# Share-slicing: Friend or Foe?

Si Gao[1], Ben Marshall[1], Dan Page[1] and Elisabeth Oswald[1,2]

[1] University of Bristol, Bristol, UK
[2] University of Klagenfurt, Klagenfurt, Austria
firstname.lastname@bristol.ac.uk

**Abstract.** Masking is a well loved and widely deployed countermeasure against side channel attacks, in particular in software. Under certain assumptions (w.r.t. independence and noise level), masking provably prevents attacks up to a certain security order and leads to a predictable increase in the number of required leakages for successful attacks beyond this order. The noise level in typical processors where software masking is used may not be very high, thus low masking orders are not sufficient for real world security. Higher order masking however comes at a great cost, and therefore a number techniques have been published over the years that make such implementations more efficient via parallelisation in the form of bit or share slicing. We take two highly regarded schemes (ISW and Barthe et al.), and some corresponding open source implementations that make use of share slicing, and discuss their true security on an ARM Cortex-M0 and an ARM Cortex-M3 processor (both from the LPC series). We show that micro-architectural features of the M0 and M3 undermine the independence assumptions made in masking proofs and thus their theoretical guarantees do not translate into practice (even worse it seems unpredictable at which order leaks can be expected). Our results demonstrate how difficult it is to link theoretical security proofs to practical real-world security guarantees.

**Keywords:** Masking · Side-Channel Analysis

## 1 Introduction

Nowadays, masking is arguably one of the most prevalent countermeasures against side channel analysis. Technically speaking, a Boolean masking scheme splits the key-dependent intermediate state $x$ into a set of shares $(x^{(1)}, x^{(2)}, ..., x^{(d)})$, where $x^{(1)} \oplus x^{(2)} \oplus ... \oplus x^{(d)} = x$. Ever since Ishai, Sahai and Wagner proposed the probing model and the ISW multiplication scheme [ISW03], various new models/masking schemes/improvements have been proposed e.g. [BBD+16, BBP+16, BDF+17]. The goal of these newer developments is to provide better security and/or to save randomness.

Despite increasingly sophisticated leakage modelling and proof techniques that feature in recent masking papers, it remains that in most cases, it is the abstract masking algorithm, rather than its specific implementation, that gets the security proof. In other words, it is up to the crypto-engineers to check whether their implementations have "faithfully" reflected how the algorithm proceeds and whether their chosen platforms satisfy the model assumption used in the security proof. The past 20 years of developments in side channel analysis suggests that this is by no means a trivial task. In the early days, dual-rail logic seemed to be a tempting solution for masking against power analysis in hardware [TV04, PM05]. However, the early propagation effect was shown to undermine its resistance in practice [MOP07]. Trichina's masked AND-gate [Tri03] was designed as a protected 2-input AND gate, yet glitches easily defeat such protection [MPG05]. Thanks to the "non-completeness" property, threshold implementations [NRR06] remain secure even


IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2020, No. 1, pp. 152–174
DOI:10.13154/tches.v2020.i1.152-174

in the presence of glitches: however a number of effects including "coupling" [CBG+17, CEM18, LBS19] may contribute to power consumption that unexpectedly re-combines the secret shares.

On software platforms similar "security gaps" can be as devastating as on hardware platforms. Unlike implementations in (custom designed) hardware, software implementations on commercial processors are restricted to work with a certain set of available instructions. Whilst the instruction set is well defined, and the architectural behaviour is clearly documented, more complex processors have a number of non-architectural registers (e.g. buffers supplying the memory subsystem) and so called micro-architectural features that are not transparent to the developer. Such micro-architectural features can lead to unexpected leakage behaviours: e.g. a masked value still leaks from the buffer in the memory subsystem several cycles after a store operation; or a shift operation which may be implemented via a dedicated Barrel shifter that is always active.

In this paper we stress that crypto-engineers must be extremely cautious about the security assumptions of higher order masking schemes on software platforms. To this end, we analyze a software implementation of Barthe et al.'s secure multiplication [BDF+17], which is provably secure in a "bounded moment model". Despite the fact that the authors did NOT directly link this scheme with any specific software or hardware implementation, due to its intrinsic parallel feature and good performance for higher-order masking [JS17], it has become a prevalent choice for a few bit-sliced masking implementations [JS17, GJRS18]. However, these implementations stipulate that all shares of secret $x$ should be stored within one register (denoted as "share-slicing" in this paper), which is in fact a less-investigated option in the previous studies of software masking [JS17]. In the context of "share slicing", the "independent leakage" assumption in the bounded moment model [BDF+17] implies that "there should be no joint leakage between bits within an operand" (we call such leakage "bit-interaction"). Journault and Standaert implemented a 2/4-share version of Barthe et al.'s multiplication on ARM Cortex-M4 processor and found no evidence for lower-order leakages [JS17]. However they conservatively phrase their security claim with only half of the security order that the theoretical scheme provided.

Other previous studies cast doubt on the ability to avoid bit-interaction leakage. Levi, Bellizia, and Standaert discovered that with appropriate experimental setups, a coupling effect can be observed for a 2-share multiplication [LBS19]. In an entirely different context McCann et al.'s work provides strong evidence for the statistical significance of bit-interaction terms for both the shift and multiplication instructions [MOW17] on an M0 and M4. Sasdrich, Bock, and Moradi also observed bit-interactions on an address bus, which defeats a global table-based threshold implementation if all 3 shares appear simultaneously in the table look-up operation [SBM18].

