Mistakes Are Proof That You Are Trying: On Verifying Software Encoding Schemes' Resistance to 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
{jbreier,djap,sbhasin}@ntu.edu.sg

Abstract. Software encoding countermeasures are becoming increasingly popular among researchers proposing code-level prevention against data-dependent leakage allowing an attacker to mount a side-channel attack. Recent trends show that it is possible to design a solution that does not require excessive overhead and yet provides a reasonable security level. However, if the device leakage is hard to be observed, attacker can simply switch to a different class of physical attacks, such as fault injection attack.
Instead of stacking several layers of countermeasures, it is always more convenient to choose one that provides decent protection against several attack methods. Therefore, in our paper we use our custom designed code analyzer to formally inspect a recently proposed software encoding countermeasure based on device-specific encoding function, and compare it with other solutions, either based on balanced look-up tables or balanced encoding. We also provide an experimental validation, using the laser fault injection setup.
Our results show that the device-specific encoding scheme provides a good protection against fault injection attacks, being capable of preventing majority of faults using different fault models.

Keywords: software encoding schemes, formal code analysis, fault injection attacks, countermeasures

1 Introduction

Small general-purpose microcontrollers can be found everywhere nowadays. With emerging technology frameworks like internet-of-things and cyber-physical systems, these devices can control various functions depending on environmental conditions and requirements. As with any other computing devices communicating over unsecured networks, security is one of the primary concerns. For securing communication channels, the obvious choice is to use cryptography. However, despite it is infeasible to break current cryptographic algorithms with

current computing capabilities, these devices can be attacked using physical attack techniques. As majority of these devices are low-cost, they usually do not contain comprehensive protection from attackers exploiting the implementation properties of the algorithm.

In the context of physical attacks, fault attacks pose a serious threat against cryptographic implementations and currently are among the most popular topics in this area. Since the first theoretical attack proposed by Biham and Shamir [1], researchers keep finding new ways to disturb the execution of cryptographic algorithms in devices every year. This has led to development of various sorts of countermeasures, aiming at protecting different parts of the system – at either hardware level, algorithm level, or design level. Many commercial products have been a victim to practical faults attacks leading to economical losses to companies. A (in)popular example is hacking of pay TV access cards using basic voltage glitch.

Since side-channel attacks are another well-utilized subclass of physical attacks, it makes sense to provide countermeasures that can help to prevent against both. Otherwise designers can only pile-up countermeasures. When it comes to masking countermeasures [6], it cannot be directly used to thwart fault attacks, because the masked value can be attacked in the same way than the original value. Hiding countermeasures [12] lower the data-dependent leakage by decreasing the signal-to-noise ratio utilizing various techniques. In contrast to masking, some of these techniques can be used in order to prevent or minimize the chance of successful fault attack.

In this paper, we focus on three software-based hiding countermeasures that can be hardened against fault injection attacks. More specifically, we analyze a bit-sliced software countermeasure following the dual-rail precharge logic (DPL), published by Rauzy et al. [9], a balanced encoding scheme providing constant side-channel leakage [4], and a customized encoding scheme built according to leakage model based on stochastic profiling [8]. For this purpose, we have developed a customized code analyzer that can show vulnerabilities in assembly code. Details on how such analyzers work can be further found, e.g. in [5], and implementation details of our analyzer are described in [2].

The rest of the paper is organized as follows. Section 2 provides the necessary background for our work, describing software encoding countermeasures proposed so far. Details on our custom code analyzer are stated in Section 3. Results of the analysis are detailed in Section 4 and their discussion is provided in Section 5. Finally, Section 6 concludes this paper and provides motivation for further work.

2 Background

The first proposal of *side-channel information hiding* in software was made by Hoogvorst et al. [7]. They suggested to adopt the dual-rail precharge logic (DPL) in the software implementation to reduce the dependance of the power consumption on the data. Their design uses a look-up table method – instead of computing

the function value, the operands are concatenated and used as an address to the resulting value. The idea was explained on PRESENT implementation on AVR microcontroller.

Building on the idea of the seminal work, there were three notable publications published in recent years. The rest of this section provides a short overview of each of them.

2.1 Software DPL Countermeasure

In 2013, Rauzy et al. [9] published a work that follows DPL encoding by utilizing bit-sliced technique for assembly instructions. They developed a tool that converts various instructions to a balanced DPL, according to their design. In their implementation, each byte is used to carry only one bit of information, encoded either as '01' for '1', or '10' for '0'. In the proposal, bits are chosen according to their leakage characteristics. In our work, we use the two least significant bits of the byte. This implementation uses look-up tables with balanced addressing instead of computing the operations directly. Assembly code we used in the code analysis is stated in Appendix A. For the sake of simplicity, we refer to this implementation as to the *'Static-DPL XOR'* throughout the paper.

