NANYANG TECHNOLOGICAL UNIVERSITY

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

Jakub Breier, Dirmanto Jap, and Shivam Bhasin

Physical Analysis and Cryptographic Engineering
Nanyang Technological University, Singapore

4 May 2016

Table of Contents

Table of Contents

Dual-Rail Precharge Logic (DPL)[2]

- Hiding countermeasure against SCA - hides the data dependant leakage from the device.
- DPL is a circuit-level countermeasure which removes data-dependant leakage by introducing a generated False (F) rail to compensate the activity of the original True (T) rail.
- *Precharge* phase is a spacer between every two *Evaluation* phases - it hides both Hamming distance and Hamming weight leakage.
- Selmane et al.[1] showed in 2009 that DPL possesses properties that resist fault attacks naturally.

---

[1] N. Selmane, S. Bhasin, S. Guilley, T. Graba, and J.-L. Danger. WDDL is Protected Against Setup Time Violation Attacks, FDTC'09.

[2] K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation, DATE'04

Software DPL

- The first software DPL was presented in 2011 by Hoogvorst et al.[3].
- In our work we examined two software DPL schemes:
  - Bit sliced implementation of balanced assembly code, using look-up tables[4], *"DPL"* implementation.
  - Balanced encoding achieved by adding complementary bits to processed data[5], *"Encoding"* implementation.

---

[3] P. Hoogvorst, J.-L. Danger, and G. Duc. Software Implementation of Dual-Rail Representation, COSADE 2011.

[4] P. Rauzy, S. Guilley, and Z. Najm. Formally Proved Security of Assembly Code Against Leakage, PROOFS 2014.

[5] C. Chen, T. Eisenbarth, A. Shahverdi, and X. Ye. Balanced Encoding to Mitigate Power Analysis: A Case Study, CARDIS 2014.

"DPL" Implementation

- All the logical gates are implemented by using look-up tables (LUT) with balanced addressing.
- Bit-slicing – one byte carries only one bit of effective information.
- Only last two bits of each byte are used – 1 is encoded as 01 and 0 is encoded as 10.

Table: Look-up tables for *"DPL"* implementation.