**Our contributions.** We demonstrate that the "independent leakage" assumption does not hold on an ARM Cortex M0 and M3. Whilst "coupling" [LBS19] was recently offered as an explanation in this context, we argue that at least in the case of the M0 and M3, their micro-architectural behaviour is the (more likely) source.

Because the internal architecture of the M0/M3 implementation of the LPC series is not documented, we explain our reasoning based on a (manual) simulation of a simple Barrel shifter. Experiments (on an M0/M3) demonstrate that second order leakage terms in the form of bit-interactions can be observed (even when the coupling effect modelled by [LBS19] does not come into play). Unfortunately some of the instructions with strong bit-interaction leakage terms are quite common among implementations of Barthe et al.'s multiplication.

Our experiments clearly show that the bit-interaction leakages are devastating for share-sliced implementations of higher-order masking. We investigate a bit-sliced second-order implementation of the ISW masking scheme (utilising share slicing) to find that

the masked AND gate has first order leaks. Next we replace the ISW AND gate with an implementation of Barthe et al. (using two as well as four shares) and find that also the latter is completely vulnerable and does not deliver on the theoretical security claims: we detect leaks on all orders lower than four, and it turns out that the second order leakage is as informative as the fourth order leakage. This implies not only a theoretical order reduction but one that leads to an attack that is better than the fourth order attack in real terms.

## 2 Software masking: where are we now?

We focus on Boolean masking schemes in this paper. In a Boolean masking scheme, a sensitive intermediate state $x$ is represented by a $d$-variate set $(x^{(1)}, x^{(2)}, ..., x^{(d)})$ where $x^{(1)} \oplus x^{(2)} \oplus ... \oplus x^{(d)} = x$. We use capital letter $A$ to represent the first encryption block, whereas B, C, D, ... denote the following concurrent encryption blocks respectively. $A_j^{(i)}$ stands for the $i$-th share of the $j$-th bit of block $A$.

Since the shared computation for any linear transformation is trivial, the differences between various masking schemes lie in the non-linear transformations (a.k.a. the Sboxes). Three popular solutions exist in literature:

1. Using table lookups [Mes01, PR07]. According to the first $d-1$ shares, a masked Sbox is generated (beforehand or on-the-fly), so that the masked Sbox can be simplified as a single table look-up.

2. Using bit slicing. Although protecting a whole Sbox can be difficult, there are various schemes that protect a simple gadget, such as a 2-input AND gate [ISW03, Tri03, BDF+17]. As the XOR operation has a trivial shared form, we can decompose the Sbox into such gadgets and apply our side channel protection in a divide-and-conquer manner. By default, this approach fits the customized hardware better, as the proposed scheme is designed for 1-bit computing unit. In most software implementations, to avoid wasting the remaining bit-width that a commercial processor provides, engineers usually deployed a bit-sliced version of these schemes, which will be further discussed in detail.

3. Using finite field arithmetic. Instead of relying on "gate-level" protections, an alternative solution would be extending the protected "gadget" to a larger field [RP10, CPRR14]. This usually means the engineers can build Sboxes based on secure field computations, which fits well for Sbox constructions based on finite fields (eg. the AES Sbox). Note that even in this case, as field computation is hardly feasible for the same bit-width of modern CPUs, in practice these schemes might still come with some bit-slicing effort [GR17].

### 2.1 Bit-slicing

Biham proposed the bit-slicing technique for software implementations of DES in order to take full advantage of the available bit-width [Bih97]. The core idea consists of "slicing" the bits that share the same operation to one register, so that all the following bit-wise operations (such as AND, XOR, OR, etc.) are performed in parallel on all bits at once. For instance, for a block cipher in the ECB mode, we know for sure that all the same bits from different blocks must share the same encryption operations. Thus, a register can be packed as Table 1:

In the following discussion, we denote this type of bit-slicing as "block-wise" slicing. Although "block-wise" slicing has little precondition, for an $n$-bit processor, it does require at least $n$ concurrent blocks to achieve its best performance. Besides, as each register only

**Table 1:** Bit allocation for block-wise bit-slicing

| Bit no | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|
| Stored bit | $A_1$ | $B_1$ | $C_1$ | $D_1$ | ... |

represents 1-bit of the state, an AES-128 would need 128 32-bit words, which is clearly out of the bound for general data registers in most CPUs. As a consequence, most of the states must be stored in memory, which leaves a moderate pressure on the memory footprint. For AES-like ciphers, as all the Sboxes in the round function are exactly the same, we can also choose an "Sbox-wise" slicing as Table 2:

**Table 2:** Bit allocation for Sbox-wise bit-slicing

| Bit no | 1 | 2 | ... | 16 | 17 | 18 | ... | 32 |
|---|---|---|---|---|---|---|---|---|
| Stored bit | $A_1$ | $A_9$ | ... | $A_{121}$ | $B_1$ | $B_9$ | ... | $B_{121}$ |

As the first bits from all 16 Sboxes are slicing together here, it only takes $\lceil \frac{n}{16} \rceil$ blocks to fill up an register. For a 32-bit processor, 2 concurrent blocks can already ensure the best possible throughput. Meanwhile, 8 32-bit words has much lower cost in terms of data registers or memory usage.