2.2 Balanced Encoding Countermeasure

Published in 2014 by Chen et al. [4], this work provides assembly-level protection against side-channel attacks by balancing the number of '1's and '0's in each instruction. The code proposed by the authors is aimed for 8-bit platforms and the constant leakage is achieved by adding complementary bit to every bit of information being processed. Therefore, in each instruction, there are four effective bits of information and four balancing complementary bits. Encoding follows $b_3\bar{b_3}b_2\bar{b_2}b_1\bar{b_1}b_0\bar{b_0}$. Other order of bits may be chosen depending on the leakage model. This choice was done based on empirical results. For fault injection evaluation, it does not matter which format is chosen, therefore all the data is transformed. Assembly code we used in the code analysis is stated in Appendix B. In [4], two basic operations are used, i.e. XOR and look-up table (LUT). For the rest of this paper, we will refer to these operations as *'Static-Encoding XOR'* and *'Static-Encoding LUT'*.

2.3 Device-Specific Encoding Countermeasure

In 2016, there was another encoding countermeasure proposal, by Maghrebi et al. [8]. The proposed encoding aims to balance the side-channel leakage by minimizing the variance of the encoded intermediate values. Previous encoding proposals were based on the assumption of Hamming Weight (HW) leakage model. However, the actual leakage model often deviates from HW, which leads to reduction in practical side-channel security of the encoding scheme. The proposal of [8] designs the encoding scheme by taking the actual leakage model into account.

Algorithm 1: Selection of the optimal encoding function [8].

Input : m: the codeword bit-length, n: the sensitive variable bit-length, β_i: the leakage bit weights of the register, where i in $[\![1, m]\!]$

Output: 2^n codewords of m-bit length

for X *in* $[\![0, 2^m - 1]\!]$ **do**
  Compute the estimated power consumption for each codeword X and store the result in table D: $D[X] = \Sigma_{i=1}^{m} \beta_i X[i]$;
  Store the corresponding value of the codeword in the index table $I : I[X] = X$;
Sort the estimated power consumption stored in table D and the index table I accordingly
for j *in* $[\![0, 2^m - 2^n]\!]$ **do**
  Find the $argmin$ of $[\![D[j] - D[j + 2^n]]\!]$;
return 2^n codewords corresponding to $[\![I[argmin], I[argmin + 2^n]\!]$

The side-channel leakage is dependent on the device, and for the microcontroller case, each register leaks the information differently (though the paper argued that most of the registers have more or less similar leakage pattern). In general, the leakage normally depends on the processed intermediate value. The leakage can be formulated as follows:

$$T(x) = L(x) + \epsilon, \tag{1}$$

where L is the leakage function that maps the deterministic intermediate value (x) processed in the register to its side-channel leakage, and ϵ is the (assumed) mean-free Gaussian noise ($\epsilon \sim N(0, \sigma^2)$). The commonly used leakage function used is the n-bit representation. For example, in 8-bit microcontroller, the leakage could be represented as $L(x) = \beta_0 + \beta_1 x_1 + ... \beta_8 x_8$, where x_i is the i-th bit of the intermediate value, and β_i is the i-th bit weight leakage for specific register [11]. For HW model, β_1 to β_8 are considered to be unity. In reality, due to several physical device parameters, β will deviate from unity in either polarity.

The deterministic part of the leakage can then be determined as $\tilde{L} = \mathbf{A} \cdot \beta$, where $\mathbf{A} = (x_{i,j})_{1 \leq i \leq N; 0 \leq j \leq n}$, with $\boldsymbol{x_i}$ as a row element of $\mathbf{A}$ and N denotes the number of measurements. We can then determine $\beta = (\beta_j)_{0 \leq j \leq 8}$ based on the set of traces $\mathbf{T}$, as follows:

$$\beta = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{T}. \tag{2}$$

After profiling of the device to obtain the weight leakages β, the encoding function can be calculated based on the method used in Algorithm 1.

Thus, the main aim of the algorithm is to choose a set of encoding, represented as a look-up table, which minimizes the variance of the estimated power leakage. This is done by considering the leakage bit weights, which are tightly connected to the device, specifically to its registers. Hence, for different registers, different encoding setup has to be considered.

Assembly code with look-up tables is stated in Appendix C. In this paper, we will refer to this implementation as to the *'Device-Specific Encoding XOR'*, as the dependence on leakage model, makes the encoding specific to a device register.

2.4 Evaluating Resistance to Fault Injection Attacks

The first two countermeasures mentioned in this section were in-depth analyzed with respect to fault injection attacks in [3]. To summarize the results, it was shown that the *Static-Encoding XOR* implementation is more vulnerable to fault attacks due to fault propagation. *Static-DPL XOR* benefits from the table look-up properties that force the value to `0x00` every time a bad address is fetched to the loading instruction. This implementation was further analyzed and improved, making the chance of a successful attack negligible.

