The Other Side of The Coin: Analyzing Software Encoding Schemes Against Fault Injection Attacks

Jakub Breier[1], Dirmanto Jap[1,2] and Shivam Bhasin[1]
[1]Physical Analysis and Cryptographic Engineering
[2]School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
Email: {jbreier, djap, sbhasin}@ntu.edu.sg

***Abstract*—The versatility and cost of embedded systems have made it ubiquitous. Such wide-application exposes an embedded system to a variety of physical threats like side-channel attacks (SCA) and fault attacks (FA). Recently, a couple of software encoding schemes were proposed as a protection against SCA. These protection schemes are based on dual-rail precharge logic (DPL), previously shown resistant to both SCA and FA. In this paper, we analyze the previously proposed software encoding schemes against FA. Our results show that software encoding offers only limited resistance to FA. Finally, improvement to software-encoding schemes is improved. With this improvement, software encoding can serve as a common SCA and FA countermeasure with an exploitable fault probability as low as 0.0048.**

I. Introduction

Security has emerged as a key parameter in new technologies, especially Internet of Things (IoT). Each node of an IoT is a computing device and does suffer from usual security threats, from operating system viruses to network vulnerabilities. With such vicinity to the adversary, threats like side-channel attacks (SCA) and fault attacks (FA) have become a reality for IoT. Thus, designers tend to deploy countermeasures against these novel threats.

Side-channel countermeasures roughly fall into two wide categories – masking [1] and hiding [2]. Masking countermeasures are generally applied at the algorithmic level. It depends on a random mask to prevent leakage of sensitive data.

On the other hand, hiding countermeasures try to hide the data dependent leakage in order to remove the basis of SCA. The most common hiding countermeasure is a dual-rail precharge logic (DPL [2]). DPL is a circuit-level countermeasure which removes data-dependent leakage by introducing a generated False (F) rail to compensate the activity of the original True (T) rail. It also introduces a notion of *Precharge* phase as a spacer between every two *Evaluation* phases such as, where valid algorithmic data is computed and propagated. The two-phase operation with a dual-rail structure (theoretically) ensures constant activity and is therefore free from any exploitable data-dependent leakage.

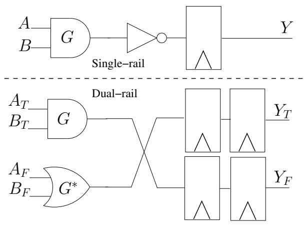
Although DPL was initially introduced as a side-channel countermeasure, Selmane et al. [3] showed in 2009 that DPL possess properties that resist FA naturally. Fault resistance is an obvious advantage of DPL countermeasures over masking. For software implementations, masking is an obvious choice as it only needs modification of the algorithm. However, some researchers have recently proposed software-oriented DPL countermeasures. The notion of software DPL was first introduced in [4], however a complete implementation was missing. In [5], Rauzy et al. extended this idea to propose a first DPL implementation of PRESENT cipher in software. Moreover, they provided a formal proof of security for the proposed DPL implementation. Another related work by Chen el al. [6] presents a different software-oriented hiding countermeasure called 'Balanced Encoding', based on [4]. All these countermeasures were proposed to counter SCA and were shown to provide a theoretical side-channel resistance.

From an attacker's perspective, one will try the easiest and most efficient attack. Therefore application of FA on side-channel protection cannot be ignored. If we can work towards a generic countermeasure, which to an extent, resists both SCA and FA, the solution will be much more appealing for real-life applications.

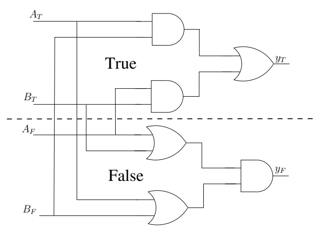
In this paper, we analyze previously proposed software variants of DPL against FA. Since DPL is already known to resist majority of faults in a hardware implementation, we want to analyze it if the software DPL still holds that property. Precisely, we target the implementations proposed in [6] and [5], when implemented on a simple microcontroller. Our analysis follows a two-step methodology. First, we develop a fault simulator, which injects fault into a given software code, with common fault models. Secondly, we perform practical fault attacks on an AVR platform, using laser fault injection, to validate our simulation results.

Based on our analysis, we propose an improved version of the implementation introduced in [5], reducing possibility of a fault attack to minimum. Overhead of our countermeasure in terms of AVR microcontroller clock cycles is $\approx 72.3\%$ compared to original implementation. Probability of a successful fault attack was reduced to 0.0048 and after conducting a practical laser fault analysis we were not able to find any exploitable fault.

The rest of the paper is organized as follows: Sec. II provides some general background on DPL.The fault simulation methodology is discussed in Sec. III, together with simulation results. Practical validation of simulated faults on an 8-bit AVR microcontroller, using laser injection technique, is explained in Sec. IV. Then, the discussion on the faults obtained from



(a) Basic Elements of WDDL [3].



(b) WDDL XOR Gate.

both simulations and experiments is given in Sec. V. Next, we propose an improved countermeasures in Sec. VI. Finally, we draw conclusions in Sec. VII.

II. General Background

In this section, we give a general background on dual-rail precharge logic, its application on software and previous work.

