Instruction Duplication: Leaky and Not Too Fault-Tolerant!

Lucian Cojocar[1], Kostas Papagiannopoulos[2], and Niek Timmers[3]

[1] l.cojocar@vu.nl, Vrije Universiteit Amsterdam
[2] k.papagiannopoulos@cs.ru.nl, Radboud University Nijmegen
[3] timmers@riscure.com, Riscure – Security Lab

Abstract. Fault injection attacks alter the intended behavior of microcontrollers, compromising their security. These attacks can be mitigated using software countermeasures. A widely-used software-based solution to deflect fault attacks is *instruction duplication* and *n-plication*. We explore two main limitations with these approaches: first, we examine the effect of instruction duplication under fault attacks, demonstrating that as fault tolerance mechanism, code duplication does not provide a strong protection in practice. Second, we show that instruction duplication increases side-channel leakage of sensitive code regions using a multivariate exploitation technique both in theory and in practice.

1 Introduction

Fault Injection (FI) and Side-Channel Analysis (SCA) attacks are a risk for microcontrollers operating in a hostile environment where attackers have physical access to the target. These attacks can break cryptographic algorithms and recover secrets either by e.g changing the control flow of the program (FI) or by monitoring the device's power consumption (SCA) with little or no evidence.

Multiple countermeasures such as random delays [12], masking [42], infection [25], data redundancy checks [33, 35] and instruction redundancy [6] have been proposed to tackle these threats, yet their impact, effectiveness and potential interactions remain open for investigation. Such countermeasures can be implemented at hardware or at software level, often translating to overheads in silicon area and execution runtime. This exacerbates the need for a detailed analysis of the benefits introduced by these countermeasures before their actual deployment.

1.1 Motivation

In this work, we focus on the Instruction Duplication (ID) countermeasure, applied as a fault tolerance mechanism in software. The assembly-level redundancy introduced by ID can prevent attacks aiming to skip instructions and alter the control flow. Recent defenses (e.g., *infection* [19]) build further on code redundancy in order to provide a stronger protection.

Manually applying these defenses, however, does not scale well for a large code base that needs to be protected: it is an error-prone process and it costs many highly skilled man-hours, therefore, in practice, it is often automated using compiler techniques [8, 37, 32]. On top of protecting against fault attacks, compilers can also provide support to reduce the information leakage through side channels [4, 34, 39, 10, 9]. While there is previous work exploring the effect of one defense mechanism on another [41, 30, 5], to the best of our knowledge, the effect of ID on side-channel leakage has not been explored before. We perform an in-depth investigation of ID, focusing on its applicability against FI as well on its interaction with side-channel attacks.

Specifically, regarding fault attacks, the defender needs to exercise caution when applying ID, since the device may not adhere to the "single instruction skip" model. In such cases, the countermeasure is ineffective and we demonstrate that it can even benefit certain fault injection strategies. In addition, we highlight how even an effective application of ID can enhance our capability to perform side-channel attacks on the underlying implementation. Thus, we establish that care needs to be taken with respect to the equilibrium between fault injection defenses and side-channel resistance.

In the process of investigating these software defenses, we built the first open-source compiler capable of generating duplicated code for any C/C++ program. In this way, we hope to stimulate further research in this area.

1.2 Contribution

We summarize our contributions as follows:

- We experimentally determine that instruction skipping is not a realistic fault model for modern ARM Cortex-M4 MCUs.
- We develop and open source [4] an instruction duplication compiler for ARM Thumb2 architectures. To our knowledge, this is is the first time that such a compiler is publicly available.
- We examine the interaction between n-plication and side-channel resistance and demonstrate the trade-off using an information-theoretic approach. In addition, we show how horizontal exploitation techniques can leverage the side-channel introduced by ID-based defenses.
- We examine how the redundancy of infective countermeasures can interact with side-channel resistance and demonstrate how a Hidden Markov Model can render infection [19] equivalent to ID from a side-channel point-of-view.

This paper starts with the background (in Section 2) and with an overview of the related work in Section 3. Sections 4 and 5 investigate the limitations of the assumed FI model as well as the limits of compiler-based ID. In Sections 6 and 7 we determine the impact of hardening code with ID on SCA attacks. We summarize our findings in Section 8.

[4] The code is available at: `https://github.com/cojocar/llvm-iskip`

The final version will appear at Springer, in CARDIS 2017 proceedings.

2 Background

Software-based instruction redundancy methods for *fault detection* were proposed by Barenghi et al. [6]. In this technique, the original stream of instructions to be executed is duplicated (or even *triplicated*), one instruction after another, either manually or automatically [8, 38, 32].

For example a load from memory (`ldm r0, [r2, #0]`) is transformed by duplication in two loads originating from the same memory. To provide *fault detection* the destination registers must be different and then checked for differences (Listing 2). Under single instruction skip model, the *fault tolerance* arises when using the same register as destination. Indeed, skipping one single instruction from Listing 1 has the same effect as executing the original instruction.

```
ldr r0, [r2, #0]
ldr r0, [r2, #0]
```

Listing 1: Fault tolerance

```
ldr r0, [r2, #0]
ldr r1, [r2, #0]
cmp r0, r1
bne fault_detected
```

Listing 2: Fault detection

In practice, Moro et al. [37] showed that every ARM Thumb-1/2 instruction can be duplicated. We differentiate three classes of instructions: idempotent instructions, separable instructions and specific instructions. While the idempotent instructions are duplicable with no extra transformation, the other two classes often require an extra register to perform the duplication.

Therefore, on ARM Thumb-1/2, ID is generic and can be applied automatically regardless of the algorithm that the instruction stream implements.

Automatic deployment. Maebe et al. [32] apply ID for fault detection at link-time for the ARM architecture. Barry et al. [8] described a compiler able to produce duplicated instructions, however their tool is not publicly available.

Our LLVM based compiler emits duplicated instructions for the ARM Thumb2 instruction set. Through code annotations, the hardening can be enabled or disabled at function level, as instructed by the developer. The modified LLVM based compiler has a similar architecture as the implementation described by Bary et al. [8] and it can compile code in any language supported by Clang (e.g. C, C++) with different optimization levels, including the AES-128 implementation used in this paper. It is designed to be a drop-in replacement for any LLVM based toolchain. Due to space constraints we omit the implementation details. The compiler is available as an open-source project.

