However, it provides a very high security against fault attacks: from our experience, we believe that the kind of faults required to thwart IRC is nowadays extremely difficult to obtain for an attacker even with a laser injection which is currently the most precise injection mean.

5 Generalization of IRC

The principle behind IRC can be generalised to architectures with a different data path width. For example, IRC can make a fault detection or a fault correction using two different blocks of data on a 64-bit architecture if we organize each word as shown in Figure 13.

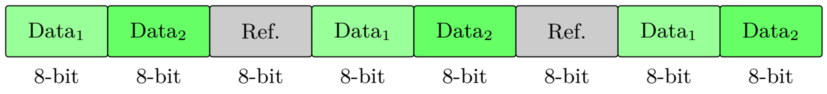


Figure 13: IRC protecting two blocks of data on 64-bit words

Indeed, IRC executes the cipher by means of a single stream of 64-bit instructions systematically operating independently on each byte. Then, at the end of encryption or decryption, IRC can make a fault detection (or a fault correction) involving each copy and the stored reference ciphertext to detect (or mask) all the faults except those having the same impact on the three (or on two or more) copies of the data without affecting the reference blocks.

IRC can also make only a fault detection using three different blocks of data with an organization as depicted in Figure 14.

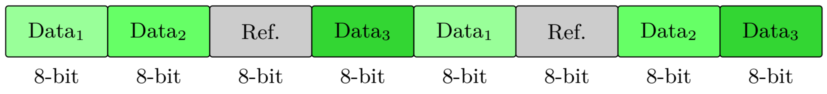


Figure 14: IRC protecting three blocks of data on 64-bit words

It is also possible to increase the security level by taking only one data block as input but we believe that it is not necessary in most cases.

More generally, IRC can turn any m-bit implementation on an ℓm-bit architecture applying the same concept provided that $\ell \geqslant 3$ in order to include at least two copies of the data and one reference block.

6 Conclusion & Discussion

In this paper we describe the Internal Redundancy Countermeasure and validate its efficiency in thwarting fault attacks against a representative lightweight block cipher PRIDE and a stream cipher TRIVIUM running on two different 32-bit architectures: an ARM Cortex-M3 micro-controller on which we tested our own overloaded operators and an ARM Cortex-M4 micro-controller where the available SIMD instructions enhanced the efficiency of the countermeasure. IRC consists in using the same bitwise operators and needs masks to implement nonlinear operators on bytes using few additional instructions systematically operating as a whole on the bytes of a word in order to ensure the uniqueness of the instruction stream. We also show that it is also possible to apply IRC on a 64-bit architecture for which we believe it is more interesting to include two different data blocks within each input word. To our best knowledge, the one work that comes close to what we propose has been reported in [39]. However the IIR [39] uses a bit-sliced implementation of the cipher which is less efficient than the IRC we propose: IIR is applied to 15 blocks of data, which increases the latency since we have to wait until the end of the encryption of the 15 blocks, which is a major drawback for lightweight applications. Moreover, IRC stores each block of reference between two blocks of data in order to avoid faults on consecutive bits, thus providing better security than IIR.

The overhead of IRC depends on the targeted cipher as illustrated in this paper. However these impacts have to be leveraged with the high fault coverage achieved since it detected all the faults injected by EM pulses. Moreover this scheme has been shown to work on a widely spread processor core and hence does not need any hardware modification with respect to the already existing processors embedding SIMD instructions. By illustrating the feasibility and efficiency of this approach, we hope to encourage chip manufacturers to integrate dedicated SIMD instructions to help tackle such a complex issue as protection against fault attacks. One further step in this research work will to be investigate how to efficiently enhance this scheme to provide resistance against side channel attacks.