A. Dual-Rail Precharge Logic

Dual-Rail Precharge Logic (DPL) is countermeasure to SCA. DPL adds a complementary logic to balance the sensitive activity of a target. In DPL, every sensitive bit x is coded as (x_T, x_F), where $x_T = x$ and $x_F = \bar{x}$. The couple (x_T, x_F) alternates in two phases: *Precharge* i.e. propagating $(0,0)$ **NULL spacer**[1], and *Evaluation* i.e. propagating $(1,0)$ or $(0,1)$ **VALID data**. The duplication of sensitive data along with a two-phase operation (theoretically) ensures constant activity. However, some imbalances in timing or structure always exist between the duplicated halves, which leads to side-channel leakage. One of the first introduced DPL was Wave Dynamic Differential Logic (WDDL) [2]. It uses only positive gates and uses duplicated master-slave registers to enable precharge propagation. An inversion in WDDL can be realised by a simple wire swapping. A simple example of WDDL conversion from a basic digital circuit is shown in Fig. 1a.

B. Software-Oriented DPL

The concept of DPL for software application was proposed in [4]. This paper only presents a basic idea and overhead estimates. Author motivates around the fact that a standard processors are not designed for DPL. Nevertheless, any function can be computed using a look-up table by concatenating its operands. Using this strategy, Hoogvost et al. proposed a DPL design for PRESENT, but no implementation was given. In a following work, Rauzy et al. [5] presented the first DPL protected implementation of PRESENT, based on a bit sliced implementation. In a following year, Chen et al. [6] extended the approach of [4] in a different way to propose another hiding countermeasure. Both countermeasures are described briefly in the following.

1) Software DPL [5]: Rauzy et al. proposed a design of balanced assembly code, following the dual-rail with precharge logic protocol. They proved the absence of power consumption leakage by a formal method and developed a tool to automatically adjust the assembly code to follow the DPL protocol. The method implements all the basic logical operations using a look-up table with a balanced addressing. Implementation uses bitslicing, therefore one byte carries only one bit of effective information. Two bits out of the byte are chosen to implement DPL. These bits are selected in a way that their leakage characteristics are roughly the same. In the following we use two least significant bits. Thus 1 is encoded as 01 and 0 is encoded as 10. Assembly code for this implementation and a look-up table for basic gates is stated in Tab. I and Tab. II. In the rest of the paper, we refer to this implementation as to *"DPL"* implementation.

[1]Unlike some DPL styles, we consider (1,1) as invalid

TABLE I: Assembly code for *DPL XOR* in AVR

#	Instruction	#	Instruction
0	ldi r1 *a*	5	andi r2 00000011
1	ldi r2 *b*	6	or r1 r2
2	andi r1 00000011	7	ldi r4 *operation*
3	lsl r1 1	8	ldd r3 r4 r1
4	lsl r1 1	9	mov d r3

TABLE II: Look-up tables for `and`, `or`, and `xor`

index	0101	0110	1001	1010
and	01	10	10	01
or	01	01	01	10
xor	10	01	01	10

2) Balanced Encoding [6]: Chen et al. presented a balanced encoding of assembly code that has a constant leakage in common linear leakage models. The target platform is an 8-bit processor where each nibble is balanced by adding complementary bits, in two forms: $b_3\bar{b}_3b_2\bar{b}_2b_1\bar{b}_1b_0\bar{b}_0$ and $b_0\bar{b}_2b_1b_3\bar{b}_1b_2\bar{b}_0\bar{b}_3$. In the paper, authors transform the coding from one to another, in order to minimize side-channel leakage. Since we focus on a fault resistance, we only consider the first of the two encodings. The encoding scheme is explained on a Prince cipher, which can be realized by using a balanced XOR and a balanced table-lookup. Balanced XOR assembly code is stated in Tab. III. In the rest of the paper, we call this implementation the *"Encoding"* implementation.

TABLE III: Assembly code for *Encoding XOR* in AVR

#	Instruction	#	Instruction
0	ldi r1 *a*	19	and r20 r1
1	ldi r2 *b*	20	and r21 r1
2	ldi r16 11110000	21	swap r21
3	ldi r17 11110000	22	or r20 r21
4	and r16 r1	23	ldi r22 00001111
5	and r17 r1	24	ldi r23 00001111
6	swap r17	25	and r22 r2
7	or r16 r17	26	and r23 r2
8	ldi r18 11110000	27	swap r23
9	ldi r19 11110000	28	or r22 r23
10	and r18 r2	29	ldi r21 10100101
11	and r19 r2	30	eor r20 r21
12	swap r19	31	eor r20 r22
13	or r18 r19	32	ldi r24 11110000
14	ldi r17 01011010	33	ldi r25 11110000
15	eor r16 r17	34	and r24 r16
16	eor r16 r18	35	and r25 r20
17	ldi r20 00001111	36	or r24 r25
18	ldi r21 00001111		

C. Side-Channel Security of Software - Oriented DPL

Software-oriented DPL is a fairly new topic with the first implementations surfacing in late 2013. The papers are mainly focused on side-channel protection. Rauzy et al. proposed a formal proof of security for their countermeasure [5] and showed that under perfect Hamming Weight (HW) model, the scheme is side channel resistant. However on a real processor, a perfect HW model is almost impossible to achieve, which breaks the security proof. As a result, the practical security gain of this scheme was 22×, i.e. needs 22× more traces to mount the attack. Similarly, balanced encoding [6] provided good resistance against standard CPA. Recently, it was shown that if the attacker chooses to perform a bit-wise CPA on balanced encoding, the security gain is drastically reduced [7]. Nevertheless, both implementations show some amount of side-channel resistance.