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

"Encoding" Implementation

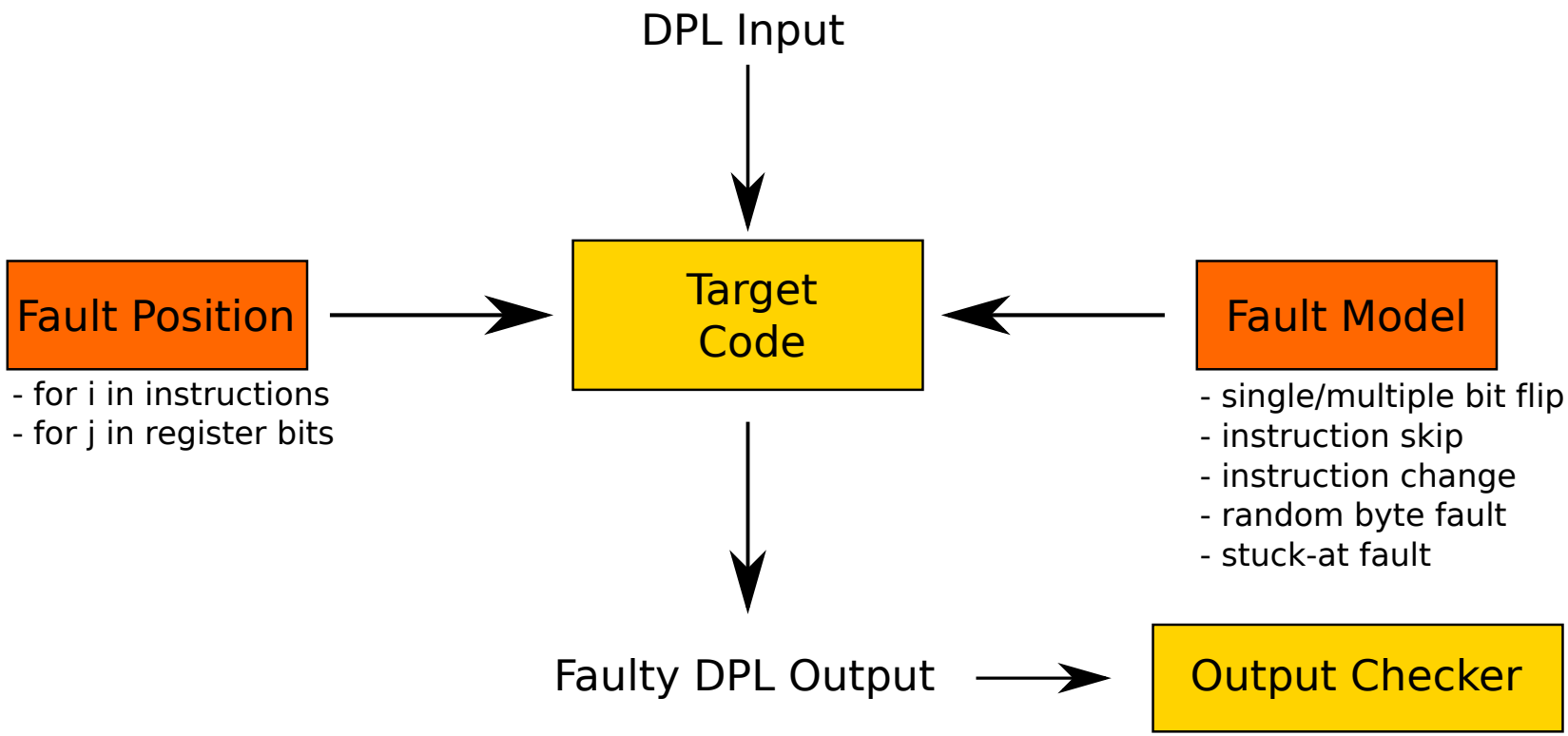
- One byte carries 4 bits of information.
- Each nibble is balanced by adding complementary bits, in one of the two forms: $b_3\bar{b_3}b_2\bar{b_2}b_1\bar{b_1}b_0\bar{b_0}$ and $b_0\bar{b_2}b_1b_3\bar{b_1}b_2\bar{b_0}\bar{b_3}$.
- Following this rule, intermediate value at every point of time has Hamming weight 4.
- The scheme is explained on Prince cipher, which can be realized by using a balanced XOR and a balanced table-lookup – we have examined both operations.

Table of Contents

Fault Simulations

- Fault simulator was written in Java.
- We used 8-bit AVR microcontroller assembly code.
- The simulator injects faults in the target code under defined fault models.
- We considered inputs and outputs already encoded.
- We analyzed every instruction of code.

Fault Simulation Methodology



Inputs/Outputs

- *DPL:* uses inputs/outputs in format 00000001 for 1 and 00000010 for 0. There are only two possible values for a valid input, resulting in 4 different combinations of operands.
- *Encoding:* has inputs/outputs in format $a_3\bar{a_3}a_2\bar{a_2}a_1\bar{a_1}a_0\bar{a_0}$ and $b_3\bar{b_3}b_2\bar{b_2}b_1\bar{b_1}b_0\bar{b_0}$. Therefore, one variable in this encoding can take 16 different values, resulting in 256 input combinations.

Outputs

We defined three possible output sets:

- *VALID* – output follows the proper encoding of each implementation.
- *INVALID* – output does not follow the proper encoding.
- *NULL* – output is all zero.

Fault Models

- *Single/multiple bitflip* – a content of the destination register of every operation was altered either to simulate single or multiple bit flip.
- *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.