## 2.2 Masking: the share allocation problem

The first question a crypto-engineer has to answer when implementing a bit-sliced masking scheme, is how the shared bits should be stored in registers/memory. Since a $d$-share masking scheme expands one block to $d$ times its size, we need $d$ times the memory space compared to the unprotected version. Previously, the preferable solution was to keep the same bit allocation as the unprotected version, while each share will take its own register. In this paper, we called this strategy "bit-only-slicing". A 2-share block-wise "bit-only-slicing" implementation on a 32-bit processor can be written as Table 3:

**Table 3:** Block-wise bit-only-slicing:the first share

| Bit no | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|
| Stored bit | $A_1^{(1)}$ | $B_1^{(1)}$ | $C_1^{(1)}$ | $D_1^{(1)}$ | ... |

Note that this type of implementation not only poses a pressure on the register/memory cost, but also risks share-combining in the memory accesses. As each data share is stored individually, loading/storing data shares sequentially might lead to transition leakage caused by the memory buffers. Obviously, the other option would be packing multiple shares into one register, which will be denoted as "share-slicing" in the following. A 2-share block-wise "bit-only-slicing" implementation is presented in Table 4: for one block encryption, this type of implementation uses far less general-purpose registers than the one in Table 3.

**Table 4:** Block-wise share-slicing

| Bit no | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|
| Stored bit | $A_1^{(1)}$ | $A_1^{(2)}$ | $B_1^{(1)}$ | $B_1^{(2)}$ | ... |

Considering most masking schemes are designed as a serial procedure, splitting data shares from one register and applying the required operation is not efficient at all. On the other hand, bit-only-slicing complies with the sequential probing model, which acts as the

foundation for many security proofs. As a consequence, until recently, bit-only-slicing was indeed the dominant solution, while share-slicing was the unpopular and neglected option. In fact, in many previous papers, "software bit-sliced implementations" usually imply bit-only-slicing [BGG+14, dGPdLP+17, GR17]. Goudarzi et al. evaluated the performance of various bit-sliced masking multiplication schemes in ARMv7 assembly [GJRS18]: except for Barthe et al.'s multiplication [BDF+17], all other schemes were implemented with "bit-only-slicing". However, in 2017, Barthe et al. proposed the "bounded moment model" [BDF+17], together with a multiplication gadget that is designed to run in a parallel environment (Algorithm 1):

**Algorithm 1** A 1-bit 2-share Barthe et al.'s multiplication

**Require:** $a = \left(a^{(1)}, a^{(2)}\right)$ and $b = \left(b^{(1)}, b^{(2)}\right)$
**Ensure:** $c = \left(c^{(1)}, c^{(2)}\right)$ that satisfy $c^{(1)} \oplus c^{(2)} = (a^{(1)} \oplus a^{(2)}) \wedge (b^{(1)} \oplus b^{(2)})$
1: $x1 \leftarrow a \wedge b$
2: $r \leftarrow \{0,1\}^2$
3: $y1 \leftarrow x1 \oplus r$
4: $x2 \leftarrow a \oplus ROT(b,1)$
5: $y2 \leftarrow y1 \oplus x2$
6: $c \leftarrow y2 \oplus ROT(r,1)$
7: **return** c

Note that *ROT* in Algorithm 1 means "rotating 1 position for all the shares". Despite the fact that the authors did not link their algorithm with any specific software or hardware implementations, as the security claim is proved in a parallel model, the natural implementation on software platforms is indeed, "share-slicing". The 32-bit Thumb-2 instruction set (explained in Section 3.1) provides the "flexible operand 2", which suggests that shifting/rotation can possibly be done without costing any instruction cycles. As a consequence, *ROT* in Algorithm 1 can be efficiently implemented with a rotation (for 32-share multiplication) or a few shifts (for less shares). Both Goudarzi et al.'s work [GJRS18] and Journault and Standaert's work [JS17] proves that this type of implementation does gain a significant advantage in terms of efficiency.

## 2.3 Well-known threats

It can be easily forgotten that Barthe et al.'s multiplication, when implemented in a share-slicing software implementation, does not share the same assumption as most previous software masking schemes. Technically speaking, the "independence assumption" used by the bounded moment model, has only been extensively studied on hardware platforms. In a share-slicing setup, the "independence assumption" must be interpreted as follows:

**Assumption** (Bit-wise independent assumption)**.** *For all bit-wise instructions used by the encryption, their leakage functions must not exhibit joint leakages of bits within the same operand (register).*

In other words, the only permissible leakage functions for bit-wise operations consists of linear combinations of their bits; no higher order term may occur.

The major threat listed in the literature for software masking, is the transition-based leakage [BGG+14, dGPdLP+17]. Balasch et al. proposed the "order-reduction" theorem, which suggested engineers can take a conservative approach to avoid security threats, by doubling the desired security order [BGG+14]. Although the authors did not make it explicit, their security proof actually relies on the fact that the masking schemes use "bit-only-slicing". However, nowadays where "share-slicing" is considered to make implementations more efficient, we must be aware that the results of [BGG+14] do **not** cover "share-slicing".

To this end, it is worthwhile to check the validity of this assumption in practice. Clearly, coupling is still an issue [LBS19], as it roots in lower-level physical circuits. However, considering the enormous micro-architectural events happening inside our tiny "black-boxes", it would not be surprising if some of them lead to some unexpected leakages. After all, are we being too optimistic to believe the leakage behaviour of software platforms will resemble its hardware counter-part, even if we know the running circuits are entirely different?