D. Fault Resistance of DPL

A point, which was left unexplored is a potential resistance of such countermeasure against other physical attacks, like faults. In 2009, Selmane et al. [3] demonstrated that WDDL is naturally protected against most (or asymmetric) fault injections. Indeed, in hardware it is very hard to fault two complementary wires with opposite polarity without affecting a third wire. Therefore, in hardware, most faults are asymmetric. Indeed, this is a very interesting property as it paves the way towards design of common countermeasures which resist both side-channel and fault attacks.

In WDDL, one can only use positive gates like AND and OR to construct complex gates. An XOR gate can be designed as shown in Fig. 1b. Under DPL property, i.e. $x_T = \overline{x_F}$, the output is VALID. When either of the inputs deviates from the DPL protocol, the output is NULL. Moreover, when the input at a subsequent step is NULL, the output will be infected towards NULL. In a typical block cipher, where plenty of XOR operations are performed one round after the other, this NULL output will infect (or erase) the whole faulted data and leave no exploitable information for the attacker. Several other DPL countermeasures [8], [9], [10] were proposed after WDDL to improve its side channel resistance. These countermeasure also made sure to not lose this property of fault resilience in their construction.

The software-oriented DPL discussed earlier were not analyzed against fault attacks. In this paper, we try to analyze this unexplored aspect of the software-oriented DPL.

III. Fault Analysis in a Simulated Setting

In this section, we analyze Chen et al. (*"Encoding"* implementation) and Rauzy et al. (*"DPL"* implementation), using a generic assembly code that can be applied to different targets. The analysis is performed in a controlled and simulated environment. In order to better understand the behavior of these countermeasures under faults, we analyzed the basic operations of these implementations. In total, we analyzed three different operations. For the *"Encoding"* implementation, the basic operations are balanced XOR and table look-up. These two operations were enough to implement a cipher like PRINCE. On the other hand, *"DPL"* implementation uses only one operation, i.e. table look-up. This same table look-up is then used for obtaining values of all logical functions. For the analysis, we focus on XOR operation, but all the operations are done in the same fashion and thus will have a similar behavior under faults. In the rest of the section, we describe our fault simulation methodology and obtained results.

A. Fault Simulation Methodology

We developed a fault simulator in 'Java' to perform a code analysis of the aforementioned operations against fault attacks. The methodology of our fault simulator is depicted in Fig. 2. The simulator injects faults in the target code under defined fault models. The output is then checked against expected output to classify it as faulted or correct. Target code is written in a generic assembly language and the inputs and outputs are already encoded. In other words, we do not consider faults while input plaintext encoding or output ciphertext decoding is performed. We simulated several fault models, attacking every instruction from the code, using all the possible inputs and testing different precharge in order to check if it can bias the final result. Simulating one implementation against all the fault models takes few milliseconds, which allows us to get all the results in a very short time.

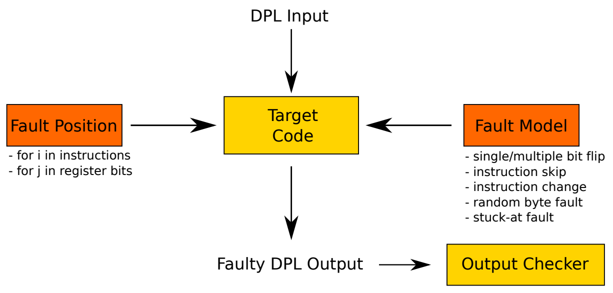


Fig. 2: Methodology for fault injection.

The components of the fault simulator of Fig. 2 are described in the following:

- **DPL Input:** input is encoded with respect to used algorithm design. *Encoding* design has inputs in format $a_3\bar{a}_3a_2\bar{a}_2a_1\bar{a}_1a_0\bar{a}_0$ and $b_3\bar{b}_3b_2\bar{b}_2b_1\bar{b}_1b_0\bar{b}_0$. Therefore, each variable in the first encoding can take 16 different values, i.e. 256 input combinations. *DPL* design uses inputs in format 00000001 for 1 and 00000010 for 0, i.e. 4 input combinations for 2 input values.
- **Faulty DPL Output:** Output encoding is same as for the input. For faulty output deviating from the encoding, an automated format check or a Hamming Weight check can discover faults. We categorize faulty outputs in three different types: *VALID*, *NULL* and *INVALID*. A *VALID* output follows the input encoding and *INVALID* does not. A *NULL* output signifies all zeros at the output.
- **Target Code:** We test the basic gates proposed in each implementation. We tested the schemes on an assembly implementation for a simple microcontroller.
- **Fault Model:** Injects fault following the predefined models. We use common fault models found in literature [11], which are:

- *Single/multiple bit flip:* a content of the destination register of every operation was altered either to simulate a single or a multiple bit flip. In case of DPL, multiple bit flips are just a subset of single and double bit flips, therefore we check vulnerability against these two. We checked all the combinations of bits for every instruction in the code.
- *Instruction skip:* we skipped one or two instructions. Again, we tested all the possible combinations of instruction skips.
- *Random byte fault:* because of the specific encoding format, random byte faults are a subset of single/multiple bit flip faults.
- *Stuck-at fault:* we changed the content of the destination register of all the instructions in the code, one instruction at a time. We tested two values, all zeros and all ones.