In this paper, we analyze the *Device-Specific Encoding XOR* implementation, compare it with previous results and we provide insights that can help designers in developing encoding schemes that are resistant against fault attacks. To make the paper self-contained, we reproduce the attacks of [3] and extend it to device-specific encoding.

3 Verification with the Code Analyzer

In order to formally verify the resistance of assembly code against fault injection attacks, we have developed a code analyzer. This section contains the design and implementation details as well as the methodology we used for analyzing the encoding implementations.

The architecture of our code analyzer is depicted in Fig. 1. Left side of the figure shows the architecture of a standard microcontroller and right side is a high-level class diagram of our analysis software written in Java language. The code analyzer is a custom instruction set simulator that is capable of simulating faults at any stage of the code execution. Time complexity of the evaluation is linear.

Program code is fetched into the analyzer as a text file, following the assembly code structure. It is then broken into instructions – different subclasses of the *Instruction* class. Memory contains all the look-up tables that are statically pre-programmed instead of fetching them from a file. Registers are implemented as arrays, containing `0x00` before the program execution. *MuC* class serves as the instruction set simulator, executing instructions and performing operations on registers and memory. This class also provides the fault injection functionality – it analyzes every instruction against a chosen fault model.

The modular approach to the code analyzer improves its reusability, where it is only necessary to extend the instruction set in order to analyze a different device.

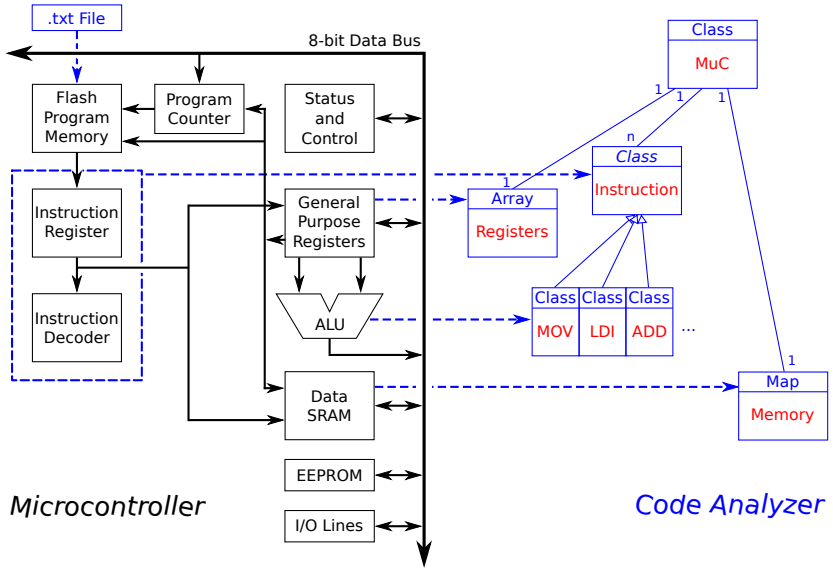


Fig. 1: Representation of microcontroller components in the Java code analyzer.

3.1 Fault Injection Analysis

The abstraction of the process of code execution and fault injection is depicted in Fig. 2. The middle part serves as a standard instruction set simulator, taking the given input, executing the code and producing the output. This process is repeated for every possible combination of inputs for every instruction and every fault model. We are analyzing resistance against four basic fault models: bit flip, random byte fault, instruction skip, and stuck-at fault. After the output is produced, it is analyzed by the validator which decides whether the fault changed the resulting value and if this value is useful for fault attack. We consider inputs and outputs already encoded, analyzing fault tolerance with respect to encoding/decoding is out of scope of this paper.

In the following, we will briefly describe parts of the code analyzer:

- **Instruction Set Simulator:** As stated previously in this section, the assembly code is fetched to the simulator as a text file. It accepts three different data encoding formats, according to what algorithm is currently being used. For the *Static-DPL XOR*, it accepts input in the bit-sliced complement form: $000000b_0\bar{b_0}$, therefore there are 4 possible input combinations. The *Static-Encoding XOR* accepts four bit complement format: $b_3\bar{b_3}b_2\bar{b_2}b_1\bar{b_1}b_0\bar{b_0}$, resulting to 256 input combinations. The same number of combinations is analyzed for the *Device-Specific Encoding XOR*, where the number of codewords is 16 for 8-bit code.
- **Fault Injection Simulator:** In order to get the information about algorithm resistance against fault injection, we analyze four fault models. In the