3 Related work

ID and the FI model. Moro et al. [38] practically evaluates instruction duplication as a defense for FI on a Cortex-M3 Microcontroller (MCU). They use electromagnetic (EMI) pulses to insert glitches and show the importance of the

fault model. Riviere et al. [45] show that the single instruction model is invalid when caches are enabled. The observed skip behavior, in the presence of an EMI glitch, is: the last 4 instructions are re-executed and 4 instructions are skipped — this partially invalidates the instruction duplication defense. Dureuil et al. [27] model the fault injection attack by including the EMI probe position. When an attack succeeds, the most probable outcome is to skip 1-4 instructions on a common smart card. They show that a probable outcome is the corruption to 0 of the destination operand of a `ld` instruction. Yuce et al. [48] show the effect of a single clock glitch on the ID scheme at clock granularity. They observe that the first instance of the instruction is corrupted and that its duplicated counterpart is transformed to a NOP instruction, thus defeating the ID. They use a 7-stage FPGA based implementation and clock glitches for experiments. Instead, we use a 3-stage pipeline off-the-shelf device and voltage glitches to investigate ID.

ID and SCA interaction. Regazzoni et al. [43] first looked at the interaction between fault injection defenses and Power Analysis (PA) attacks. Specifically, they studied an AES implementation with parity based error detection circuitry. They conclude that the presence of a parity error detection circuit will leak important information to an attacker through PA. One year later, Regazzoni et al. [44] experimentally show the exploitability of an known-by-the-attacker error detection circuit. Pahlevanzadeh et al. [40] look at three fault detection methods designed specifically for AES: double module redundancy, parity checks, inverse execution; all implemented on an FPGA. They find that parity checks are actually improving the resistance against standard Correlation Power Analysis (CPA). Similarly, Luo et al. [31] use CPA to attack an FPGA implementation of AES which is hardened for fault detection. They conclude that duplication does not improve the success rate of the attack in respect to the unhardened AES implementation. However, we stress that the approaches of [40, 31] use naive CPA attacks and do not rely on multivariate, horizontal exploitation of the leakage. Such attack-dependent techniques do not reveal the full picture and may lure the side-channel evaluator in a false sense of security.

4 FI preliminaries

Because ID and n-plication are defenses for faults, we experimentally evaluate them in a realistic fault injection scenario.

4.1 Fault injection background

Fault injection attacks change the intended behavior of a target by manipulating its environmental conditions. This can be accomplished using different fault injection techniques such as: *voltage FI*, *electromagnetic FI* and *optical FI*. In this paper we focus only on *voltage FI* where glitches are introduced in the voltage signal that powers the subsystem responsible for executing software. Voltage FI is easy to mount as it does not require sophisticated equipment and it is invasive.

FI model. Faults can target different physical layers of the device: single transistors, logic gates or computation units [47]. In this paper, we are interested in the observable effect of faults, namely, in faults that can cause a change in the program flow and that manifest at the instruction level. We note several types of faults in respect to instructions: single instruction skip [7], multiple instruction skip [45, 46], instruction re-execution [29, 45] and instruction corruption [46]. These types of faults are from now on referred to as the *fault model*.

Fault injection parameters. The following glitch parameters are important when performing voltage FI:

- the *Normal Voltage* is the voltage supplied to the target.
- the *Glitch Voltage* is the voltage *subtracted* from the *Normal Voltage* when the glitch is injected.
- the *Glitch Offset* is the time between when the trigger is observed and when the glitch is injected.
- the *Glitch Length* is the time for which the *Glitch Voltage* is set.

Finding the right parameters for a target is defined as *characterization*.

4.2 Experimental FI setup

Fault injection target. All fault injection experiments described in this section are performed targeting an off-the-shelf development platform built around an STM32F407 MCU. This MCU is implemented using 90nm technology and includes an ARM Cortex-M4 core running at 168 MHz. This Cortex-M4 based MCU has an instruction cache, a data cache and a prefetch buffer.

Related research used a similar experimentation target. Moro et al. [36, 38] used a development board designed around an 130nm technology MCU featuring an ARM Cortex-M3 core running at 56 MHz. The Cortex-M3 and Cortex-M4 are very similar and we expect the differences to have minimal impact. The latter includes additional specialized instructions which are not targeted in this paper. The pipeline size (3 stages) and the rest of the instruction set are the same.

To avoid instruction re-execution, which was shown to be possible by Rivier et al.[45], all experiments are performed with the prefetch buffer disabled and with caches enabled, unless otherwise stated.

Fault injection tooling. The voltage FI test bed is created using Riscure's VC Glitcher product[5] that generates an arbitrary voltage signal with a pulse resolution of 2 nanoseconds. Similarly to previous work, in a synthetic setup, we use a General Purpose Input Output (GPIO) signal to time the attack which allows us to inject a glitch at the moment the target is executing the targeted code. The target's reset signal is used to reset the target prior to each experiment to avoid data cross-contamination.

[5] https://www.riscure.com/security-tools/hardware/vc-glitcher

4.3 Fault injection characterization

We use the code snippet from Listing 3 for two purposes: (a) to find the glitch parameters (characterization) and (b) to invalidate the single instruction skip model for the target described in Section 4.2. The code is a copy-loop construction that is known to be a common target for fault injection because it has significant duration [46]. The targeted code is executed in a loop to minimize the impact of the *Glitch Offset* parameter as it does not matter what iteration of the loop and which part of the loop is hit.

```
0: ldm r0, {r4-r10}
   stm r0, {r4-r10}
   subs r1, #1
   bne 0b    //loop back
```

Listing 3: Characterization code

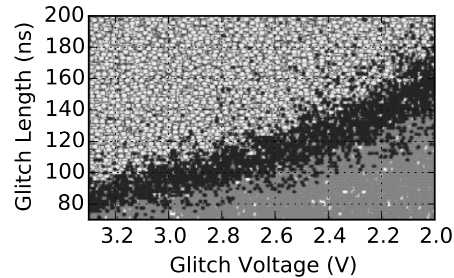


Fig. 1: Behavior under faults

The target's susceptibility to voltage FI attack is determined using the following glitch parameters: voltage $\in [-3.3V, -2.0V]$, offset $\in [2\mu s, 5\mu s]$ and length $\in [70ns, 200ns]$. The normal voltage is set to 3.3V. The results of FI experiments can be classified in three groups: *Expected*, *Successful* and *Mute*. The experiments are plotted in Figure 1 and show a clear relationship between the voltage and the length of the glitch. For the *Successful* experiments (black, the diagonal boundary) we observed a change in the target's behavior without affecting its continuation. For all *Mute* experiments (light gray, above the diagonal) the target halted or performed a reset. The *Expected* experiments (dark gray, below diagonal) are the ones for which we did not observe a change in the execution.