B. Results on 'Encoding' Implementation

In this subsection we present simulation results on two operations proposed in [6], `xor` operation and a balanced table look-up.

1) XOR Operation With Constant HW: The code for the balanced XOR is well described in [6] (Tab. III). It shows no vulnerability against a single bit flip attack and a single instruction skip attack. It is, however, vulnerable to a double bit flip attack (Tab. IV) and a double instruction skip attack (Tab. V). Please note that the table columns in this section follow the same structure: the first column contains number of *VALID* faults out of all the possible faults. The following column (or the last two columns, in the case of a double instruction skip attack) contains the instruction together with its line number corresponding to Tab. I and III. Finally, in the case of bit flips, the last column contains bit position(s) where the fault was induced. Our simulator counts bits from left to right.

Also, the implementation is vulnerable against stuck-at faults at two positions. The first one is instruction number 14 (`xor`), the second one is instruction number 29 (`xor`). Both instructions are used to produce the final output, either left (instr. 14) or right (instr. 29) nibble. Because of the behavior of this implementation, stuck-at fault reveals left or right nibble of the second input, either in the plaintext (all zeros stuck-at fault) or as its complement (all ones stuck-at fault). Out of all the inputs, 192 are vulnerable against this type of attack.

2) Table Look-up With Constant HW: Since the balanced look-up table size is 16×16, containing only 16 valid output values, the only way to achieve a valid faulty output is to interchange between these values. Our simulations show that the only valid fault model is a double bit flip fault model, changing consecutive bit pairs in input so that the encoding is satisfied. Results are stated in Tab. VI. Please note that in our implementation, we force the unused addresses to contain *NULL*, i.e. all zeros. The content of unused addresses is not mentioned in the original paper and using all zeros favors our scenario of reducing vulnerability to faults. The transformation of encoding as proposed in the original paper, for side-channel resistance, is ignored in our implementation. This is because the side-channel aspect of this scheme is fairly dealt in the original paper and lies out of scope of this paper.

TABLE IV: Double bit flip faults for *Encoding XOR* implementation.

# of *VALID* faults	Instruction	Bit couples
256	0 LDI	0-1, 2-3, 4-5, 6-7
256	1 LDI	0-1, 2-3, 4-5, 6-7
256	4 AND	0-1, 2-3
256	7 OR	0-1, 2-3
256	10 AND	0-1, 2-3
256	13 OR	0-1, 2-3
256	14 LDI	0-1, 2-3
256	15 XOR	0-1, 2-3
256	16 XOR	0-1, 2-3
256	19 AND	4-5, 6-7
256	29 LDI	4-5, 6-7
256	22 OR	4-5, 6-7
256	25 AND	4-5, 6-7
256	28 OR	4-5, 6-7
256	30 XOR	4-5, 6-7
256	31 XOR	4-5, 6-7
256	34 AND	0-1, 2-3
256	35 AND	4-5, 6-7
256	36 OR	0-1, 2-3, 4-5, 6-7

TABLE V: Double instruction skip faults for *Encoding XOR* implementation.

# of *VALID* faults	Instr. 1	Instr. 2
240	0 LDI	2 AND
192	2 AND	8 AND, 10 OR, 14 LDI, 15 LDI, 16 AND
192	4 OR	8 AND, 10 OR, 14 LDI, 15 LDI, 16 AND
192	6 LDI	16 AND
192	8 AND	14 LDI, 15 LDI
192	10 OR	14 LDI, 15 LDI
192	14 LDI	16 AND
192	15 LDI	16 AND
192	17 AND	23 AND, 25 OR, 29 LDI, 30 LDI, 31 AND
192	19 OR	23 AND, 25 OR, 29 LDI, 30 LDI, 31 AND
192	21 LDI	29 LDI
192	23 AND	29 LDI, 30 LDI
192	25 OR	29 LDI, 30 LDI
192	29 LDI	31 AND
192	30 LDI	31 AND

TABLE VI: Double bit flip faults for *Encoding* implementation of table look-up.

# of *VALID* faults	Instruction	Bit couples
16	0 LDI	0-1, 2-3, 4-5, 6-7
16	1 LDD	0-1, 2-3, 4-5, 6-7

TABLE VII: Double bit flip faults for *DPL XOR* implementation.

# of *VALID* faults	Instruction	Bit couples
4	0 LDI	6-7
4	1 LDI	6-7
4	2 ANDI	6-7
4	3 LSL	5-6
4	4 LSL	4-5
4	5 ANDI	6-7
4	6 OR	4-5, 6-7
4	8 LDD	6-7
4	9 MOV	6-7

TABLE VIII: Single and double instruction skip faults for *DPL XOR* implementation.

# of *VALID* faults	Instr. 1	Instr. 2
4	3 LSL	–
4	4 LSL	–
2	3 LSL	2 ANDI, 5 ANDI
2	4 LSL	2 ANDI, 5 ANDI

C. Results on 'DPL' Implementation

The implementation of Rauzy et al. is coded in two bits. We have chosen the least significant bit pair to implement the design. The rest of the bits are unused.