Table of Contents

Vulnerability Against Fault Models

Fault model	*Encoding* XOR	*Encoding* LUT	*DPL*
Single bit flip	No	No	No
Double bit flip	Yes	Yes	Yes
Single instruction skip	No	No	Yes
Double instruction skip	Yes	No	Yes
Stuck-at fault	Yes	No	No

Experimental Setup

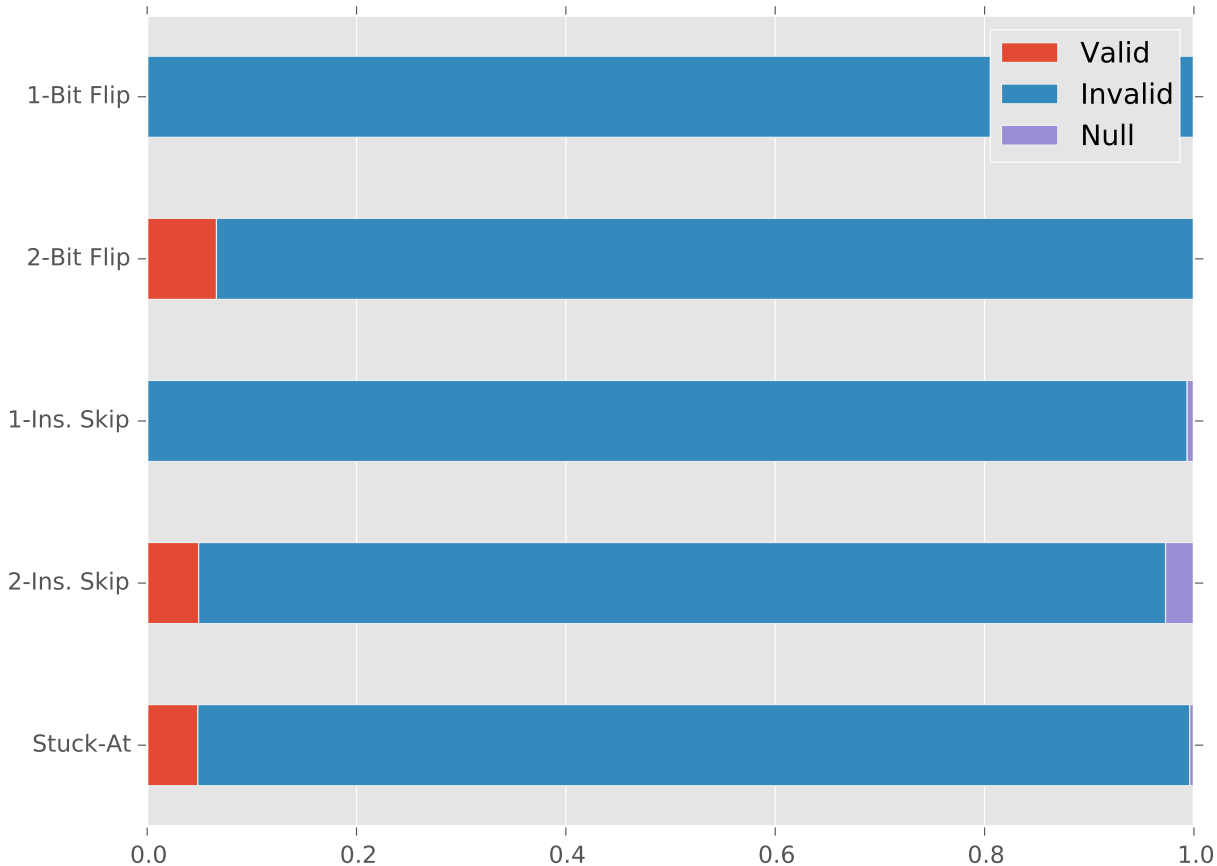
- DUT: Atmel ATmega328P microcontroller running at 16 MHz
- Laser: Infrared 1064 nm diode pulse laser.
- We found all the three kinds of faults, i.e. *INVALID*, *VALID* and *NULL*.
- The sensitive area of the chip is approximately 1100×80 μm^2 large, out of 3×3 mm^2 ($\approx$ 0.98% of the whole chip area).