# 3 CPU Datapaths: Assumptions v.s. Reality

As previously stated, software developers are often required to treat a CPUs as a "black box" in terms of how it executes the instructions they program. Further, we have also seen that certain assumptions about how the CPU handles data are essential for the security of bit-sliced cryptographic implementations.

Clearly it is a matter of some concern that the assumption on which the security of a crypto-system depends cannot be verified due to the black-box nature of many CPUs prior to implementing the system.

Most modern CPUs are implemented as data processing pipelines, where the execution of instructions is spread over time into stages. This reduces the time taken to produce a result for each stage, and means a CPU can be run at a higher clock rate, thus improving its performance. The ARM M0 and M3 CPUs, used for the experiments in Section 3.3 both use 3-stage pipelines. Pipelines are a well known artifact in side-channel analysis, and care must be taken to make sure that operands which are assumed to be independent are not operated on adjacently in the pipeline which would otherwise reduce the security level.

While pipeline depth and the function of each stage is often lightly documented, the exact datapath structure inside the pipeline almost never is and can even change between instantiations of the same CPU core. This makes it hard to discern if and how interaction between adjacent bits of the same register occurs, even in instructions which explicitly only operate in a bitwise fashion. It is this sort of interaction which must be avoided to preserve the security level of bit-sliced masking schemes.

## 3.1 ARM Cortex-M family

The ARM Cortex-M family is a group of 32-bit RISC processors, firstly introduced back in 2004 [Mar13]. Unlike the Cortex-A family (for supreme performance) and the Cortex-R family (for reliable mission), the Cortex-M family aims at various micro-controller usage in many energy-efficient systems. In this paper, we select two Cortex-M processors as our targets: one Cortex M0 (LPC 1114) and one Cortex M3 (LPC1313). Both processors were manufactured by NXP. Technically, both processors have 16 32-bit registers, executing the Thumb instruction set [ARMa, ARMb]. However, the Cortex-M0 implements the ARMv6-M architecture, which predominantly supports 16-bit Thumb instructions (a.k.a. Thumb-1) [ARMa]. The Cortex-M3, on the other hand, implements the ARMv7-M architecture, which supports "a large number of 32-bit instructions that Thumb-2 technology introduced into the Thumb instruction set" [ARMb].

One important difference (for cryptographic developers at least), is that the Thumb-2 technology in ARMv7-M allows the "flexible operand 2", which suggests one of the operand can be shifted without costing extra clock cycles. In Cortex-M0, performing the same operation must take at least one explicit shift instruction. The instruction set supports four types of shifting: the logical shift left/right (LSL/LSR), the arithmetic shift right (ASR), and the rotate-right (ROR). Note that for Cortex-M0, the rotation number of ROR has to be stored in a register instead of an immediate value. As our experiments run both on

Cortex-M0 and Cortex-M3, throughout this paper, our targeted masked implementation [1] always uses 16-bit Thumb-1 instruction by default, despite the fact that on Cortex-M3 there might be more efficient options.

## 3.2 Specific Example: Barrel Shifters and ARM

Here, we look at how some implementations of an ALU can jepoardise the security of a bit-sliced implementation.

It is understandable to assume that if a bitwise instruction is used, no two bits will interact at a hardware level. However, this is not the case in a typical CPU design. Implementing multiple different instructions in hardware requires that their results be calculated simultaneously, and then the desired one selected, with the others ignored. Similarly, the exact implementation of a hardware datapath can introduce unexpected interactions which *cannot be avoided*, as they are intrinsic to the datapath circuit.

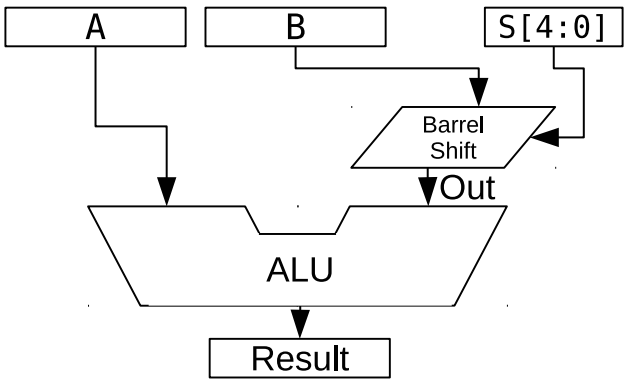


**Figure 1:** A simplified block diagram of the ARM ALU, showing the operand being fed to a barrel shifter before reaching the ALU.

Whilst the exact architecture of the Cortex-M processor is not publicly accessible, most public sources show a diagram as Figure 1 [Fur00]. Figure 1 shows a block diagram of how this fits into the ARM ALU. As one input operand to the ALU can be rotated prior to being fed to the rest of the ALU, such architecture explains the reason why Cortex-M3 processors can support the 'flexible operand 2" feature. No clue has been given on whether Cortex-M0 follows the same diagram as Figure 1: however, as Cortex-M0 also supports shift instructions, there must be a shifter somewhere on the datapath [2].

In hardware, shift or rotation by a constant is simply a re-arrangement of wires and requires no logical interaction of the bits. This is the impression given in lines 3 and 4 of Algorithm 1. Rotation or shifting by a variable amount (or, a constant amount using hardware capable of variable amounts) however requires that result bits be sourced from one of $N$ different input bits. In the case of the ARM M0 and M3, $N = 32$. To implement this in a single processor clock cycle, a circuit construct known as a barrel shifter is used. In a 32-bit barrel shifter, 5 multiplexing stages are used to select the correct output bit from the inputs. We refer to bit $I$ of the input word to the barrel shifter by $B_i$, and the bits of the shift amount by $S_j$.