Simulating faults on implementation shown in Tab I shows no vulnerability against single bit flip faults and stuck-at faults, however the code is vulnerable to instruction skips and dual bit flip attacks. Instruction skips are stated in Tab. VIII and double bit flip faults are stated in Tab. VII, using `xor` as an operation.

IV. Experimental Validation

In this section, we extend our analysis to a practical setting. We perform a fault injection in an 8-bit AVR microcontroller using laser, while running the previously tested software operations. This allows us to validate our assumptions for the simulations and attest if the simulated fault model and effects are practically sound.

A. Experimental Setup and Fault Injection Platform

Our setup is composed of a near-infrared diode pulse laser. The properties of the laser setup are as follows:

- Pulse power: 20 W (reduced to 8 W with 20x objective and to 7 W with 50x objective)
- Pulse repetition: 10 MHz
- Spot size: 30x12 μm (15x3.5 μm with 20x objective and 6x1.4 μm with 50x objective)
- Response to trigger pulse: $\leq$ 100 ns

Intentional `nop` are inserted at beginning of each node to overcome the delay between trigger and laser injection. We used Atmel ATmega328P microcontroller as the DUT, depackaged and mounted on Arduino UNO development board. The chip is 3x3 mm^2, manufactured in 350 nm CMOS technology. Laser injection is performed using an X-Y positioning table with a step precision 0.05 μm. There is a trigger signal set on HIGH (5 V) during the algorithm execution in order to identify the correct time for the fault injection.

Communication with the DUT is done via RS232 interface. We used an oscilloscope for measuring the power consumption of the DUT, for capturing the trigger signal and checking the laser diode current, so that we could determine the delay between sending the trigger signal and activating the laser beam.

B. Experimental Results

In this section we present experimental results obtained by using a laser fault injection technique. The presented results were obtained using the *Encoding XOR* operation as described in Tab III. Similar area results were obtained for other operations as well. First, we tested the device with different values of precharge. However, the results showed that the faults were dependent only on the area and offset, and thus precharge values have no influence on faults.

In total, we were able to obtain 78 different types of faults in several different areas of the chip for all possible input combinations. We found all the three kinds of faults, i.e. *INVALID*, *VALID* and *NULL*. The sensitive area of the chip is approximately 1100x80 μm^2 large ($\approx$ 0.98% of the whole chip area). *VALID* faults were only produced in two areas (with the X coordinate between 1150-1250 μm and 1050-1150 μm). As far as the fault model is considered , there was a majority of 1-bit flips and 2-bit flips, however we also found few single instruction skips, double instruction skips and stuck-at faults. Also most of the faults were impacting the data bus, which was loading operands from the memory.

V. Discussion

We simulated and experimentally verified fault injection on *Encoding XOR*, *Encoding table look-up* and *DPL XOR*. In this part, we make an attempt to bring everything on the same page and derive objective conclusions. To do this, we plot the number of faults for each fault model and resultant output. The analysis is done on faults obtained from simulations and experiments.

The results are presented in Fig. 4, 5 and 6. Bar plots show simulation outputs and pie charts show output distribution from our experimental setting. Since the number of inputs and code-size is different for each tested operation, we normalize the simulation results to represent it on an equivalent scale, i.e [0,1].

For *Encoding XOR*, majority of the faults are *INVALID* for all fault models. Few faults are *VALID* and a negligible number of faults are *NULL*. In Fig. 4, we find a very good coherence between simulation and experimental results, specially when the majority of faults are 1-bit or 2-bits flip.

In *Encoding table look-up* and *DPL XOR*, the simulations report a good mix of *INVALID* and *NULL* (Fig. 5, Fig. 6), which is also observed in experimental fault analysis. It is the number of *VALID* faults which is deviating from simulations to experiments. In simulations, 2% and 10% of 2-bit flip faults are *VALID* for *DPL XOR* and *Encoding table look-up* respectively. *DPL XOR* also encapsulates two instructions vulnerable to single ($\approx$13.3%) and double ($\approx$4.7%) instruction skips.

In the experiments, we see an inflated number of *VALID* and *INVALID* faults than simulations, especially when considering look-up table implementations. Unlike simulations, the fault models are not uniformly distributed for experiments. As previously stated, in our experiments, most of the faults were bit flips and on the data bus. This scenario explains a higher percentage of *VALID* faults. Also, that is why there are more *VALID* faults for *Encoding table look-up* than for *DPL XOR*, despite the fact that only the latter one is vulnerable to instruction skip attacks. To understand the higher number of *INVALID* faults rather than *NULL*, we refer to the area of *DPL*

TABLE IX: Fault propagation in *Encoding XOR* implementation.

Input 1	**Input 2**	**Output**
INVALID	VALID	INVALID
INVALID	INVALID	**VALID**
INVALID	NULL	**VALID**
NULL	VALID	INVALID
NULL	NULL	**VALID**

XOR. It is evident from Fig. 3 that a much larger area produces *INVALID* faults compared to *NULL*. Thus it is more likely to inject *INVALID* faults. This can be due to several factors like placement of the data and instruction bus, underlying register, etc.