Instruction corruption model. Executing under faults the code from Listing 3 yields the following result: the memory pointed by `r0` after the loop is different that its contents before the loop (*Successful*). If only the instruction skip fault model applies to the target, then the memory pointed by `r0` should be the same as before the loop executes (*Expected*). We ran the experiment 20K times and, in 15.91% (SE=25x10^{-4}) of cases were *Successful*. In 65.59% (SE=33x10^{-4}) of cases the device crashed or failed to answer and, the rest of the cases were *Expected*. The standard error (SE) is computed as $\sqrt{P * (1 - P)/N}$, where N is the number of experiments (20K in this case) and P is the success rate.

The non-negligible number of *Successful* cases indicates that the target adheres to a more complex fault model than single instruction skip model – i.e. the *instruction corruption* model. We say the *instruction corruption* fault model holds *iff* the observed behavior of the target under faults cannot always be explained by removing one (or multiple) instructions from the execution stream. This loose definition captures as well the data corruption fault model.

The final version will appear at Springer, in `CARDIS 2017` proceedings.

	original	$n=2$ (ID)	$n=3$	$n=4$	$n=5$
SR (%)	15.91	15.61	11.59	13.5	11.96
SE (x10^{-4})	25	25	22	24	22

Table 1: Success rate of FI and n-plication levels

5 Fault injection effectiveness

In this section, we practically evaluate ID under instruction corruption FI model.

5.1 Inaccuracies in the FI model

We resort to two experiments, that show how ID can negatively affect the fault tolerance of ID if a different model than single instruction skip holds. Furthermore, we show that when applying ID the runtime configuration of the target must be considered.

ID and the "real" FI model. We determine the impact of ID by *duplicating* and n-plicating code from Listing 3. For each code instance, we perform 10K experiments, using the glitch parameters outlined in Section 4.3.

Table 1 shows that ID does not provide fault tolerance for software for our target. Even if the instruction is n-plicated three times or more, the fault tolerance is not substantially improved. Because we use a real target with no access to low level hardware features (i.e. flip-flop states), we do not aim to detail the root cause of this behavior. Instead, we note that the instruction corruption model captures this result.

Limitations of a static FI model. When ID is deployed automatically at compile time, the compiler is not aware of the runtime configuration (e.g. cache configuration). In this experiment, we show how ID and n-plication affects the success of FI when several runtime configurations are used.

In Figure 2 we enable and disable the prefetch buffer (`p`), the instruction cache (`i`) and the data cache (`d`) and plot the fault injection success rate on the code similar to Listing 5. A capital letter in the title of the subplot means that the specific feature is enabled. We use the color scheme defined in Section 4.3.

Because the data on which our test operates is stored in registers, toggling the data cache has no impact on the fault tolerance. However, we observe four interesting results. First, ID increases the probability of a successful fault when the device is used with all its functionality enabled (`PID`). In this case, n-plication with $n=3$ and $n=4$ has the highest fault tolerance. Second, when all features are disabled (`pid`), none of the n-plication level improve the fault tolerance. Thirdly, when the instruction cache is disabled, enabling the prefetch buffer makes ID the most effective amongst the n-plication levels (`pid`, `piD` vs. `Pid`, `PiD`). Finally, comparing the right-most four subplots with the left-most subplots, the instruction cache offers an improved resilience against voltage glitches.

As a consequence, the compiler must be aware of the runtime configuration of the device when it emits redundant instructions.

The final version will appear at Springer, in CARDIS 2017 proceedings.

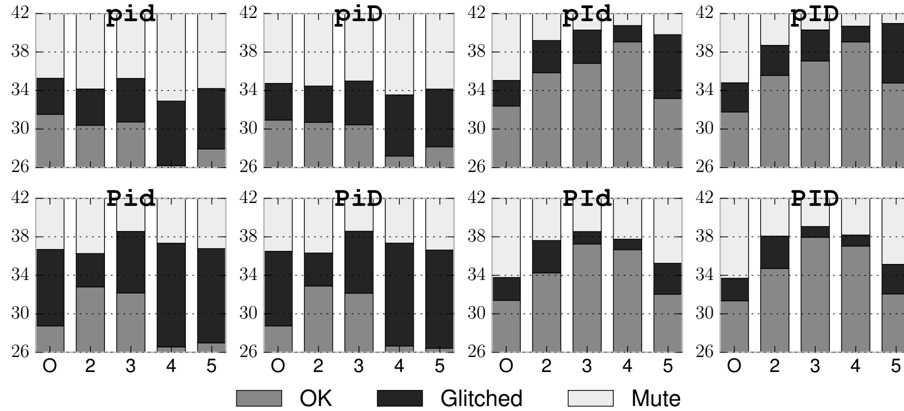


Fig. 2: SR of faults vs. multiple n-plication levels and runtime configurations

5.2 Impact of compiler techniques

We now explore two compiler techniques that affect the effectiveness of ID.

Register allocation pressure. Register Allocation (RA) is the process in which the compiler maps the *virtual* (unlimited) registers to physical (limited) registers. This process is highly optimized to yield the best space and runtime performance. In this section we show that the modified register allocation scheme that ID requires has a negative impact on the fault tolerance.

```
add r5, r5, #1
add r7, r7, #1
```

Listing 4: Registers are incremented

```
add r4, r5, #1
mov r5, r4
add r6, r7, #1
mov r7, r6
```

Listing 5: Code ready for duplication

Listing 5 is the transformation of the code from Listing 4 with the `add` being replaced by an idempotent sequence that uses an extra temporary register (see Section 2). We define a successful glitch with respect to the contents of the registers `r5` and `r7`. If the contents of the registers is different than what is expected (i.e. the number of iterations added to the initial value of the registers) then we count this trial as a success. Otherwise, the glitch was not inserted or the parameters caused a *mute*.