From Figure 2, we can see that in the first layer of the barrel shifter, any bit $B_i$ could interact with bit $B_{(i+1)\%32}$ based on bit $S_0$ of the rotate amount if the logic path from the operand sources ($B_i$,$B_{(i+1)\%32}$) is shorter than the path to the multiplexor input-select

[1] This only applies to the triggered part of the masked implementation. Compilers can decide which instruction set to use for other non-critical parts of the program.

[2] If a barrel shifter is not placed before the ALU, it could possibly be parallel to the ALU or even packed as a part of the ALU.

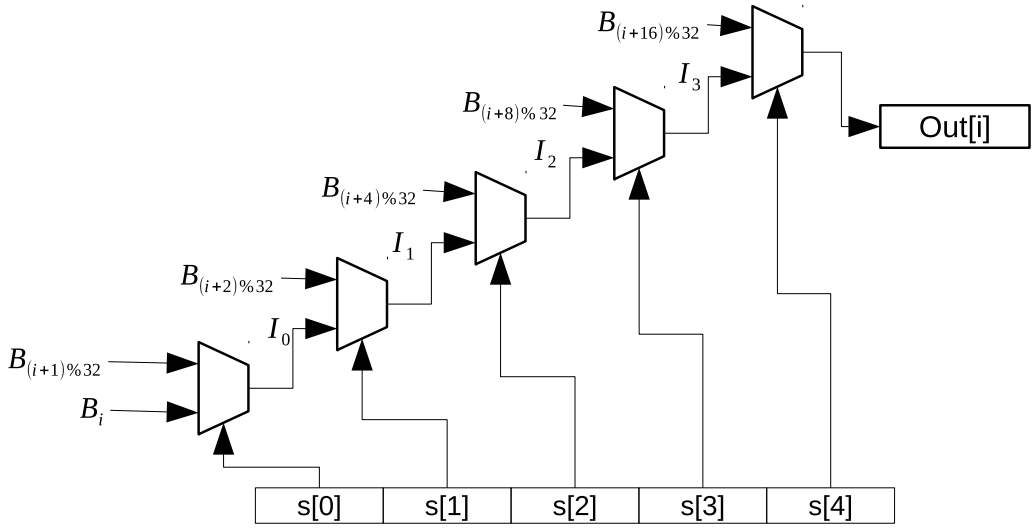


**Figure 2:** Logic function for a single output bit of a 32-bit wide barrel shifter, as found in the ARM ALU. An input word $B$, is rotated by an amount $s$ to produce a result $Out$. Note the intermediate nets $I_x$ at each stage of the circuit.

signal ($S_0$). This would cause the intermediate net $I_0$ to transition. Likewise in layer 1, $I_0$ can interact with bit $b_{(i+2)\%32}$, and so on down to interactions between $I_3$ and $b_{(i+16)\%32}$.

These interactions are heavily dependent on the physical implementation of the barrel shifter, and how the multiplexor input-select signals causes hamming distance leakage between bits of the register. This hamming distance leakage can occur when the input select signal to the multiplexor arrives *later* than the input signals, or otherwise changes after the input signals have stabilised. This can be caused by the combination of two effects: 1) when the input-select bits of the multiplexor glitch. 2) when the input-select bits of the multiplexor do not glitch, but arrive later within the clock cycle than the input bit values. These effects are shown for a single bit of the barrel shifter in Figure 3. Clearly, this sort of behaviour can be expected to break the bit independence assumption stated in Section 2.3.

## 3.3 Instruction-wise leakage analysis on bit-interaction

As we have discussed in the previous sections, due to various micro-architectural effects, the leakage of a commercial processors can sometimes falsify the "bit-wise independent assumption". Unlike the physical coupling effect that "uniformly" exists for every instruction that operating on the shared data, such micro-architectural level leakages strongly depend on the specific instruction. To this end, we have performed TVLA-based [GGJR+11] leakage analysis on several 16-bit Thumb-1 instructions on both our Cortex M0 (NXP LPC1114) and M3 processors (NXP LPC1313). More specifically, for the fix group, the 32-bit operand is composed as 16 2-shared bits of 0s (in other words, $\{a, a, b, b, ..., p, p\}$, where $a, b, ..., p$ are 16 1-bit randomness). The random group is composed with all random values as usual. Theoretically speaking, a fix-vs-random test here should reveal any leakage that combines bit $2i$ and $2i + 1$, which could increase the signal-to-noise radio (SNR) compared to operating 1 bit with 2 shares and setting all other bits to constant[3].