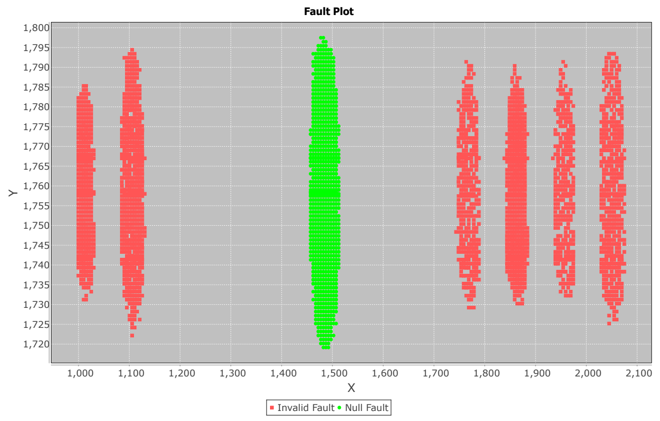


Fig. 3: Distribution of *INVALID* and *NULL* faults for *Encoding table look-up*.

A. Fault Propagation

Another important parameter to consider is a fault propagation. This means, if a particular operation is faulted, how the following operations deal with the fault. In a standard differential fault analysis, a fault in a cipher computation has to propagate to the output. The attacker then uses the faulty and correct ciphertext pair to extract the secret key. If a code is to be made fault resistant, it should not propagate the faults.

A *VALID* fault will always propagate to the output. However, it can be seen in figures that majority of faults are either *INVALID* or *NULL*. To analyze the fault propagation of an operation, we need to check its output when the inputs are faulted. Any *INVALID* or *NULL* input to *DPL XOR* and *Encoding table look-up* will lead to a *NULL* at the output. When a *NULL* is propagated from one stage to another, the final ciphertext will also be *NULL* and erase any sensitive information which allows fault analysis. *Encoding table look-up* will only hold this property if the unused addresses contain all zeros. An alternative solution can be to check on encoding (ex. Hamming weight verification) at the end of the table look-up. However, this problem is similar to redundant PIN verification, where a fault can be used to skip the verification step.

On the other hand, *Encoding XOR* does propagate faults. As we see in Tab. IX, several combinations of inputs can lead to *VALID* output. Also, a combination of *NULL* input and *VALID* input leaks information about the input. For example, inputs $A = 0x00$ and $B = 0xA5$ will result in $0x0F$, i.e directly leaking input B in the form $\bar{b}_3\bar{b}_3\bar{b}_2\bar{b}_2\bar{b}_1\bar{b}_1\bar{b}_0\bar{b}_0$, $b_i \in B$. For instance, a final key XOR with a faulted *NULL* value can directly reveal the last round key.

In a nutshell, *DPL XOR* and *Encoding table look-up* are well designed to hinder fault propagation. However, unlike hardware, in software it is possible to create symmetric or 2-bit flip faults which makes the design vulnerable to fault attacks to an extent. *Encoding XOR* needs to be redesigned to not propagate faults. For example, an *Encoding XOR* based on principle of *DPL XOR*, i.e. based on look-up table can do the trick. In conclusion, to be able to design a fault-resistant countermeasure, *VALID* faults should be minimized and all *INVALID* faults must be converted to *NULL*.

TABLE X: Comparison of different countermeasures.

Countermeasure	SCA resistance	FA resistance	Universality
Encoding [6], [5]	✓	✓	✓
Masking [1]	✓	✗	✓
Shuffling [12]	✓	✗	✓
Error-correcting codes [13]	✗	✓	✗
Time redundancy [14]	✗	✓	✓

Additionally, we try to draw comparison among different software countermeasures in Tab. X. The most studied countermeasure is masking. Masking provides strong resistance against side-channel attacks and its universally applicable being a algorithmic level countermeasure. Similarly, shuffling provides side-channel resistance and can be applied to any algorithm. Both these countermeasure, do not protect against fault attacks. Their security is very hard to prove under the bias introduced by faults. Encoding countermeasures presented in this paper are capable of SCA resistance, FA resistance and can be applied across implementations. For traditional FA countermeasures, error-correcting codes obviously can detect and correct faults. However, SCA resistance of such codes is quite limited and its hard to apply such codes on non-linear operations. Time-redundancy, on the other hand can be widely applied and protect fault attacks, while its SCA resistance rest very low.

VI. Proposed Improvements to Software DPL Countermeasure

In this section we will propose some improvements to the studied *DPL* implementation, i.e. hardening the code against fault attacks. We have chosen this implementation for improvement because it already provides a decent protection thanks to properties of the look-up table. Our aim is to further reduce its vulnerability against fault attacks with minimum possible overhead.

To do so we deeply analyzed the faults we observe in the *DPL* implementation. As shown in Tab. VII and Tab. VIII, this implementation is vulnerable to 2-bit flips, single and double instruction skips.

For resisting single-instruction skips, we need to avoid shift instructions, because these can produce *VALID* outputs in the case one of LSL instructions is skipped. For this purpose we

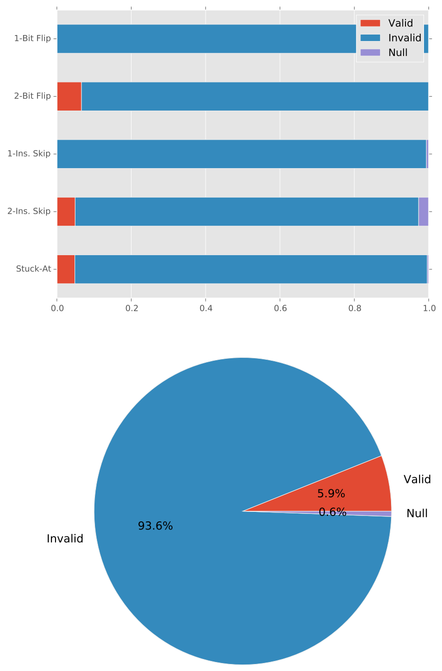


Fig. 4: Fault distributions of *Encoding XOR* simulations and experiments.