Results Overview

- We tested five different fault models and the faulty output could attain three possible states (*VALID*, *INVALID*, *NULL*).
- 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* – this situation corresponds with experimental results.
- In *Encoding table look-up* and *DPL XOR*, the simulations report a good mix of *INVALID* and *NULL* – however, number of *VALID* faults deviates from simulations to experiments.

Encoding XOR Implementation - Simulations and Experiment

Simulations

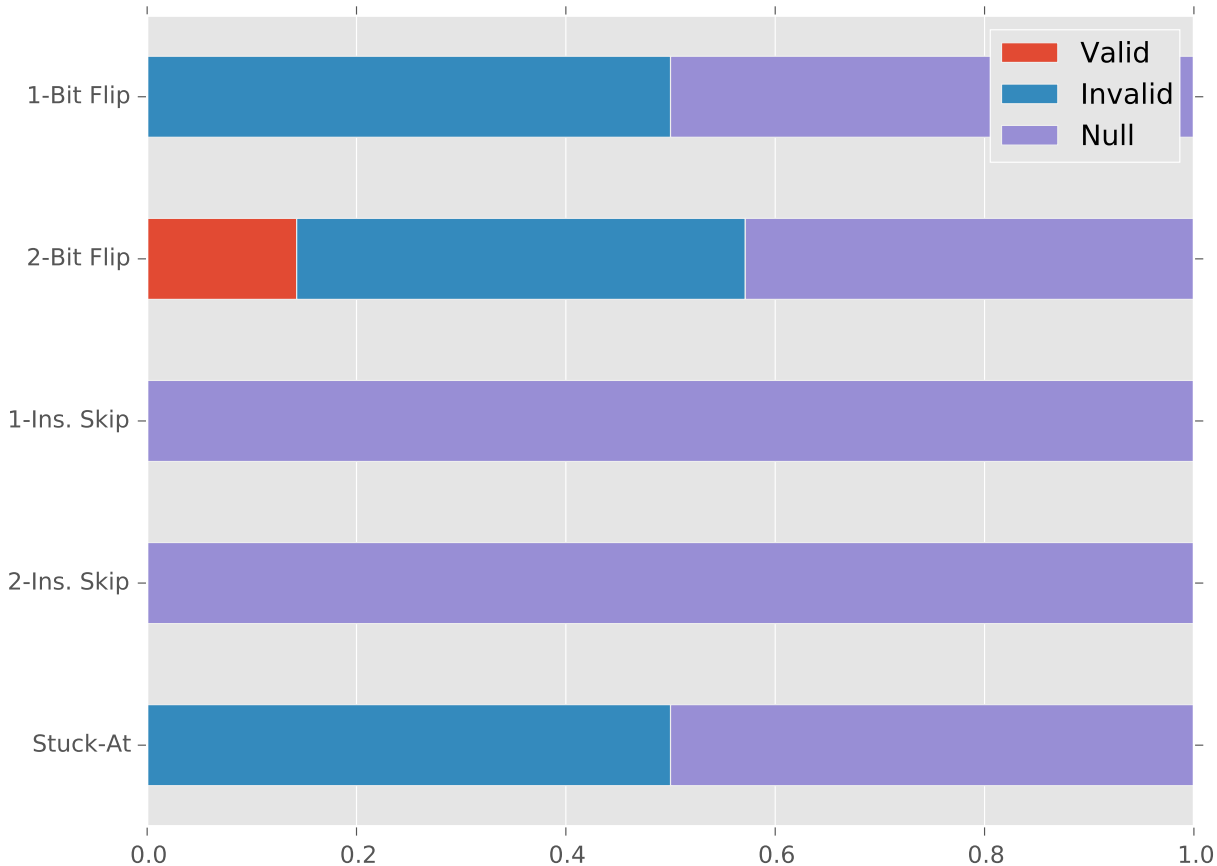


Experiment

VALID	5.9%
INVALID	93.6%
NULL	0.6%

Encoding LUT Implementation - Simulations and Experiment

Simulations

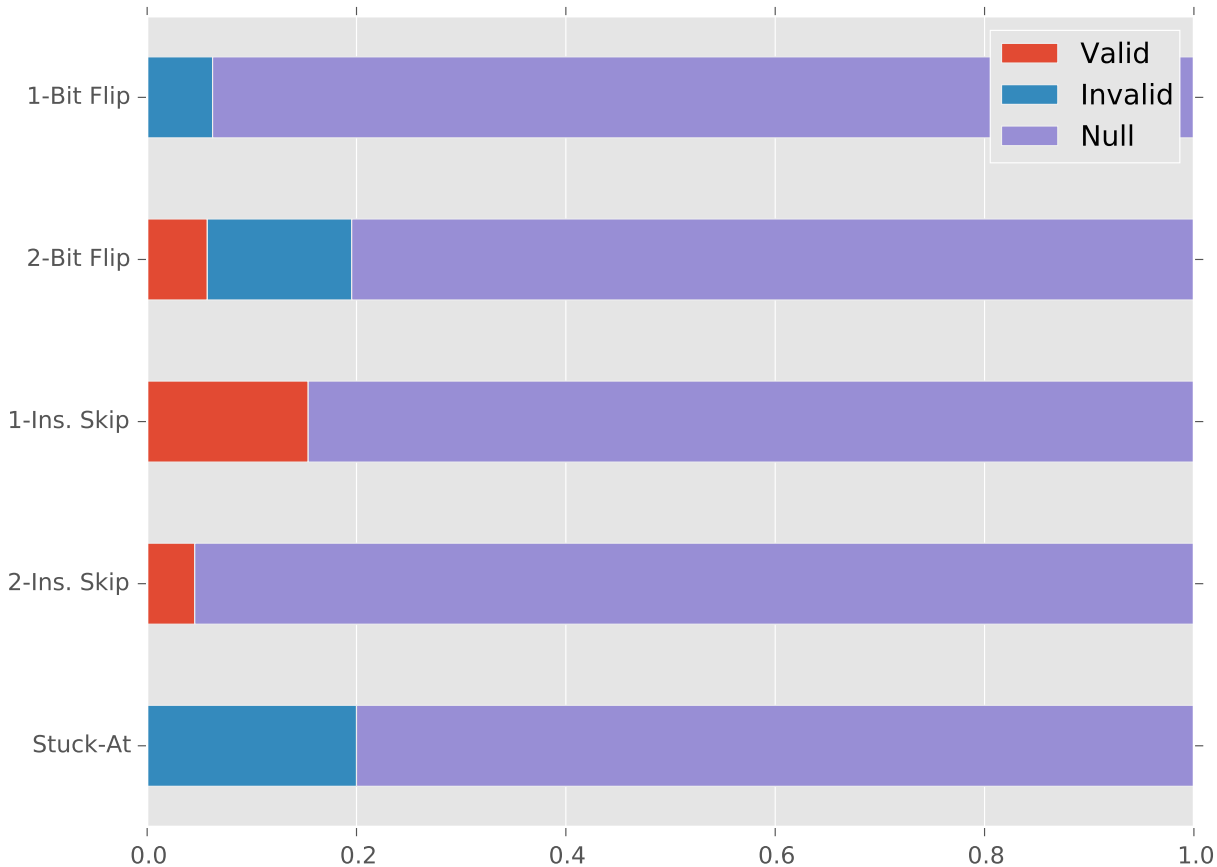


Experiment

VALID	32.4%
INVALID	47.7%
NULL	19.8%

DPL Implementation - Simulations and Experiment

Simulations



Experiment

VALID	22.2%
INVALID	54.7%
NULL	23.1%

Fault Propagation

- A *VALID* fault will always propagate to the output.
- Any *INVALID* or *NULL* input to *DPL XOR* and *Encoding* LUT will lead to a *NULL* at the output.
- *Encoding* XOR does propagate faults – there are several combinations of inputs that lead to *VALID* output.
- Also, a combination of *NULL* input and *VALID* input leaks information about the input.
- For instance, let first input be $A = 0x00$ and the second input be $B = 0xA5$. The output will be in the form $\bar{b_3}b_3\bar{b_2}b_2\bar{b_1}b_1\bar{b_0}b_0$, $b_i \in B$, therefore in this case it will result in $0x0F$.