A Class IRC in C++ language

In order to test IRC, we created a class in C++ language which implements it in a completely generic way — i.e. independently of the cipher. The class provides all the overloaded operators that one wants to work on bytes such as:

```
class IRC{
    private:union{
        uint8_t bytes[4];
        uint32_t word;
    }state;
    public:
    IRC(){state.word=0x0;}
    IRC(uint8_t data, uint8_tknownPT){
        state.bytes[0]=knownPT;
        state.bytes[1]=data;
        state.bytes[2]=knownPT;
        state.bytes[3]=data;}
    IRC &operator>>=(const uint8_tshift){
        static const uint32_t mask[9] = {
            0xffffffff,0x7f7f7f7f,0x3f3f3f3f,
            0x1f1f1f1f,0x0f0f0f0f,0x07070707,
            0x03030303,0x01010101,0x00000000};
        assert (shift>=0 &&shift<=8);
        state.word>>=shift;
        state.word&=mask[shift];
        return *this;}
    IRC operator>>(const int shift)const{
        IRC res(*this);
        res>>=shift;
        return res;}
    IRC &operator+=(const IRC &other){
        uint32_t result=state.word&0x7f7f7f7f;
        result+=other.state.word&0x7f7f7f7f;
        uint32_t carry=result&0x80808080;
        result&=0x7f7f7f7f;
        carry+=state.word&0x80808080;
        carry+=other.state.word&0x80808080;
        carry&=0x80808080;
        state.word=result|carry;
        return *this;}
    IRC &operator+=(const uint8_tbyte){
    IRC tmp(byte,byte);
    (*this)+=tmp;
    return *this;}
    ...
}
```

B TRIVIUM 8-bit implementation

In this section, we provide the 8-bit software implementation of TRIVIUM that we used in this paper. First, we introduce the 8-bit keystream generation from the 36 bytes of the internal state by the function NextByte:

```
static const uint8_t T[72] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33, 34, 35, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35};
uint8_t k=0;

uint8_t NextByte(uint8_t s[36]){
    uint8_t t1,t2,t3,z;
    // Step 1: Update of the temporary registers and the output
    // t1 ← s58···s65 ⊕ s85···s92
    t1=(s[T[7+k]]<<2|s[T[8+k]]>>6)^(s[T[10+k]]<<5|s[T[11+k]]>>3);
    // t2 ← s154···s161 ⊕ s169···s176
    t2=(s[T[19+k]]<<2|s[T[20+k]]>>6)^(s[T[21+k]]<<1|s[T[22+k]]>>7);
    // t3 ← s235···s242 ⊕ s280···s287
    t3=(s[T[29+k]]<<3|s[T[30+k]]>>5)^s[T[35+k]];
    // 8−bit keystream calculation
    z=t1^t2^t3;
    // t1 ← t1 ⊕ (s83···s90 & s84···s91) ⊕ s163···s170
    t1^=(s[T[10+k]]<<3|s[T[11+k]]>>5)&(s[T[10+k]]<<4|s[T[11+k]]>>4);
    t1^=(s[T[20+k]]<<3|s[T[21+k]]>>5);
    // t2 ← t2 ⊕ (s167···s174 & s168···s175) ⊕ s256···s263.
    t2^=(s[T[20+k]]<<7|s[T[21+k]]>>1)&s[T[21+k]]^s[T[32+k]];
    // t3 ← t3 ⊕ (s278···s285 & s279···s286) ⊕ s61···s68
    t3^=(s[T[34+k]]<<6|s[T[35+k]]>>2)&(s[T[34+k]]<<7|s[T[35+k]]>>1);
    t3^=(s[T[7+k]]<<5|s[T[8+k]]>>3);
    // Step 2: Shift of 1 byte to right of the internal state (update k)
    k+=35;
    k=T[k];
    // Step 3: Update of 3 bytes of the internal state
    // s0···s7 ← t3
    s[T[0+k]]=t3;
    // s93···s100 ← t1
    s[T[11+k]]=(s[T[11+k]]&0xf8)|(t1>>5);
    s[T[12+k]]=(t1<<3)|(s[T[12+k]]&0x07);
    // s177···s184 ← t2
    s[T[22+k]]=(s[T[22+k]]&0x80)|(t2>>1);
    s[T[23+k]]=(t2<<7)|(s[T[23+k]]&0x7f);
    return z;
}
```