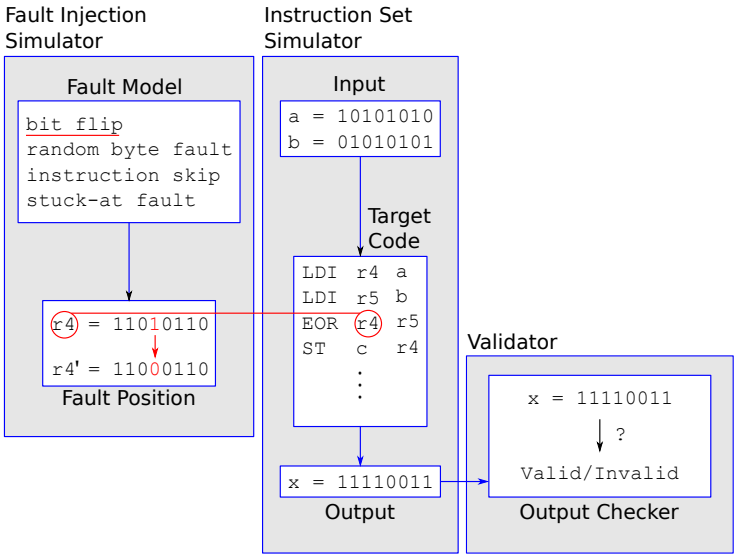


Fig. 2: Schematic of injecting a fault during the execution in the code analyzer.

case of a fault being injected into the data, we change the content of the destination register of an instruction.

In bit flip fault model, we inject single and double bit flips into the *Static* countermeasures. There is no need to test other multiple bit flips, since all of them are just a subset of those two, because of DPL properties. Therefore, e.g. if an algorithm is not vulnerable against single bit flips, it will not be vulnerable against other odd-number bit flips, and vice-versa. In case of the *Device-Specific Encoding XOR*, we test all possible combinations of bit flips.

Random byte fault model is a subset of bit flip fault model when it comes to code analysis, therefore this model is already included in the previous testing.

To analyze vulnerable parts against instruction skip attack, we skip either one or two instructions from the code, checking all the possible combinations. More complex instruction skip models are not considered because of the impracticability to implement them in the real environment.

Finally, to analyze the resistance against the stuck-at-fault model, we change the value of the destination register either to `0x00` or to `0xFF`.

– **Validator:** The final part of the code analyzer checks the resulting output and assigns it to one of the following pre-defined groups:

  - *VALID:* This is the most useful type of output an attacker can get. Outputs in this category follow the proper encoding of analyzed algorithm, but the value deviates from the expected value with respect to given inputs. A *VALID* fault can be directly exploited with fault injection attack.

- *INVALID:* This type of output does not follow the encoding. Therefore, it can be easily recognized by an output checker which can then decide to discard the value instead of further propagation.
- *NULL:* This type of fault has one of the following values: `0x00` or `0xFF`. These outputs are mostly produced by look-up table implementations and can be easily recognized as well.

4 Results

To analyze different software encoding countermeasures against fault injection attacks, we implemented the basic operations of each previously discussed encoding scheme, i.e. *Static-DPL XOR*, *Static-Encoding XOR*, *Static-Encoding LUT* and *Device-Specific Encoding XOR* implementation. The corresponding code is provided in the appendices. The analysis follows a two-step approach. The first step involves a comprehensive fault analysis by putting the code under specially designed code analyzer. The main objective of this comprehensive code analysis is to uncover any native vulnerabilities in the encoding scheme under individual fault models. Such analysis cannot be done in a practical setting due to limited control over the injected fault model for a given equipement setting. Albeit it is possible to inject all the discussed fault models, it is not easy to control the fault model at will. In the following step, the corresponding code is implemented on a real AVR microcontroller and tested under laser fault injection. The objective of practical validation is to find which of the known vulnerabilities are producible with equipement at hand.

4.1 Code Analysis

To analyze vulnerabilities in the software encoding schemes, the basic operations were fed to the code analyzer. The analyzer considers 3 different fault models, i.e. stuck-at, bit flips and instruction skip. Both single and multiple bit-flips are possible. The first three analysed operations i.e. *Static-DPL XOR*, *Static-Encoding XOR*, *Static-Encoding LUT* are a special case, where more than 2-bit flips are equivalent to 1-bit or 2-bit flips eventually. The analyzer reports the impact on the final output in presence of discussed fault models. This is represented as a normalized distribution of faulty output for each considered fault models. Three outputs are expected: `VALID`, `INVALID` and `NULL`. `VALID` implies that final faulty output stays within the encoding. Similarly, `INVALID` refers to the faulty output which is no longer in the applied encoding. `NULL` faults are `0x00` or `0xFF` values at the output. While `VALID` faults stay within the encoding and can lead to differential fault analysis (DFA), it is rather less likely with `INVALID` faults. On the other hand, `NULL` deletes any data dependent information, disabling any further exploitation by DFA. Therefore, `VALID` faults must be prevented at all costs, while keeping `INVALID` in check and maximizing `NULL` faults. The analysis results for Static-DPL XOR, Static-Encoding XOR,