The ID aware RA yields a higher success rate for FI (SR=18.64%, SE=$20\text{x}10^{-5}$) than the unmodified one (SR=10.63%, SE=$16\text{x}10^{-5}$). Apart from runtime performance degradation, the increased register pressure induced by the custom RA has a two fold negative impact on the fault tolerance. First, it increases the probability of a register to be *spilled* on the stack. As a consequence, the compiler will likely chose complex multi-memory access operations over simple load or stores. The multi-memory operations (e.g. `ldm`, `stm`) are more prone to faults than single memory operations [46] or than register to register operations. Second, an extra instruction to write back the result is needed (`mov`). This extra

instruction is duplicated, therefore it increases the window in which a fault can be injected and it adds another leakage point.

In short, not only does the ID register allocation works against the established RA optimizations, but it also has a negative effect on the fault tolerance guarantees. This is a fundamental limitation of ID.

Instruction ordering. The compiler has the freedom to emit instructions in any order. This is done either for optimization purposes (e.g. benefit from a multi-stage pipeline) or to avoid a certain illegal order of instructions. Barry et al. [8] showed that the correct scheduling of duplicated instructions can reduce the runtime overhead of the duplicated code, from 2.14X down to 1.70X-2.09X on a software AES implementation. Yuce et al. [48] hint at the interaction between ID and the processor pipeline.

To analyze what is the impact of the instruction order on the success rate of injected faults we compare the success rate of the code Listing 6 and its possible scheduled version Listing 7. We define a successful trial whenever the memory pointed by `r6` is different than its initial value. Our results show that instruction scheduling *decreases* the success rate of injecting a fault, from 8.51 % to 4.00%.

Intuitively, the pipeline for Listing 6 contains the protected instruction and its copy right after another. Therefore, the chances that a fault affects the protected instruction and its copy at a given clock cycle is higher than in the case when the protected instruction and its copy are one (or more) instruction apart (Listing 7). These results are in line with the work of Yuce et al. [48], which shows that ID can be bypassed with a single glitch because multiple instructions are in the pipeline at a given clock cycle.

When emitting duplicated code the order is important, yet to date a FI model that captures the order interaction does not exist, let alone a compiler that uses this model. We leave the design of such a model and compiler as future work. We conclude that compiler optimization techniques (e.g. instruction scheduling, register allocation optimality) interact with the fault tolerance guarantees of ID.

```
add r0, r4, r1
add r0, r4, r1
ldr r5, [r6, #0]
ldr r5, [r6, #0]
```

Listing 6: Natural order

```
add r0, r4, r1
ldr r5, [r6, #0]
add r0, r4, r1
ldr r5, [r6, #0]
```

Listing 7: Possible re-ordering

6 SCA of ID and Infection Countermeasures

This section demonstrates the interactions between the redundancy-based FI countermeasures and the side-channel resistance of an implementation that is employing them. In Section 6.1 we analyze the theoretical effect of ID and n-plication on SCA using an information-theoretic approach. Section 6.2 demonstrates how to perform SCA on infective countermeasures using a Hidden Markov

Model that simplifies the exploitation phase of infection to that of ID. Throughout this section, capital letters denote random variables and small case letters denote instances of random variables or constants. Bold letters denote vectors.

6.1 Information-Theoretic Evaluation of ID for SCA

From a side-channel perspective, the ID countermeasure increases the available leakage in a horizontal manner, either as a fault detection or as a fault tolerance mechanism. Analytically, in the case of an unprotected implementation (without ID) a univariate adversary can acquire the leakage of a key-dependent value v, i.e. observe $L_v \sim \mathcal{N}(v, \sigma)$, assuming identity leakage model. On the contrary, when instruction n-plication is implemented ($n > 1$), the adversary can observe over time an n-dimensional leakage vector $\mathbf{L}_v = [L_v^{t=1}, \ldots, L_v^{t=n}]$. The vector contains n independent observations of value v under the same noise level, i.e. we assume that $L_v^t \sim \mathcal{N}(v, \sigma)$, $t = 1, \ldots, n$.

Given that the side-channel adversary has located the sample positions of the repeated leakages, he can perform a pre-processing step where he averages all available samples that leak v, i.e. he computes $\bar{L}_v = (1/n) * \sum_{t=1}^{n} L_v^t$. The averaging step results in noise reduction of factor $\sqrt{n}$, obtaining $\bar{L}_v \sim \mathcal{N}(v, \sigma/\sqrt{n})$ and as a result side-channel attacks can be enhanced. Note that noise reduction can be particularly hazardous even when additional side-channel protection is implemented. For instance, both masking and shuffling countermeasures [13, 14] amplify the existing noise of a device and will perform poorly if the noise level has been reduced by a large factor $\sqrt{n}$. In order to demonstrate the effect of noise reduction, we employ the information-theoretic framework of Standaert et al. [13] which evaluates the resistance against the worst possible attack scenario. The MI between the key-dependent value V and leakage $\mathbf{L}_v$ can be computed using the following formula: $MI(V; \mathbf{L}_v) = H[V] + \sum_{v \in \mathcal{V}} Pr[v] \cdot \int_{\mathbf{l}_v \in \mathcal{L}^n} Pr[\mathbf{l}_v|v] \cdot \log_2 Pr[v|\mathbf{l}_c] \, d\mathbf{l}_v$, where $Pr[v|\mathbf{l}_v] = \frac{Pr[\mathbf{l}_c|v]}{\sum_{v^* \in \mathcal{V}} Pr[\mathbf{l}_v|v^*]}$.

From Figure 4 we derive the following three conclusions. First, we observe that n-plication (for $n > 1$) shifts the MI-curve to the right, i.e. the FI countermeasure produces repeated leakages which have a direct impact on the side-channel security of the implementation. Second, we note that if ID translates to more than two assembly instructions that manipulate the same value, we will likely observe even more hazardous repetitions. Third, it follows that a countermeasure designer needs to balance the need for side-channel resistance and FI resistance by fine-tuning the parameter n.

6.2 Converting Infection to ID for SCA

It is important to point out that, apart from straightforward instruction duplication, a wide variety of FI countermeasures rely on some form of spatio-temporal redundancy. For instance, detection methods such as full/partial/encrypt-decrypt duplication & comparison of a cipher [21] produce repetitions of intermediate

values that are exploitable by the side-channel adversary. Thus, an MI-based evaluation of duplication & comparison is identical to Figure 4. Similarly, countermeasures that rely on particular error detection/correction codes [22] also introduce redundancy that has been evaluated in the side-channel context by Regazzoni et al. [26].

In this section, we expand in the same direction and examine the interaction between side-channel analysis and the more recent infective countermeasure [19][6]. Specifically, we demonstrate how the application of a Hidden Markov Model (HMM) [2, 16] in a low-noise setting can render infective countermeasures equivalent to ID from a side-channel point-of-view.