The target code snippet includes one target instruction, with 8 NOP-s (*mov r8,r8*)

[3]Strictly speaking, it still depends on the specific leakage model: however, in practice, we found this approach provided a better SNR in most cases

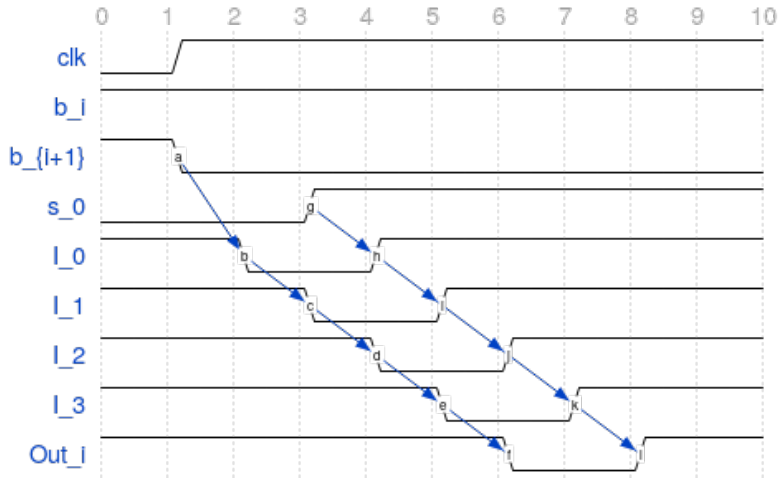


**Figure 3:** Timing diagram for a single output bit of a barrel shifter, corresponding to figure 2. At time 1, on the rising edge of the `clk` signal, input bits $B_i$ and $B_{i+1}$ toggle, where $B_i$ remains constant and and $B_{i+1}$ toggles. The toggling of $B_{i+1}$ is propagated to $I_0$, since the multiplexor select signal $s_0$ is still low. This value continues to propagate all the way to $Out_i$ at time 6. However, the signal $s_0$ changes at time 3, due to differences in path length to the multiplexor. This means that at time 4, the output of the first multiplexor, $I_0$ changes *back* to represent the value of $B^i$. This difference in path lengths to the multiplexor between $B_i$, $B_{i+1}$ and $s_0$ has hence leaked the hamming distance between $B_i$ and $B_{i+1}$.

before/after the target instruction. An example code snippet can be found in Figure 4. All relevant registers (except for the one that stores the shared operand) have been cleared to 0 beforehand, so that there will be no leakage from other sources. The target cores run on two SCALE boards [Pag], where all the compiling flags are set to the default values (details can be found the Github repository of the SCALE project). By default, SCALE boards use 3.3 V power supply with a 100 ohm shunt resistor [Pag]: since the authors of [LBS19] stated the coupling effect can be amplified with larger external resistors, one may ask whether this setup causes significant coupling effect. We would like to argue that in the following experiments, coupling effect is at least, not the only contributor to the bit-interaction leakage: even if there is no external resistor, such leakage can still be detected in our experiments. Interested readers can find the experimental results and discussions in the Appendix.

Our target processors always run at 12 MHz, so the whole snippet takes around

```
1 Instruction_test:
2   //r1= shared data, all other registers cleared
3   nop
4   ...                 //8 nops
5   nop
6   lsls  r3, r1,#1     //Target instruction
7   nop
8   ...                 //8 nops
9   nop
```

**Figure 4:** Instruction test code for Thumb-1 instruction LSL

$17 \times 83.33ns = 1417ns$. We have captured our traces with a Picoscope 2206B, running at 250MSa/s. To make sure the target leakage is captured, our traces contain 600 samples, which also cover a short period after the target snippet. For each tested instruction, we have captured 1 million traces for T-test.