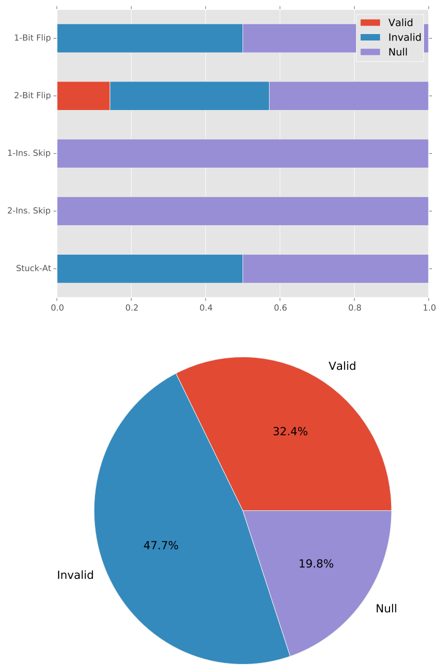


Fig. 5: Fault distributions of *Encoding table look-up* simulations and experiments.

propose to use another look-up table (Tab. XI), returning the concatenated value of both inputs (normally obtained after instruction 6). After this modification, attacker can no longer use an instruction skip attack, since all the faulty outputs will be either *NULL* or *INVALID*.

As for double instruction skips, we propose a solution

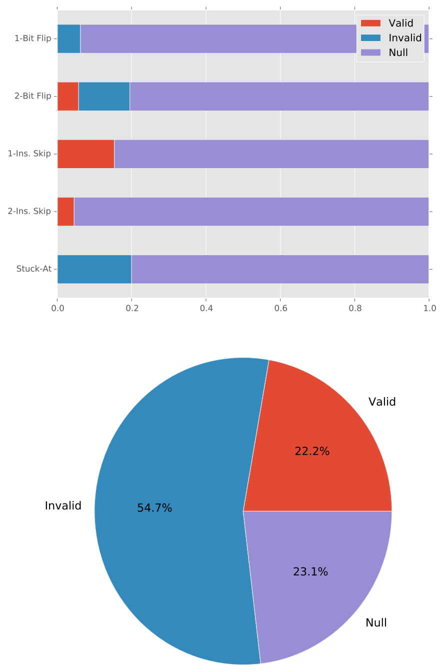


Fig. 6: Fault distributions of *DPL XOR* simulations and experiments.

TABLE XI: Look-up table for concatenating values instead of using shifting and then `or`.

inputs	01	10
01	0101	0110
10	1001	1010

based on partial redundancy, where the values are processed independently and they are compared in the end. Our improved code is stated in Table XII. Number of instructions changed from 10 to 15 and we use three more registers to store the data. In terms of AVR microcontroller clock cycles, overhead is $\approx$ 72.3% (in total 19 clock cycles compared to 11 in original proposal). Modified implementation is secure against single and double instruction skips, single bit flip faults and stuck-at faults. We have checked our implementation with respect to Hamming weight and Hamming distance leakage, both values remain constant for every instruction and different input data.

There are only two instructions vulnerable against double bit flip faults, i.e. instruction 13 (`and`) and 14 (`mov`). Instruction 13 is used for comparison of redundant data. As double-bit flips are the limit of the implemented encoding, it is not possible to detect or resist such faults using only this encoding. Some supplementary code such as added parity or error detection must be used for that purpose. These supplementary methods remains out of scope of this paper.

A. Results

The countermeasure is thoroughly tested under simulated environment and real laser setup. The results are presented in Fig. 7. Simulations were carried out using our `Java` based simulator. The proposed countermeasure resists single bit faults, single and double instruction skips, and stuck-at faults.

TABLE XII: Assembly code for modified *DPL XOR* implementation in AVR

#	Instruction	#	Instruction
0	`ldi r1` *a*	8	`ldd r1 r1 r2`
1	`ldi r2` *b*	9	`ldd r3 r3 r4`
2	`ldi r3` *a*	10	`ldi r6` *operation*
3	`ldi r4` *b*	11	`ldd r5 r6 r1`
4	`andi r1 00000011`	12	`ldd r7 r6 r3`
5	`andi r2 00000011`	13	`and r7 r5`
6	`andi r3 00000011`	14	`mov d r7`
7	`andi r4 00000011`		

In other words, the countermeasure produces no *VALID* faults for these fault models. The only way to produce a *VALID* output is to perform a double bit flip fault in one of the last two instructions. The probability of injecting such bit-flip faults in this countermeasure is 0.0048. Fault propagation stays the same as in the case of *DPL XOR* implementation, therefore *INVALID* and *NULL* inputs lead to *NULL* outputs.