Infective countermeasures were developed as a solution to the vulnerabilities of the duplicate & compare methods [25]. Instead of vulnerable comparisons, infection diffuses the effect of faults in order to make the ciphertext unexploitable. In particular, we focus on the infective countermeasure of Tupsamudre et al. [19], which has been proven secure against DFA [24], given that the adversary cannot subvert the control flow and that certain fault models are not applicable [20]. The countermeasure is shown in Algorithm 1.

The infective countermeasure alternates between real, redundant and dummy cipher rounds (step 8). It requires an r bit random number $rstr$ (step 3), consisting of $2n$ 1's that trigger computation rounds (redundant or real) and $(r-2n)$ 0's that trigger dummy rounds (steps 5-7). In the event of FI, the difference is detected via function $BLFN : size(R) \rightarrow 1$, where $BLFN(0) = 0$ and $BLFN(x) = 1, \forall x \neq 0$. The error is propagated via step 11.

From a side-channel perspective, the infective countermeasure can be viewed as a random sequence of r round functions, where only the $2n$ computation rounds are useful for exploitation. Thus, the objective of the side-channel adversary is to uncover the hidden sequence of rounds and to isolate the useful ones. Subsequently, one can exploit e.g. the first redundant and first real round together via averaging, which is identical to the afore-mentioned exploitation of ID. Distinguishing effectively dummy rounds from computational ones is nontrivial, especially when extra randomization steps are involved [23]. However, the presence of control logic in the infective countermeasure such as variables λ, ζ and κ can emit noisy side-channel information about the sequence of rounds. We model such leakage as $\mathbf{L}_c = [\Lambda, Z, K] + \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$, where the deterministic part $[\Lambda, Z, K]$ is defined over $\{0,1\}^3$ and $\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ denotes 3-dimensional noise vector with zero mean and diagonal covariance matrix $\mathbf{\Sigma}$.

The suggested HMM is constructed the following way. We encode the main loop of Algorithm 1 using two states, i.e. at a given time t, the state $s_t = i \in \{C, D\}$, where C corresponds to a computational round and D to a dummy round. The transitions in the sequence of states is described by matrix T, where $T_{i,j} = Pr(s_{t+1} = j | s_t = i)$. Figure 3 shows the state diagram and the probabilities for matrix T, namely $p = 2n/r$. We note that it is possible to unroll the loop and use additional states to describe the transitions, such that we can fine-tune

[6] Infective countermeasures in this [19] work do not pertain to the modular arithmetic infective techniques used by Rauzy et al. [3]

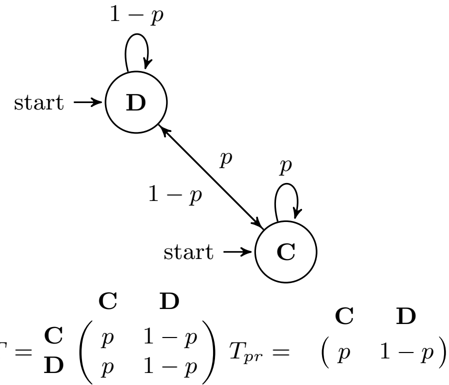
the probabilities. However, we opt for such simple representation to minimize the model's data complexity.

Algorithm 1: INFECTION TUPSAMUDRE ET AL. [19]

Input: Plaintext P, key K, round j key k^j, $\forall j = 1, \ldots, n$, n number of rounds, dummy plaintext β, dummy round key k^0

Output: Ciphertext C=Cipher(P, K)

1. Real $R_0 \leftarrow P$, Redundant $R_1 \leftarrow P$, Dummy $R_2 \leftarrow \beta$
2. $i \leftarrow 1$
3. $rstr \in_R \{0,1\}^r$ `//r random bits`
4. **for** $q = 1$ *until* r **do**
5. $\quad \lambda \leftarrow rstr[q]$
6. $\quad \kappa \leftarrow (i \wedge \lambda) \oplus 2(\neg\lambda)$
7. $\quad \zeta \leftarrow \lambda \lceil i/2 \rceil$
8. $\quad R_k \leftarrow RoundFunction(R_k, k^{\zeta})$
9. $\quad \gamma = \lambda(\neg(i \wedge 1)) \cdot BLFN(R_0 \oplus R_1)$
10. $\quad \delta \leftarrow (\neg\lambda) \cdot BLFN(R_2 \oplus \beta)$
11. $\quad R_0 \leftarrow (\neg(\gamma \vee \delta) \cdot R_0) \oplus ((\gamma \vee \delta) \cdot R_2)$
12. $\quad i \leftarrow i + \lambda$
13. $\quad q \leftarrow q + 1$
14. **end**
15. **return** R_0



$$T = \begin{array}{c} \\ \mathbf{C} \\ \mathbf{D} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{C} & \mathbf{D} \end{array} \\ \begin{pmatrix} p & 1-p \\ p & 1-p \end{pmatrix} \end{array} \quad T_{pr} = \begin{array}{c} \begin{array}{cc} \mathbf{C} & \mathbf{D} \end{array} \\ \begin{pmatrix} p & 1-p \end{pmatrix} \end{array}$$

Fig. 3: The Markov model describing the states, transition probabilities T and prior probabilities T_{pr}.

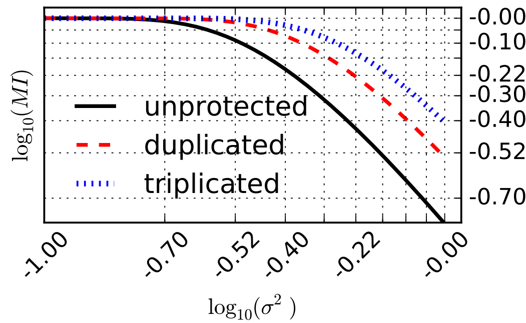


Fig. 4: MI of instruction n-plication

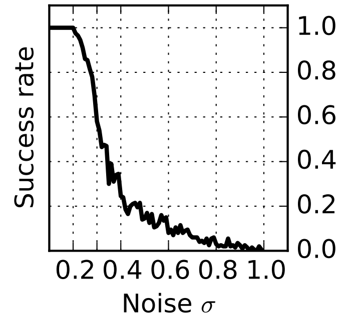


Fig. 5: Success rate of HMM-based sequence detection vs. noise level σ

In the HMM, the round sequence $\mathbf{s} = [s_1, \ldots, s_r]$ is unknown, but the adversary is assisted by leakage observations $[\mathbf{l}_c^{t=1}, \ldots, \mathbf{l}_c^{t=r}]$. To exploit the observations, the HMM associates every state $i \in \{C, D\}$ with an estimated emission probability function, i.e. emission $e_i(\mathbf{l}_c^t) = Pr(\mathbf{l}_c^t | s_t = i)$.