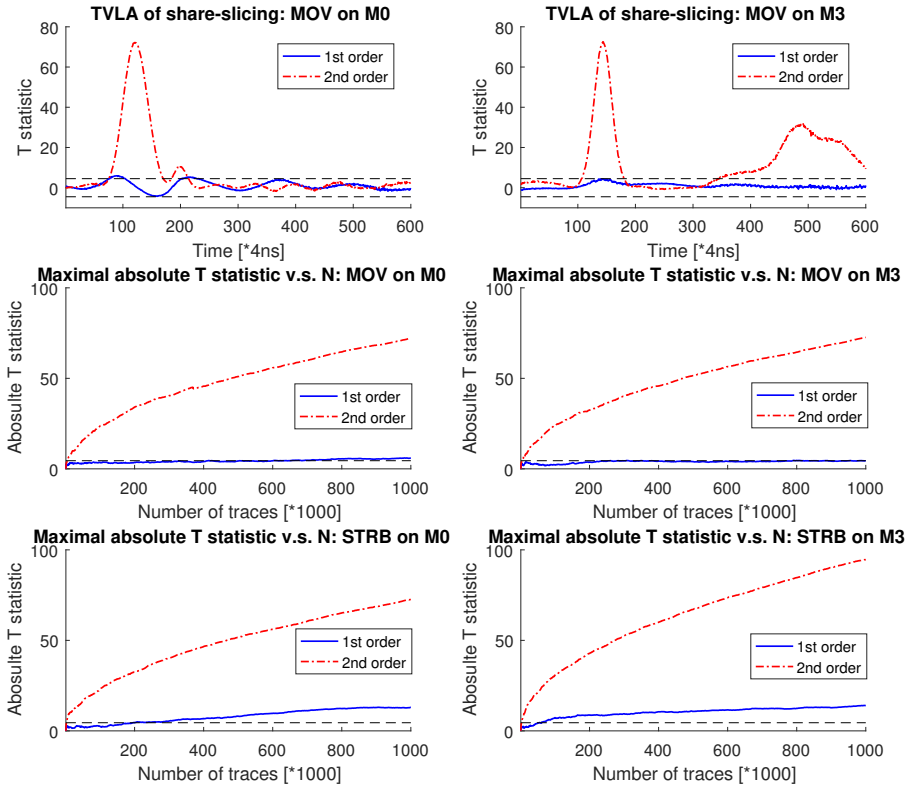


**Figure 5:** Instruction-wise leakage analysis: MOV/STRB

The top two figures in Figure 5 demonstrate the T statistics we got from testing the *MOV* instruction on both M0 and M3. Apparently, the target instruction lies somewhere around sample 150, which always shows a strong second-order leakage. *MOV* did not show clear first order leakage on M0 or M3, within 1 million traces. This suggests that the "coupling" effect [LBS19] has not become the dominant factor yet. On the other hand, in our experiments, many instructions did show some barely exploitable leakages, such as *STRB* in the bottom of Figure 5. Explaining such leakage could be a difficult task: the "coupling" effect [LBS19] could be one of the reasons, but all micro-architecture effects may also contribute. Nonetheless, the magnitude difference suggests such leakage might never be useful in practice, as the higher-order attacks seem to be much more efficient.

An intriguing phenomenon can be spotted from Figure 6. Unlike other instructions, shift instructions (LSL/LSR) can sometimes create a much stronger bit interaction, which leads to strong first-order leakage. Such leakage can even surpass the magnitude of the second-order leakage, making it a perfect target for power analysis. Surprisingly, only one of the shift instructions will leak this way: for our M3 core (NXP LPC1313), it is the left shift (*LSL*) has a strong first-order leakage; on our M0 core (NXP LPC1114), it is the other way around. Nonetheless, such leakage can possibly lead to devastating threats in

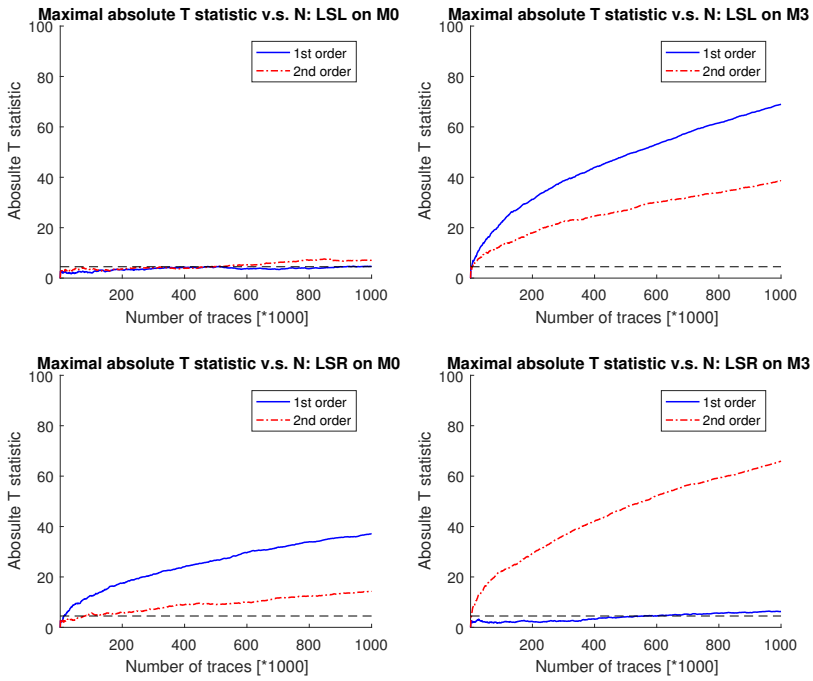


**Figure 6:** Instruction-wise leakage analysis: LSL/LSR

practice: if the implementation accidentally uses the "leaking shift", the security order can be significantly reduced without investigating any external amplifying effort [LBS19]. We suspect this phenomenon is caused by the "barrel shifter", although we cannot confirm it as both cores' design codes are not publicly available.

**Other tested instructions.** In our experiments, we have tested 12 16-bit Thumb-1 instructions. Most of other instructions' results, to some extent, resembles Figure 5. Some of those involve subtle leakage behaviours that might root in the unused datapaths within the ALU. Nonetheless, as these results are significantly weaker than Figure 6, we leave the detailed investigation as future study. Due to the page limit, we report the rest of our tests in the Appendix.

# 4 Studying the Concrete Impact of Bit-Interactions

## 4.1 A masked AES with share-slicing

A natural question to ask is whether bit-interaction leaks will lead to any real threat in practice. Our previous sections seem to suggest it may bring severe security reductions, at least for the "leaking shift". However, strictly speaking, we still need to validate such an effect in a complete masked cipher rather than some small and somewhat artificial code snippets. The only publicly-available share-slicing example we found on Github is from Virginia Tech [YYP+18][4]: the underlying masking scheme is a 2-share one, with a 2-share ISW multiplication as the basic non-linear gadget for the AES Sbox. The entire project is written in C, which makes it fairly easy to adapt to any new platform.