Then, the function Init loads the key and the initialization vector (iv) into the initial state and allows to perform the 144 warm-up rounds (4·288/8) needed before generating keystream bytes. In case of IRC, it is necessary to compute one additional NextByte after loading the inputs and then to load one more time the inputs on two bytes without modifying the others.

```
void Init(uint8_t s[36], const uint8_t key[10], const uint8_t iv[10]){
    uint8_t i;
    for(i=0;i<36;i++){
        s[i]=0x0;}
    s[35]=0x07;
    // Insert the key at position 0
    for(i=0;i<10;i++){
        s[i]=key[i];}
    // Insert the iv at position 93
    s[11]=iv[0]>>5;
    for(i=0;i<9;i++){
        s[i+12]=(iv[i]<<3)|(iv[i+1]>>5);}
    s[21]=iv[9]<<3;

    // 144 warm-up rounds.
    for(i=0;i<144;i++){
        NextByte(s);}
}
```

C TRIVIUM 32-bit implementation

In this section, we provide the 32-bit software implementation of TRIVIUM that we used in this paper. First, we introduce the 32-bit keystream generation from the 9 words of the internal state by the function NextWord:

```
static const uint8_t T[18] = {0, 1, 2, 3, 4, 5, 6, 7, 8,
0, 1, 2, 3, 4, 5, 6, 7, 8};
uint8_t k=0;

uint32_t NextWord(uint32_t S[9]){
    uint8_t i;
    uint32_t t1,t2,t3,z;
    // Step 1: Update of the temporary registers and the output
    // t1 ← s34···s65 ⊕ s61···s92
    t1=(S[T[1+k]]<<2|S[T[2+k]]>>30)^(S[T[1+k]]<<28|S[T[2+k]]>>4);
    // t2 ← s130···s161 ⊕ s145···s176
    t2=(S[T[4+k]]<<2|S[T[5+k]]>>30)^(S[T[4+k]]<<17|S[T[5+k]]>>15);
    // t3 ← s211···s242 ⊕ s256···s287
    t3=(S[T[6+k]]<<19|S[T[7+k]]>>13)^S[T[8+k]];
    // 32-bit keystream calculation
    z=t1^t2^t3;
    // t1 ← t1 ⊕ (s59···s90&s60···s91) ⊕ s139···s170
    t1^=(S[T[1+k]]<<27|S[T[2+k]]>>5)&(S[T[1+k]]<<28|S[T[2+k]]>>4);
    t1^=(S[T[4+k]]<<11|S[T[5+k]]>>21);
    // t2 ← t2 ⊕ (s143···s174&s144···s175) ⊕ s232···s263
    t2^=(S[T[4+k]]<<15|S[T[5+k]]>>17)&(S[T[4+k]]<<16|S[T[5+k]]>>16);
    t2^=(S[T[7+k]]<<8|S[T[8+k]]>>24);
    // t3 ← t3 ⊕ (s254···s285&s255···s286) ⊕ s37···s68
    t3^=(S[T[7+k]]<<30|S[T[8+k]]>>2)&(S[T[7+k]]<<31|S[T[8+k]]>>1);
    t3^=(S[T[1+k]]<<5|S[T[2+k]]>>27);
    // Step 2: Shift of 1 word to right of the internal state (update k)
    k+=8;
    k=T[k];
    // Step 3: Update of 3 words of the internal state
    // s0···s31 ← t3
    S[T[0+k]]=t3;
    // s93···s124 ← t1.
    S[T[2+k]]=(S[T[2+k]]&0xfffffff8)|(t1>>29);
    S[T[3+k]]=(S[T[3+k]]&0x00000007)|(t1<<3);
    // s177···s208 ← t2
    S[T[5+k]]=(S[T[5+k]]&0xffff8000)|(t2>>17);
    S[T[6+k]]=(S[T[6+k]]&0x00007fff)|(t2<<15);
    return z;
}
```