Having established the HMM for our scenario, we perform a simulated experiment where we try to identify the round sequence for a gradually increasing

The final version will appear at Springer, in CARDIS 2017 proceedings.

noise level. The simulated sequence contains 22 computational rounds and 78 dummy rounds, i.e. it corresponds to a computation of AES-128 using infection with $r = 100$. For every noise level we apply the Viterbi algorithm [1], which can recover the most probable sequence $\mathbf{s}$ of length r, while factoring in the leakage observations $\mathbf{l}_c^{t=1\ldots r}$ and the transition probabilities of T. The simulation (Figure 5) shows that for fairly small noise levels (e.g. $\sigma < 0.3$) we are able to uncover the hidden sequence with high probability, making the side-channel exploitation of infection equivalent to the exploitation of instruction duplication.

7 Practical SCA Results

In this section, we apply the exploitation techniques of Section 6.1 in our experimental setup that protects an AES-128 implementation using ID. We verify the technique's applicability to real-world scenarios by showing their increased efficiency compared to standard SCA methods. We use an AVR MCU (XMEGA128D4) as the main target for our SCA experiments and we collect power traces using the open-source ChipWhisperer product[7]. The clock frequency of the target is 7.3728 MHz and we sample the power consumption of the target 4 times per clock cycle.

We use three different code patterns to evaluate the interaction between SCA and ID in different scenarios. Patterns Ⓐ and Ⓑ demonstrate how ID affects different instructions, namely instructions `eor` and `ld` respectively. Pattern Ⓒ showcases the duplicated key addition and Sbox parts of a lookup-table-based AES implementation.

Ⓐ	`eor r10, r17`
Ⓑ	`ld r10, Y`

Ⓒ	`eor r9, r17`
	`add r28, r9`
	`ld r10, Y`

Fig. 6: Code snippets for the SCA experiments. `Y` is the output buffer and `r17` contains the hardcoded secret key.

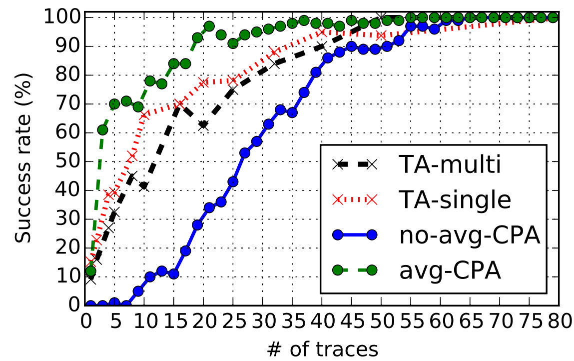


Fig. 7: CPA vs. Template on Ⓒ

7.1 Horizontal Exploitation using CPA

For the afore-mentioned patterns, we perform an experimental evaluation where we put forward a variant of the traditional Correlation Power Analysis (CPA) [11]. In the case of n-plication, we involve a horizontal averaging pre-processing strategy as follows.

[7] https://newae.com/tools/chipwhisperer/

The final version will appear at Springer, in CARDIS 2017 proceedings.

1. Locate the intervals pertaining to the n different repeated leakages. In every interval, heuristically select the point in time with the highest correlation to the targeted key-dependent value, obtaining vector $\mathbf{l} = [l^{t=1}, \ldots, l^{t=n}]$.
2. For every vector $\mathbf{l}$ compute the average value $\bar{l} = (1/n) * \sum_{t=1}^{n} l^t$, thus reducing the noise level.
3. Perform CPA using the averaged values ($\bar{l}$).

In Figure 8, we observe how the averaged CPA using a Hamming weight model outperforms naive CPA that ignores horizontal leakage, since it requires less traces to converge. Thus, the theoretical results of Section 6.1 are confirmed in practice and we conclude that horizontal averaging rejects noise. In addition, the difference between the naive CPA on the original code and averaged CPA on the duplicated code is larger on the duplicated `eor` pattern rather than on the duplicated `ld`. This behavior is attributed to the SNR of `ld/st` instructions, which is significantly higher compared to the SNR of ALU operations (such as `eor`)[8], since the later do not manipulate the memory bus. As a result, there is less need to reject noise on memory instructions. Last, we observe that a naive CPA attack when ID is in place may be slower to converge due to interference between duplicated consecutive instructions.

This work focuses on n-plication used as a fault tolerance mechanism, the same averaging technique can be applied when n-plication is used as a fault detection mechanism. In the latter case the instruction stream is the same as in the former case when no faults are injected, therefore, the side channel is similarly amplified.

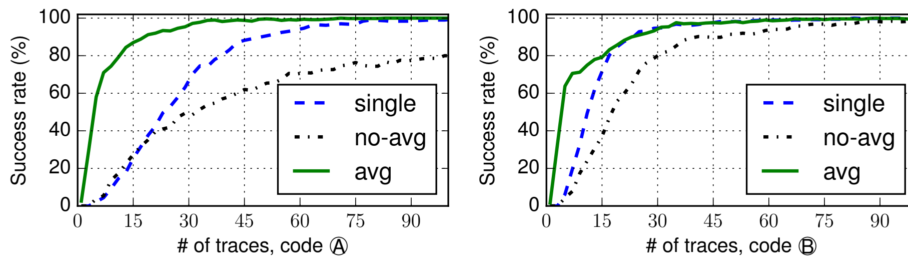


Fig. 8: Success rate of the CPA attack. *single* is CPA on the original code. On duplicated code, *no-avg* is the naive CPA and *avg* is the CPA with averaging.

7.2 Horizontal Exploitation using Templates

In order to fully exploit the available horizontal leakage, we also use a template-based approach [18, 15], which comprises two phases for attacking an AES-128 implementation: a profiling phase, in which templates are built for 256 key candidates of an AES-128 key byte and an extraction phase, where a number of traces are used to recover the unknown key. In our experiments, for the profiling

[8] SNR(Ⓐ)=2.23 and SNR(Ⓑ)=18.20

The final version will appear at Springer, in CARDIS 2017 proceedings.

phase, we use 3.2k traces of the device per key candidate and perform dimensionality reduction, selecting Points of Interest (POIs) via Principal Component Analysis [17]. We deployed the following two template attacks. To ensure that the side-channel effect of ID is exploited during the heuristic step of POI selection, the first attack breaks the trace in multiple intervals, each containing a single assembly instruction and performs POI selection in every interval separately. The second template attack considers the full trace as a single interval and performs POI selection in the whole region.