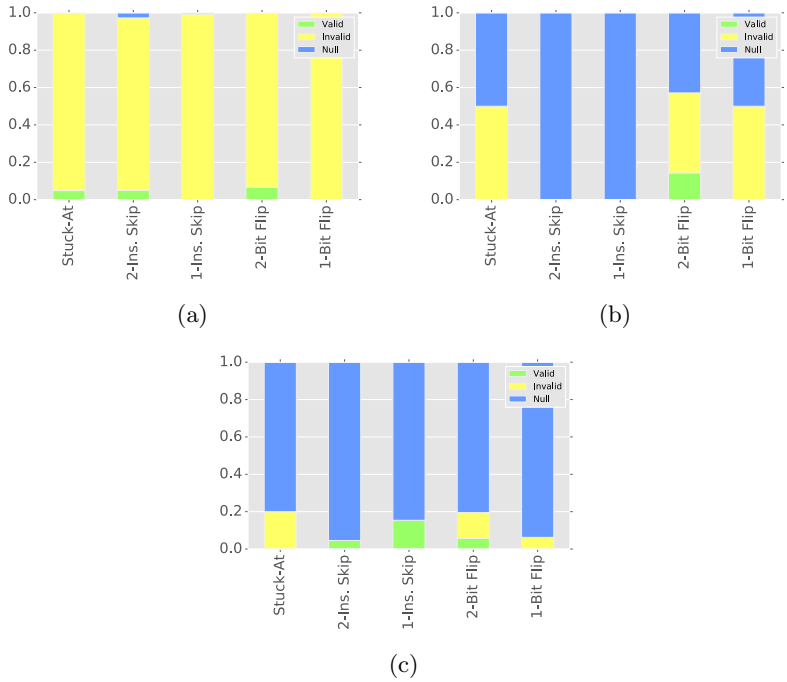


Fig. 3: Fault distributions of (a) *Static-Encoding XOR*,(b) *Static-Encoding LUT*, (c) *Static-DPL XOR* code analysis.

Static-Encoding LUT are shown in Fig. 3. We discuss each of the results in the following.

The fault distribution of *Static-Encoding XOR* is shown in Fig. 3 (a). This encoded operation does not produce any `VALID` faults for 1-instruction skip and 1-bit flip. The percentages of `VALID` faults for other fault models stay between 4-6%. Majority of the faults (92-100%) result in `INVALID` faults while only double instruction skip result in a non-negligible `NULL` faults (2.7%). Although `INVALID` faults are more desirable than `VALID` faults, later we will show that some `INVALID` can be exploitable in particular for *Static-Encoding XOR*.

The *Static-Encoding LUT* shows an altogether different fault resistance (Fig. 3 (b)). This encoded operation produces much more `NULL` faults than the previous case, which is a desirable property. Instruction skips result in 100% `NULL` faults, while stuck-at and 1-bit flips produce 50% `INVALID` and 50% `NULL` faults. The only way to produce `VALID` faults in this operation is to inject 2-bit flips which result in 14.2% `VALID` faults. Rest of the faults would result in `INVALID` or `NULL` faults with equal probability.

The analysis results of *Static-DPL XOR* are shown in Fig. 3 (c). While no `VALID` faults are possible for stuck-at and 1-bit flip, it stays below 6% for 1-instruction skips and 2-bit flips. The worst performance is under 2-instruction skip model, where the percentage of `VALID` faults is as high as 15.3%. The high

vulnerability against 2-bit flips can be explained by the fact, that 2-bit flips are the limit of the dual-rail encoding scheme. Apart from these, the other faults are more likely to be `NULL` rather than `INVALID`, which is desirable.

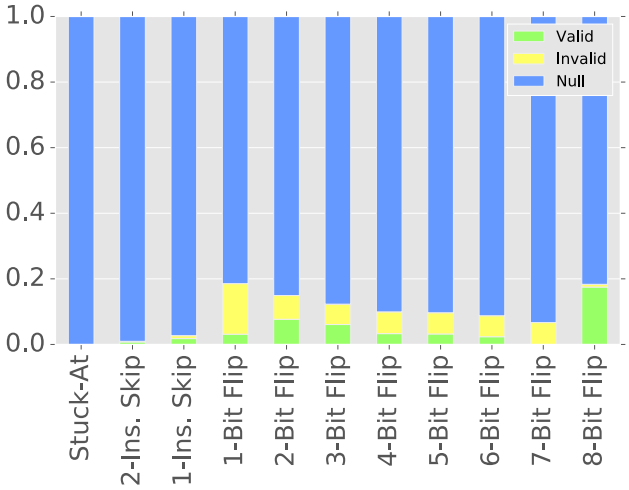


Fig. 4: Fault distributions of *Device-Specific Encoding XOR* code analysis.