Then, the function Init loads the key and the iv into the initial state and allows to perform the 36 warm-up rounds (4·288/32) needed before generating keystream words. Note that the key and the iv must be completed by 16 zeros.

```
void Init(uint32_t S[9], const uint32_t key[3],const uint32_t iv[3]){
    uint8_t i;
    for(i=0;i<9;i++){
        S[i]=0x0;}
    S[8]=0x07;
    // Insert the key at position 0
    for(i=0;i<3;i++){
        S[i]=key[i];}
    // Insert the iv at position 93
    S[2]=iv[0]>>29;
    for(i=0;i<2;i++){
        S[i+3]=(iv[i]<<3|iv[i+1]>>29);}
    S[5]=iv[2]<<3;

    // 36 warm-up rounds.
    for(i=0;i<36;i++){
        NextWord(s);}
}
```

References

1. 29192-3:2012, I.: Information technology – security techniques – lightweight cryptography – part 3: Stream ciphers (2012)
2. Adomnicai, A., Lac, B., Canteaut, A., Fournier, J.J., Masson, L., Sirdey, R., Tria, A.: On the importance of considering physical attacks when implementing lightweight cryptography. In: NIST Lightweight Cryptography Workshop 2016. Gaithersburg, Maryland (October 2016)
3. Agoyan, M., Dutertre, J., Naccache, D., Robisson, B., Tria, A.: When clocks fail: On critical paths and clock faults. In: Gollmann, D., Lanet, J., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 182–193. Springer, Berlin, Germany, Passau, Germany (April 14-16, 2010)
4. Akkar, M.L., Giraud, C.: An Implementation of DES and AES, Secure against Some Attacks. In: Çetin Koç, Naccache, D., Paar, C. (eds.) Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01). LNCS, vol. 2162, pp. 309–318. Springer-Verlag, Paris, France (2001)
5. Albrecht, M.R., Driessen, B., Kavun, E.B., Leander, G., Paar, C., Yalçin, T.: Block ciphers - focus on the linear layer (feat. PRIDE). In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 57–76. Springer, Berlin, Germany, Santa Barbara, CA, USA (Aug 17–21, 2014)
6. Babbage, S., Dodd, M.: The mickey stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs: The eSTREAM Finalists, pp. 191–209. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-68351-3_15
7. Baysal, A., Sahin, S.: Roadrunner: A small and fast bitslice block cipher for low cost 8-bit processors. In: Güneysu, T., Leander, G., Moradi, A. (eds.) LightSec 2015. LNCS, vol. 9065, pp. 58–76. Springer, Berlin, Germany, Bochum, Germany (September 10-11, 2015)
8. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: SIMON and SPECK: Block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/585 (2015), http://eprint.iacr.org/2015/585
9. Benot, O.: Fault attack. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, pp. 452–453. Springer US, Boston, MA (2011), http://dx.doi.org/10.1007/978-1-4419-5906-5_505
10. Bertoni, G., Breveglieri, L., Koren, I., Maistri, P., Piuri, V.: A parity code based fault detection for an implementation of the advanced encryption standard. In: DFT '02: Proceedings of the 17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems. pp. 51–59. IEEE Computer Society, Washington, DC, USA (2002)
11. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO'97. LNCS, vol. 1294, pp. 513–525. Springer, Berlin, Germany, Santa Barbara, CA, USA (Aug 17–21, 1997)
12. Bilgin, B., Bogdanov, A., Knezevic, M., Mendel, F., Wang, Q.: Fides: Lightweight authenticated cipher with side-channel resistance for constrained hardware. In: Bertoni, G., Coron, J. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2013. LNCS, vol. 8086, pp. 142–158. Springer, Heidelberg, Germany (2013), http://doc.utwente.nl/89342/
13. Blömer, J., Seifert, J.P.: Fault based cryptanalysis of the advanced encryption standard (AES). In: Wright, R. (ed.) FC 2003. LNCS, vol. 2742, pp. 162–181. Springer, Berlin, Germany, Guadeloupe, French West Indies (Jan 27–30, 2003)

14. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Berlin, Germany, Vienna, Austria (Sep 10–13, 2007)
15. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 37–51. Springer, Berlin, Germany, Konstanz, Germany (May 11–15, 1997)
16. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knežević, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Berlin, Germany, Beijing, China (Dec 2–6, 2012)
17. De Cannière, C.: Trivium: A stream cipher construction inspired by block cipher design principles. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) Information Security: 9th International Conference, ISC 2006. LNCS, vol. 4176, pp. 171–186. Springer (2006), http://dx.doi.org/10.1007/11836810_13
18. De Cannière, C., Preneel, B.: Trivium. In: Robshaw, M.J.B., Billet, O. (eds.) New Stream Cipher Designs - The eSTREAM Finalists, LNCS, vol. 4986, pp. 244–266. Springer (2008)
19. Dehbaoui, A., Dutertre, J., Robisson, B., Tria, A.: Electromagnetic transient faults injection on a hardware and a software implementations of AES. In: Bertoni, G., Gierlichs, B. (eds.) FDTC 2012. pp. 7–15. IEEE Computer Society, Leuven, Belgium (September 9, 2012)
20. Dobraunig, C., Eichlseder, M., Korak, T., Lomné, V., Mendel, F.: Statistical fault attacks on nonce-based authenticated encryption schemes. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 369–395 (2016)
21. El-Baze, D., Rigaud, J.B., Maurine, P.: A fully-digital EM pulse detector. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 439–444 (March 2016)
22. Gérard, B., Grosso, V., Naya-Plasencia, M., Standaert, F.: Block ciphers that are easier to mask: How far can we go? In: Bertoni, G., Coron, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2013. LNCS, vol. 8086, pp. 383–399. Springer (2013)
23. Gong, Z., Nikova, S., Law, Y.W.: KLEIN: A new family of lightweight block ciphers. In: Juels, A., Paar, C. (eds.) RFID. Security and Privacy - RFIDSec 2011. LNCS, vol. 7055, pp. 1–18. Springer (2012)
24. Grosso, V., Leurent, G., Standaert, F.X., Varici, K.: LS-designs: Bitslice encryption for efficient masked software implementations. In: Cid, C., Rechberger, C. (eds.) FSE 2014. LNCS, vol. 8540, pp. 18–37. Springer, Berlin, Germany, London, UK (Mar 3–5, 2015)
25. Hell, M., Johansson, T., Meier, W.: Grain: a stream cipher for constrained environments. International Journal of Wireless and Mobile Computing 2(1), 86–93 (2007), http://www.inderscienceonline.com/doi/abs/10.1504/IJWMC.2007.013798
26. Karaklajic, D., Schmidt, J.M., Verbauwhede, I.: Hardware designer's guide to fault attacks. IEEE Trans. Very Large Scale Integr. Syst. 21(12), 2295–2306 (Dec 2013), http://dx.doi.org/10.1109/TVLSI.2012.2231707