Table of Contents

Hardening the *DPL* Implementation

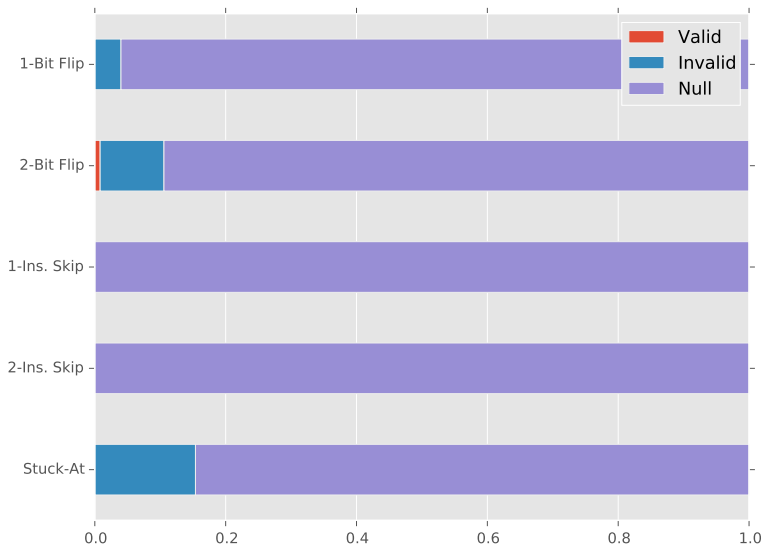
- For resisting single-instruction skips, we need to avoid two shift instructions, because these can produce *VALID* outputs in the case one of `LSL` instructions is skipped.
- We propose to use another look-up table to avoid these two instructions:

`inputs`	01	10
01	0101	0110
10	1001	1010

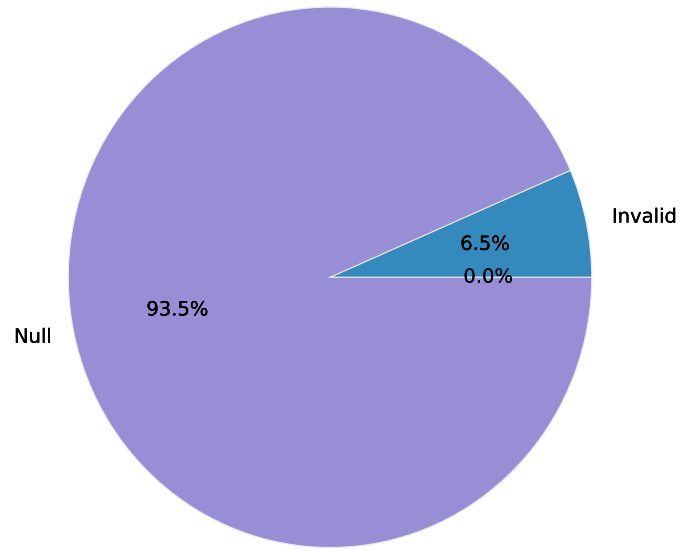
- To prevent double instruction skips, we added a partial redundancy.
- Clock cycle overhead: 11 $\rightarrow$ 19 (72.3%)

Countermeasure - Results

The only way to overcome the countermeasure is to inject double instruction skips on two instructions. Probability of successfully performing such a fault is 0.0048.



Simulations



Experiment

Table of Contents

Conclusion

- We have examined two software DPL proposals with respect to fault injection attacks – our results show weaknesses of these implementations.
- We simulated different fault models and validated our findings experimentally using laser fault injection station.
- Double bit flip fault model is the most effective one to disturb the algorithm while still producing a valid output.
- In comparison to hardware DPL, it takes significantly lower effort to disturb its software version using this fault model.
- We provided a countermeasure lowering the probability of a successful attack to 0.0048.

Thank you!

Any questions?