Next, we implemented the countermeasure on AVR chip and tested under back-side laser fault injection. In total, we injected faults in $240,000$ executions of the countermeasure and $13,187$ were faulted. From all the injected faults, $\approx 93.5\%$ resulted in a *NULL* output, the rest were *INVALID* outputs. This situation corresponds well with the simulation results, with the exception of *VALID* faults. As we stated before, for these types of outputs it is necessary to inject the fault very precisely. This brings the difficulty of producing exploitable faults to the same level as for the hardware-based DPL countermeasures.

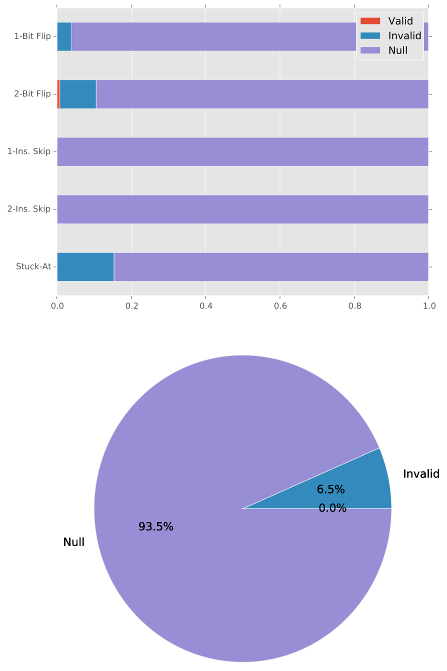


Fig. 7: Fault distributions of *DPL XOR* simulations and experiments, implemented with our countermeasure.

VII. Conclusions

This paper tests two software DPL countermeasures for fault resistance property. Several fault models were tested and a 2-bit flip fault turned out to be most efficient one, often resulting in *VALID* faults. We also show that, unlike hardware DPL, it is not difficult to find a fault with 2-bit flip model in a software case. For instance, in *Encoding table-look up*, about 33% of the total injected faults followed 2-bit flip model. *Encoding XOR* implementation exhibits minimum *VALID* faults but is not resistant to fault propagation. Other two implementations do not have this problem because of the table look-up properties.

Next, we used our understanding to propose an improvement to *DPL XOR*. In simulated settings, the probability of achieving a *VALID* fault is 0.0048. However, we were unable to find a *VALID* fault in experimental verification, owing to the extremely low probability.

To summarize, DPL in hardware can serve as a common countermeasure, however it is not the case for software DPL. Nevertheless, software DPL do exhibit fault resilience. The improvement we proposed in this paper extends the fault resistance property of software encoding schemes to the same level as DPL in hardware.

References

[1] L. Goubin and J. Patarin, “DES and Differential Power Analysis. The "Duplication” Method,” in *CHES*, ser. LNCS. Springer, 1999, pp. 158–172, Worcester, MA, USA.

[2] K. Tiri and I. Verbauwhede, “A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation,” in *DATE’04*, 2004, pp. 246–251, Paris, France.

[3] N. Selmane, S. Bhasin, S. Guilley, T. Graba, and J.-L. Danger, “WDDL is Protected Against Setup Time Violation Attacks,” in *FDTC*, 2009, pp. 73–83.

[4] P. Hoogvorst, J.-L. Danger, and G. Duc, “Software Implementation of Dual-Rail Representation,” in *COSADE*, 2011, Darmstadt, Germany.

[5] P. Rauzy, S. Guilley, and Z. Najm, “Formally Proved Security of Assembly Code Against Leakage,” *IACR Cryptology ePrint Archive*, vol. 2013, p. 554, 2013.

[6] C. Chen, T. Eisenbarth, A. Shahverdi, and X. Ye, “Balanced Encoding to Mitigate Power Analysis: A Case Study,” in *CARDIS*, ser. Lecture Notes in Computer Science. Springer, November 2014, paris, France.

[7] Y.-S. Won, P. Hodgers, M. O’Neill, and D.-G. Han, “On the Security of Balanced Encoding Countermeasures,” in *CARDIS*, ser. Lecture Notes in Computer Science. Springer, 2015, bochum, Germany.

[8] S. Bhasin, S. Guilley, F. Flament, N. Selmane, and J.-L. Danger, “Countering Early Evaluation: An Approach Towards Robust Dual-Rail Precharge Logic,” in *WESS*. ACM, 2010, pp. 6:1–6:8.

[9] M. Nassar, S. Bhasin, J.-L. Danger, G. Duc, and S. Guilley, “BCDL: A High Performance Balanced DPL with Global Precharge and Without Early-Evaluation,” in *DATE’10*. IEEE Computer Society, 2010, pp. 849–854, Dresden, Germany.

[10] A. Moradi and V. Immler, “Early Propagation and Imbalanced Routing, How to Diminish in FPGAs,” in *Cryptographic Hardware and Embedded Systems - CHES 2014*, 2014, pp. 598–615.

[11] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, “Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures,” *Proceedings of the IEEE*, vol. 100, pp. 3056–3076, 2012.

[12] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, “Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note,” in *ASIACRYPT 2012*, ser. LNCS, 2012, vol. 7658, pp. 740–757.

[13] T. G. Malkin, F.-X. Standaert, and M. Yung, “A Comparative Cost/Security Analysis of Fault Attack Countermeasures,” in *Proceedings of the 3rd FDTC*, 2006, pp. 159–172.

[14] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, “Experimental Evaluation of Two Software Countermeasures against Fault Attacks,” in *HOST, 2014 IEEE*, 2014, pp. 112–117.