Finally we applied the code analysis on *Device-Specific Encoding XOR* operation as shown in Fig. 4. The difference from previous cases is that, here multi-bit flips cannot be dealt as a subset of 1-bit and 2-bit flips. Therefore the analysis covers bit flips from 1-bit to 8-bits, i.e. the data width of the target processor. It can be easily observed that this encoding scheme is more likely to produce `NULL` faults which is highly desirable. For the `VALID` faults, stuck-at model produce nones, while only $< 2\%$ can be achieved by instruction skips. In case of bit flips, the percentage of `VALID` faults stays between 2-7% with the exception of 7 and 8 bit flips. However, for different β coefficients used in the leakage function, results on bit flips should be slightly different, but consistent with the expectations. The total value of *VALID* bit flips for all the possibilities range within 4.2-4.7%, but their distribution is different, depending on used coefficient.

4.2 Experimental Evaluation

Following the code analysis, the fault resistance is experimentally verified. The fault injection is done with a near-infrared diode pulse laser with a pulse power of 20 W (reduced to 8 W with 20× objective). The pulse repetition rate is 10MHz and spot size is 30×12 μm (15×3.5 μm with 20× objective). Intentional `nop` are inserted at beginning of each node to overcome the 100 ns delay between trigger and laser injection. The target platform is Atmel ATmega328P microcontroller,

de-packaged and mounted on Arduino UNO development board. The surface area of the chip is 3x3 mm^2, which is manufactured in 350 nm CMOS technology. An X-Y positioning table with a step precision 0.05 μm is used to scan the chip surface and perform laser injection. The timing of injection is synchronised with executed code using a code-generated trigger. The injection platform along with the target is shown in Fig. 5.

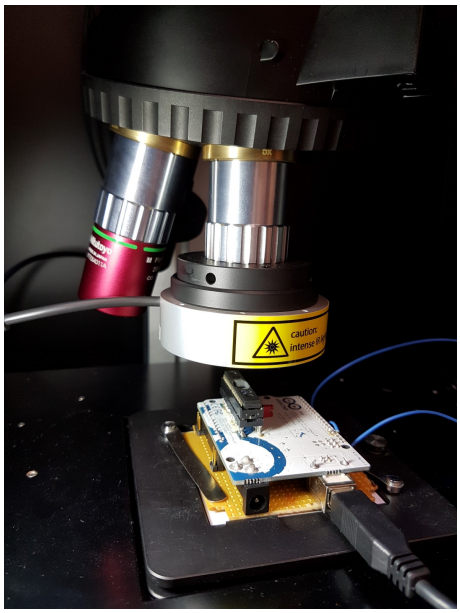


Fig. 5: ATmega328P device under a near-infrared diode laser injection setup.

The prime difference from the previous analysis is that in the experimental validation, we do not precisely control the fault model. Moreover the fault models are not uniformly distributed. Before starting the real experiment, we performed some profiling on the target with basic assembly code and verified that all the fault models are possible to produce experimentally. Next, we flash the assembly code of the four previously discussed software encoding operations. We essentially note the percentage of `VALID`, `INVALID` and `NULL` faults produced for each tested operation. The results are summarised in Fig. 6.

Static-Encoding XOR shows the best consistency with the simulated analysis previously (see Fig. 6 (a)). While 93.56% of the faults are `INVALID`, only 5.88% `VALID` were produced. Moving towards *Static-Encoding LUT*, we observe a 32.42% `VALID` faults in Fig. 6 (b). Since a `VALID` fault in this implementation can only result from even bit flips, this infers that the fault model distribution is biased towards multiple bit flips in our experiments. Similarly, we also observe a 22.2% `VALID` faults in *Static-DPL XOR* (Fig. 6 (c)) owing to the prevalent multiple bit flip model.

When it comes to *Device-Specific Encoding XOR* (Fig. 6 (d)), results show distribution very similar to the one obtained by the code analysis. Because it is more likely to produce bit flips when injecting faults in the microcontroller, at 13.5% an inflated number of *VALID* faults can be observed in this case, with a relatively small number of *INVALID* faults. As expected, *NULL* outputs are dominant i.e. 82.5% , because of the look-up table properties.

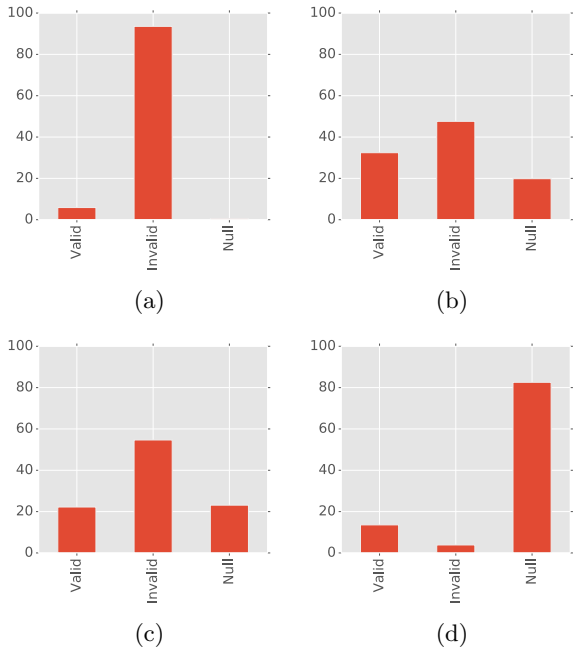


Fig. 6: Fault distributions of (a) *Static-Encoding XOR*,(b) *Static-Encoding LUT*, (c) *Static-DPL XOR*, and (d) *Device-Specific Encoding XOR* experiments.