27. Karri, R., Kuznetsov, G., Goessel, M.: Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers. In: Walter, C., al. (eds.) Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03). pp. 113–124. No. 2779 in LNCS, Springer-Verlag, Cologne, Germany (September 7-10 2003)
28. Kim, C.H., Shin, J.H., Quisquater, J.J., Lee, P.J.: Safe-Error Attack on SPA-FA Resistant Exponentiations Using a HW Modular Multiplier. In: Nam, K.H., Rhee, G. (eds.) Information Security and Cryptology - ICISC 2007. LNCS, vol. 4817, pp. 273–281. Springer (2007), http://dx.doi.org/10.1007/978-3-540-76788-6_22
29. Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and other systems. In: Proceedings of Advances in Cryptology (CRYPTO'96). pp. 104–113. LNCS, Springer-Verlag (1996)
30. Koeune, F., Standaert, F.: A tutorial on physical security and side-channel attacks. In: Foundations of Security Analysis and Design III, FOSAD 2004/2005 Tutorial Lectures. LNCS, vol. 3655, pp. 78–108. Springer, Berlin, Germany (2005)
31. Lac, B., Beunardeau, M., Canteaut, A., Fournier, J.J., Sirdey, R.: A First DFA on PRIDE: from Theory to Practice. In: International Conference on Risks and Security of Internet and Systems - CRiSIS 2016. LNCS, vol. 10158, pp. 214–238. Springer (September 2016)
32. Lac, B., Canteaut, A., Fournier, J.J., Sirdey, R.: DFA on LS-Designs with a Practical Implementation on SCREAM. In: Constructive Side-Channel Analysis and Secure Design - COSADE 2017. LNCS, Springer (2017), to appear
33. Lashermes, R., Fournier, J., Goubin, L.: Inverting the final exponentiation of Tate pairings on ordinary elliptic curves using faults. In: Bertoni, G., Coron, J.S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 365–382. Springer, Berlin, Germany, Santa Barbara, California, US (Aug 20–23, 2013)
34. Li, H., Markettos, T., Moore, S.: Security Evaluation Against Electromagnetic Analysis at Design Time. In: B.Sunar, J.Rao (eds.) Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05). pp. 280–292. No. 3659 in LNCS, Springer-Verlag, Edinburg, Scotland (August 29th - September 1st 2005)
35. Messerges, T., Dabbish, E., Sloan, R.: Power Analyis Attacks of Modular Exponentiation in Smartcards. In: Ç.K. Koç, Paar, C. (eds.) Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999). pp. 144–157. No. 1717 in LNCS, Springer-Verlag (1999)
36. Mestiri, H., Benhadjyoussef, N., Machhout, M., Tourki, R.: A robust fault detection scheme for the advanced encryption standard. IJCNIS 5(6), 49–55 (2013)
37. Mohamed, M.S.E., Bulygin, S., Buchmann, J.A.: Using SAT solving to improve differential fault analysis of trivium. In: Kim, T., Adeli, H., Robles, R.J., Balitanas, M.O. (eds.) ISA 2011. Communications in Computer and Information Science, vol. 200, pp. 62–71. Springer, Berlin, Germany, Brno, Czech Republic (August 15-17, 2011)
38. Moore, S., Anderson, R., Cunningham, P., Mullins, R., Taylor, G.: Improving Smart Card Security using Self-timed Circuits. In: Proceedings of 8th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC' 02. vol. IEEE, pp. 23–58 (2002)
39. Patrick, C., Yuce, B., Ghalaty, N., Schaumont, P.: Lightweight fault attack resistance in software using intra-instruction redundancy. In: Selected Areas in Cryptography - SAC 2016. LNCS, Springer (2016), to appear

40. Piret, G., Roche, T., Carlet, C.: PICARO – a block cipher allowing efficient higher-order side-channel resistance. In: Bao, F., Samarati, P., Zhou, J. (eds.) Applied Cryptography and Network Security – ACNS 2012. LNCS, vol. 7341, pp. 311–328. Springer, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31284-7_19
41. Pornin, T.: Implantation et optimisation des primitives cryptographiques. Ph.D. thesis, Université Paris 7 (2001)
42. Sakiyama, K., Li, Y., Iwamoto, M., Ohta, K.: Information-theoretic approach to optimal differential fault analysis. IEEE Transactions on Information Forensics and Security 7(1), 109–120 (2012)
43. Skorobogatov, S.: Semi-invasive attacks - A new approach to hardware security analysis. Technical Report 630, University of Cambridge (April 2005)
44. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Kaliski Jr., B.S., Koç, Çetin Kaya., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Berlin, Germany, Redwood Shores, California, USA (Aug 13–15, 2003)
45. Song, L., Hu, L.: Differential fault attack on the PRINCE block cipher. Cryptology ePrint Archive, Report 2013/043 (2013), http://eprint.iacr.org/2013/043
46. Tupsamudre, H., Bisht, S., Mukhopadhyay, D.: Differential fault analysis on the families of SIMON and SPECK ciphers. Cryptology ePrint Archive, Report 2014/267 (2014), http://eprint.iacr.org/2014/267
47. van Woudenberg, J.G.J., Witteman, M.F., Menarini, F.: Practical optical fault injection on secure microcontrollers. In: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 91–99 (Sept 2011)
48. Zhao, X., Wang, T., Guo, S.: Improved side channel cube attacks on PRESENT. Cryptology ePrint Archive, Report 2011/165 (2011), http://eprint.iacr.org/2011/165