In Figure 7, we focus on code pattern Ⓒ. We perform the CPA attack (naive and averaged) that exploits the duplication of the `ld` instruction computing the Sbox output. Moreover, we perform the multi-interval and single-interval template attacks. We observe that both template attacks achieve similar performance and surpass the averaged CPA. Thus, we verify the applicability of templates in a horizontal context and conclude that they constitute an optimized way to exploit repeated leakages. We note that template attacks are inherently multivariate and may often require an extensive profiling phase to effectively characterize the model. On the other hand, averaged CPA compresses multiple samples, i.e. it is a univariate technique with a less informative model compared to templates, yet it has the upside of being faster to train and compute.

8 Conclusion

In this paper we analyzed the limitations of Instruction Duplication (ID) as a fault tolerance mechanism. First, we proved that the model under which ID operates has fundamental limitations, rendering the ID ineffective or even harmful. ID is designed under the assumption of a single fault model. However, in practice a more complex model can hold for a specific target, thus relying only on ID as a fault tolerance mechanism is not effective against FI attacks.

Second, the information leakage through side channel is amplified. We showed that the side channel introduced by instruction by ID, can be successfully exploited to extract secret information. Moreover, other instruction redundancy based defenses suffer from the same weaknesses in respect to side channels.

Finally, while automatically applying redundancy based defenses is promising, the FI model has to be fine tuned and extended for each targeted device according to its runtime configuration. The compiler must use this model to carefully balance fault tolerance guarantees and performance. Whether or not this is possible is still an open question.

9 Acknowledgements

This research was supported by the NWO CYBSEC "OpenSesame" project (628.001.005) and the NWO project ProFIL (628.001.007). We thank our anonymous reviewers and our shepherds, Fischer Jean-Bernard and Romailler Yolan for their invaluable feedback. We also thank Marius Schilder and Dominic Rizzo from Google Inc. for their support in developing the compiler.

The final version will appear at Springer, in `CARDIS 2017` proceedings.

References

[1] A. Viterbi. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE Transactions on Information Theory* 13.2 (Apr. 1967), pp. 260–269. ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1054010.

[2] L. R. Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE* 77.2 (Feb. 1989), pp. 257–286. ISSN: 0018-9219. DOI: 10.1109/5.18626.

[3] P. Rauzy and S. Guilley. “Countermeasures against High-Order Fault-Injection Attacks on CRT-RSA”. In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. Sept. 2014, pp. 68–82. DOI: 10.1109/FDTC.2014.17.

[4] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. “Automated Instantiation of Side-Channel Attacks Countermeasures for Software Cipher Implementations”. In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF ’16. Como, Italy: ACM, 2016, pp. 455–460. ISBN: 978-1-4503-4128-8. DOI: 10.1145/2903150.2911707.

[5] Frederic Amiel et al. “Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis”. In: *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop On*. IEEE, 2007, pp. 92–102.

[6] Alessandro Barenghi et al. “Countermeasures against Fault Attacks on Software Implemented AES: Effectiveness and Cost”. In: *Proceedings of the 5th Workshop on Embedded Systems Security*. ACM, 2010, p. 7. URL: http://dl.acm.org/citation.cfm?id=1873555 (visited on 10/14/2016).

[7] Alessandro Barenghi et al. “Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures”. In: *Proc. IEEE* 100.11 (2012), pp. 3056–3076.

[8] Thierno Barry, Damien Couroussé, and Bruno Robisson. “Compilation of a Countermeasure Against Instruction-Skip Fault Attacks”. In: *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*. ACM, 2016, pp. 1–6. URL: http://dl.acm.org/citation.cfm?id=2858931 (visited on 10/14/2016).

[9] Ali Galip Bayrak et al. “A First Step Towards Automatic Application of Power Analysis Countermeasures”. In: *Proceedings of the 48th Design Automation Conference*. DAC ’11. San Diego, California: ACM, 2011, pp. 230–235. ISBN: 978-1-4503-0636-2. DOI: 10.1145/2024724.2024778.

[10] Ali Galip Bayrak et al. “Automatic Application of Power Analysis Countermeasures”. In: *IEEE Transactions on Computers* (2015).

[11] Eric Brier, Christophe Clavier, and Francis Olivier. “Correlation Power Analysis with a Leakage Model”. In: *CHES ’04*.

[12] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. “Differential Power Analysis in the Presence of Hardware Countermeasures”. In: *CHES ’00*.

The final version will appear at Springer, in CARDIS 2017 proceedings.

[13] François-Xavier Standaert et al. “The World Is Not Enough: Another Look on Second-Order DPA”. In: *ASIACRYPT ’10*.

[14] Nicolas Veyrat-Charvillon et al. “Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note”. In: *ASIACRYPT ’12*.

[15] Omar Choudary and Markus G. Kuhn. “Efficient Template Attacks”. In: *CARDIS ’13*.

[16] François Durvaux et al. “Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models”. In: *CARDIS ’12*.

[17] Cédric Archambeau et al. “Template Attacks in Principal Subspaces”. In: *CHES ’06*.

[18] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *CHES ’02*.

[19] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. “Destroying Fault Invariant with Randomization - A Countermeasure for AES Against Differential Fault Attacks”. In: *CHES ’14*.

[20] Alberto Battistello and Christophe Giraud. “A Note on the Security of CHES 2014 Symmetric Infective Countermeasure”. In: *COSADE ’16*.

[21] Victor Lomné, Thomas Roche, and Adrian Thillard. “On the Need of Randomness in Fault Attack Countermeasures - Application to AES”. In: *FDTC ’12*.

[22] Tal Malkin, François-Xavier Standaert, and Moti Yung. “A Comparative Cost/Security Analysis of Fault Attack Countermeasures”. In: *FDTC ’06*. 2006.

[23] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. “Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output”. In: *LATINCRYPT*. 2012.

[24] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. “Fault Tolerant Infective Countermeasure for AES”. In: *SPACE ’15*.