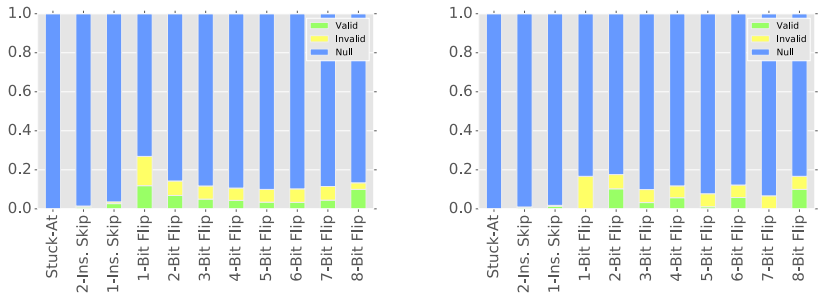
5 Discussion

In this section, we will discuss some important parameters of particular encoding implementation with respect to fault injection attacks.

5.1 Selection of β Coefficients

We considered several parameters for the code analysis of *Device-Specific Encoding XOR*. We analyzed different β values scenarios. We considered the case where the variance of the β is relatively high (the βs might be cancelling each other), and the case where the variance of the β is low (almost Hamming weight).

The most significant difference can be observed in the result for implementation with β coefficients that do not follow Hamming weight leakage model (stated in Fig. 7 (a)). From the figure, it can be observed that the number of 1-bit flips is inflated, compared to the almost Hamming weight case (stated in Fig. 7 (b)). The behavior of the faults shows contrast between different beta values, which is not the case for other encoding schemes, and hence could be further investigated.



(a) (b)

Fig. 7: Fault distributions of *Device-Specific Encoding XOR* code analysis with (a) high variance and (b) almost Hamming weight

5.2 Fault Propagation

When considering security of different implementations, fault propagation is an important factor that can significantly affect the possibility to mount an attack. In case we want to prevent a successful fault attack, it is necessary to avoid the propagation of an *INVALID* output when it is fed as an input to a next iteration of the algorithm. Otherwise, this output could leak some information about the processed data and therefore, allow an attacker to use the differential fault analysis.

From this point of view, look-up table implementations have an advantage, since every input that does not follow the encoding will be automatically converted to *NULL*. Analysis results of *Static-DPL XOR*, *Static-Encoding LUT*, and *Device-Specific Encoding XOR* show that if any of the inputs is either *INVALID* or *NULL*, it will always output *NULL*. Situation with the *Static-Encoding XOR* is different because of the algorithm design. There are several combinations of inputs that lead to *VALID* faults – more specifically, any combination of:

- Two *INVALID* inputs,
- Two *NULL* inputs,
- *INVALID* and *NULL* inputs.

Moreover, for a combination of *VALID* and *NULL* inputs leaks a complete information about the *VALID* input in the form $\bar{v_3}\bar{v_3}\bar{v_2}\bar{v_2}\bar{v_1}\bar{v_1}\bar{v_0}\bar{v_0}$, where $v_3v_2v_1v_0$ is the original input.

To summarize, table look-up implementations provide solid protection against fault attacks when it comes to fault propagation. Any other implementation that uses standard operations performed by using ALU, can be vulnerable if it is not directly designed with such goal in mind. Therefore, when designing a fault resistant algorithms along with the side-channel resistance, look-up tables can offer fault propagation cancellation by default.

6 Conclusion

This paper summarizes fault attack resistance of three software-based encoding schemes that were introduced to prevent side-channel attacks. We mainly aim at analyzing the *Device-Specific Encoding XOR* implementation that was proposed recently. Our results provide insights and comparison against the other schemes (*'Static'* encoding schemes) that have been analyzed in previous work [3].

In general, table look-up schemes offer higher level of security by thwarting the fault propagation in case of several algorithm iterations. *Static-Encoding XOR* might look the best from the experimental results, however, fault propagation properties of this design allow attacker to easily mount a DFA attack or to directly observe inputs passed to the algorithm in case of other input being of a *NULL* type. After considering this phenomenon together with code analysis and experimental results, we conclude that the *Device-Specific Encoding XOR* is currently the most secure scheme with respect to fault attacks and provides a decent level of security.

For the future work, we would like to extend the code analyzer to support pipelined architectures, being able to discover vulnerabilities w.r.t. more comprehensive fault models, like cache attacks (e.g. as described in [10]).

References

1. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: CRYPTO '97, LNCS, vol. 1294, pp. 513–525 (1997)
2. Breier, J.: On analyzing program behavior under fault injection attacks (to appear). In: Availability, Reliability and Security (ARES), 2016 Eleventh International Conference on. pp. 1–5. IEEE (Aug 2016)
3. Breier, J., Jap, D., Bhasin, S.: The other side of the coin: Analyzing software encoding schemes against fault injection attacks. In: 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 209–216. IEEE (2016)
4. Chen, C., Eisenbarth, T., Shahverdi, A., Ye, X.: Balanced Encoding to Mitigate Power Analysis: A Case Study. In: CARDIS. Lecture Notes in Computer Science, Springer (November 2014), paris, France
5. Dureuil, L., Potet, M.L., de Choudens, P., Dumas, C., Clédière, J.: From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference, pp. 107–124. Springer International Publishing, Cham (2016), `http://dx.doi.org/10.1007/978-3-319-31271-2_7`

6. Goubin, L., Patarin, J.: DES and Differential Power Analysis. The "Duplication" Method. In: CHES. pp. 158–172. LNCS, Springer (1999), Worcester, MA, USA
7. Hoogvorst, P., Danger, J.L., Duc, G.: Software Implementation of Dual-Rail Representation. In: COSADE (2011), Darmstadt, Germany
8. Maghrebi, H., Servant, V., Bringer, J.: There is wisdom in harnessing the strengths of your enemy: Customized encoding to thwart side-channel attacks – extended version –. Cryptology ePrint Archive, Report 2016/183 (2016), `http://eprint.iacr.org/`
9. Rauzy, P., Guilley, S., Najm, Z.: Formally Proved Security of Assembly Code Against Leakage. IACR Cryptology ePrint Archive 2013, 554 (2013)
10. Rivière, L., Najm, Z., Rauzy, P., Danger, J.L., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of ARMv7-M architectures. In: Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on. pp. 62–67 (May 2015)
11. Schindler, W., Lemke, K., Paar, C.: A stochastic model for differential side channel cryptanalysis. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 30–46. Springer Berlin Heidelberg (2005)
12. Tiri, K., Verbauwhede, I.: A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In: DATE'04. pp. 246–251 (2004), Paris, France.

A Assembly Code For *Static-DPL XOR* Implementation

Table in this section contains assembly code used for the code analysis. Note that there are several differences in comparison to the original paper. We precharge all the registers before the code execution, therefore there is no need to use precharge instructions. The other change is in instructions 7 and 8, where we first load the operation code (can take values `01010101` for `and`, `10101010` for `or`, and 01100110 for `xor`) and then we execute `ldd` instruction using the destination register, operation code and value. Look-up tables are stated in Table 2.

Table 1: 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 2: Look-up tables for `and`, `or`, and `xor`

index	0000 - 0100	0101	0110	0111 - 1000	1001	1010	1011 - 1111
and	00	01	10	00	10	01	00
or	00	01	01	00	01	10	00
xor	00	10	01	00	01	10	00

B Assembly Code for *Static-Encoding XOR* Implementation

The code stated in Tab. 3 follows the originally proposed algorithm for *Static-Encoding XOR*. This implementation uses several constants, either for clearing and precharging the registers before loading the data (e.g. `ldi r16 11110000`), or for changing the data to proper encoding format (e.g. `ldi r17 01011010`).

Table 3: 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 Assembly Code for *Device-Specific Encoding XOR* Implementation

In this section, we describe the code used for *Device-Specific Encoding XOR*. After determining the bit leakage weights, and computing the encoding based on Algorithm 1, several look up tables are constructed. In Table 4, the pseudocode for the encoding is presented. First, the upper nibble is retrieved for input a and b (a_h and b_h) under the encoding format ($f(a_h)$ and $f(b_h)$), using the *luthb* table, followed by the lookup table *lutop* used to perform xor operation ($LUT(f(a_h) << 4||f(b_h)) = f(a_h \oplus b_h)$). Similar procedure is done for the lower nibble, using the *lutlb*.

Table 4: Assembly pseudocode for *Device-Specific Encoding XOR* in 8-bit AVR.

#	Instruction	#	Instruction
1	ldi r1 *a*	12	eor r4 r4
2	ldi r2 *b*	13	ldd r4 *lutlb* r1
3	eor r3 r3	14	eor r5 r5
4	ldd r3 *luthb* r1	15	ldd r5 *lutshift* r4
5	eor r4 r4	16	eor r6 r6
6	ldd r4 *lutshift* r3	17	ldd r6 *lutlb* r2
7	eor r5 r5	18	or r5 r6
8	ldd r5 *luthb* r2	19	eor r4 r4
9	or r5 r4	20	ldd r4 *lutop* r5
10	eor r3 r3		
11	ldd r3 *lutop* r5		