[25] Marc Joye, Pascal Manet, and Jean-Baptiste Rigaud. “Strengthening hardware AES implementations against fault attacks”. In: *IET Information Security* (2007).

[26] Francesco Regazzoni et al. “Interaction Between Fault Attack Countermeasures and the Resistance Against Power Analysis Attacks”. In: *Fault Analysis in Cryptography. ’12*.

[27] Louis Dureuil et al. “From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference”. In: *CARDIS ’15*.

[28] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. “Differential Fault Analysis on AES”. In: *International Conference on Applied Cryptography and Network Security*. Springer, 2003, pp. 293–306.

[29] Thomas Korak et al. “Clock Glitch Attacks in the Presence of Heating”. In: *FDTC ’14*.

[30] Yang Li et al. “Fault Sensitivity Analysis”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2010, pp. 320–334.

The final version will appear at Springer, in `CARDIS 2017` proceedings.

[31] Pei Luo et al. "Side-Channel Power Analysis of Different Protection Schemes against Fault Attacks on AES". In: *ReConfig '14*.

[32] Jonas Maebe et al. "Mitigating Smart Card Fault Injection with Link-Time Code Rewriting: A Feasibility Study". In: *International Conference on Financial Cryptography and Data Security '13*.

[33] Paolo Maistri and Régis Leveugle. "Double-Data-Rate Computation as a Countermeasure against Fault Analysis". In: *IEEE Trans. Comput. 57*.

[34] Pedro Malagón et al. "Compiler Optimizations as a Countermeasure against Side-Channel Analysis in MSP430-Based Devices". In: *Sensors* 12.6 (2012).

[35] Marcel Medwed and Jörn-Marc Schmidt. "A Generic Fault Countermeasure Providing Data and Program Flow Integrity". In: *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop On*. IEEE, 2008, pp. 68–73.

[36] Nicolas Moro et al. "Electromagnetic Fault Injection: Towards a Fault Model on a 32-Bit Microcontroller". In: *FDTC '13*.

[37] Nicolas Moro et al. "Formal Verification of a Software Countermeasure against Instruction Skip Attacks". In: *J. Cryptogr. Eng.* (2014).

[38] Nicolas Moro et al. "Experimental Evaluation of Two Software Countermeasures against Fault Attacks". In: *HOST '14*.

[39] Andrew Moss et al. "Compiler Assisted Masking". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 58–75.

[40] Hoda Pahlevanzadeh, Jaya Dofe, and Qiaoyan Yu. "Assessing CPA Resistance of AES with Different Fault Tolerance Mechanisms". In: *ASP-DAC '16*.

[41] Sikhar Patranabis et al. "One Plus One Is More than Two: A Practical Combination of Power and Fault Analysis Attacks on PRESENT and PRESENT-like Block Ciphers". In: *FDTC'17*. IEEE, 2017.

[42] Emmanuel Prouff and Matthieu Rivain. "Masking against Side-Channel Attacks: A Formal Security Proof". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 142–159.

[43] Francesco Regazzoni et al. "Power Attacks Resistance of Cryptographic S-Boxes with Added Error Detection Circuits". In: *DFT '07*.

[44] Francesco Regazzoni et al. "Can Knowledge Regarding the Presence of Countermeasures against Fault Attacks Simplify Power Attacks on Cryptographic Devices?" In: *DFT '08*.

[45] Lionel Riviere et al. "High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures". In: *HOST '15*.

[46] Niek Timmers, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM Using Fault Injection". In: *FDTC '16*.

[47] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. "The Fault Attack Jungle-a Classification Model to Guide You". In: *FDTC '11*.

[48] Bilgiday Yuce et al. "Software Fault Resistance Is Futile: Effective Single-Glitch Attacks". In: FDTC '16, 2016, pp. 47–58.

The final version will appear at Springer, in `CARDIS 2017` proceedings.

Appendix

Differential Fault Analysis (DFA) attack on software AES-128

In Section 5 we determined the impact of ID as a fault tolerance mechanism on synthetic code. Now we show the interaction between ID and the number of trials needed to conduct a fault based attack. To this extent, we automatically apply ID on a large and complex code construction, the *AES-128* cryptographic algorithm, and perform the DFA attack described by Dusart et al. [28]. The goal of the attack is to extract the fixed key by observing the faulty output.

We use the tiny-AES128-C[9] implementation of the AES-128 cipher, in ECB mode for our target to encrypt a fixed input with a fixed key. A trigger is implemented between the 9th and the 10th round to guarantee we always hit the right location within the algorithm. Two versions of the AES-128 implementation are compiled: a *hardened* version (with ID in place) and an *non-hardened* version.

A 2K trace set containing traces with faulty outputs is acquired for each implementation. We randomly select n_t from these trace sets and use them in the DFA attack. We repeat this process 100 times for each implementation and we plot how often the attack is successful in Figure 9.

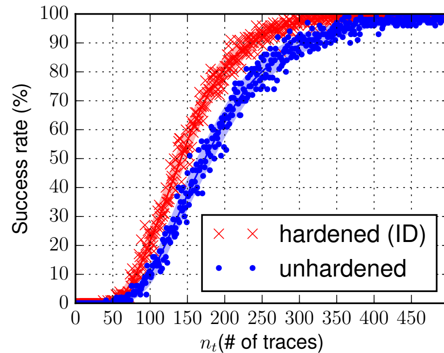


Fig. 9: DFA on AES-128

	3 or less	4	5 or more
hardened (ID)	0.2%	64.0%	35.7 %
unhardened	1.1%	41.5%	57.4 %

Table 2: Bytes changed in the output

The non-hardened implementation outperforms the hardened implementation in terms of FI tolerance. A clear indication that ID is not effective for protecting the AES-128 algorithm when the instruction corruption fault model holds. Depending on the time penalty required for a single experiment, the small difference can have a noticeable effect. If the target needs to be reset before each experiment then tens of seconds are added for each experiment. Moreover, the target might remove or change the keys after a limited amount of encryptions.

We analyzed the outputs in more detail and counted how often multi byte changes are observed in both implementations (Table 2). From the number of all faults observed (i.e. at least 1 byte difference), 4 bytes faults[10] are more probable to be observed in the hardened implementation.

To conclude, fewer successful faults are needed to attack the hardened AES.

[9] `https://github.com/kokke/tiny-AES128-C`

[10] These are the faults useful for DFA on AES

The final version will appear at Springer, in `CARDIS 2017` proceedings.