[4]Github repository at: https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation

**Table 5:** 2-share block-wise share-slicing

| Bit no | 1 | 2 | 3 | 4 | ... | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|
| Stored bit | $A_1^{(1)}$ | $A_1^{(2)}$ | $B_1^{(1)}$ | $B_1^{(2)}$ | ... | $O_1^{(1)}$ | $O_1^{(2)}$ | $P_1^{(1)}$ | $P_1^{(2)}$ |

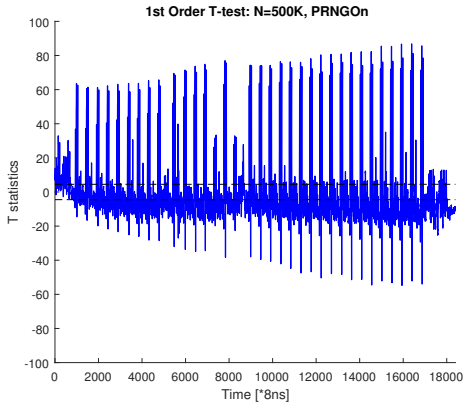


**Figure 7:** 1st Order TVLA on the AES Sbox part

Although not formally documented, this implementation stores both shares in the same register (a.k.a. share-slicing). Block-wise share-slicing offers good flexibility: for the linear transformations, as every bit in one register comes from exactly the same position within a block, the computation code will always stay the same. For clarity, we provide the register allocation for this implementation in Table 5.

Implementing the ISW multiplication [ISW03] is not trivial in this representation, as the ISW multiplication is designed to operate sequentially on each share. The authors [YYP+18] used a "bit-swap" table, which swaps the first and second share through a table look-up. Obviously, any transition from this share-swapped state to the normal state leads to exploitable leakage. Besides, sending all the shares to the address bus might also cause problems on certain platforms [SBM18]. Since the ISW multiplication is also written in C, the compiler will autonomously decide the detailed execution, which can easily lead to shares recombination through register or bus transitions. As a consequence, it is indeed expected to see some first order leakage, at least for the multiplication (or AND2) gadgets; that is exactly what the standard first order T-test shows in practice. As we can see in Figure 7 all 34 AND gates used in the bit-sliced AES Sbox produce significant leakages. We suspect the majority of the leakages originate from the register/bus transitions happening within the ISW multiplication. In that sense the security loss complies with the "order reduction theorem" [BGG+14].

## 4.2 Share-slicing with Barthe et al.'s multiplication

As the ISW scheme is designed with a serial computation model in mind, one may argue that the security flaw comes from the mismatch of the ISW multiplication and the intrinsically parallel share-slicing implementation. Thanks of the flexibility of the Virginia Tech implementation, it is easy to replace the ISW multiplication with Barthe et al.'s multiplication and to include a 4-share version to evaluate higher order leakage.

We wrote the code for both the 2-share and 4-share AND gates in Thumb-16 assembly: we tried our best to resemble Goudarzi et al.'s code [GR17], which contains ARM assembly instructions that are not available on the Cortex M0. Note that Goudarzi et al.'s code

```
1  Triggered computation:
2      // r0 = a, r1 = b, r2=r
3      movs      r4, r0          //r4=a
4      ands      r4, r1          //r4=a&b
5      eors      r4, r2          //r4=(a&b)^r
6      // Generate Mask 0xeeeeeeee
7      movs     r5, #0xee
8      ...
9      eors      r5, r6          //r5=0xeeeeeeee
10     //rotation of b by 1
11     mov       r8, r8          //Clear HD
12     lsls      r6, r1, #1      //r6=b<<1
13     mov       r8, r8          //Clear HD
14
15     //The following computation has been commented out
16     //ands      r6, r5        //r6=(b<<1)&0xeeeeeeee
17     //lsrs      r7, r1, #3    //r7=(b>>3)
18     //bics      r7, r5
19     //eors      r6, r7        //r6=(b<<<1)
20     //...
```

**Figure 8:** "Trimmed code" for 4-share Barthe et al.'s multiplication

actually contains both shift left and shift right operations [5]: the unrolled version uses LSR (a.k.a right shift), whereas the rolled version uses LSL (a.k.a left shift).

To ensure that any leakage that we may encounter is related to bit-interactions rather than transitions we would need to manually check the code for the potential existence of transition based leaks. This task is much more complicated than it sounds at a first glance: keeping track of all possible transitions in the micro-architecture is almost impossible for external developers, considering the micro-architecture is not fully transparent. A possible workaround is to focus only the shifting part (eg. line 4 in Algorithm 1) instead of the whole multiplication. As our target in this paper is the bit-interaction leakage, in the following experiments, we always examine the leakage for this particular "trimmed code" for acquisitions (Figure 8). Two extra nop-s ("mov r8, r8") haven been added to make sure there is no transition-based leakage on data $b$.

As mentioned before, complex power simulators like ELMO [MOW17] can detect the potential bit-interaction here. This is hardly surprising though: Table 9 in [MOW17] explicitly explained that second order bit-interaction terms exist in ELMO's power model. Having said that, as the bit-interaction is relatively weaker than the other terms, the target leakage can be easily buried by algorithmic noise. In our experiments, the default version of ELMO cannot report such leakage within 1 million traces, unless the power model is tweaked in a certain way. Due to the space limit, details about ELMO simulation will be presented in the Appendix. In the following sections, our experiments only focus on realistic acquisitions.

Our attack setup is fairly straightforward: for this block-wise bit-sliced implementation, we set the first block as our plaintext and set all other blocks to random values. This corresponds to the most practical yet challenging scenario: only 2 bits (or 4 bits in the 4-share version) are part of our attack target, all other bits are treated as noise. We chose one of the multiplications close to the Sbox output as our target so that the key guesses

[5] Here we focus on the shift-s that involve the full shares. There is usually an opposite shift operating on part of the shares, which can be slightly more complicated for security analysis.

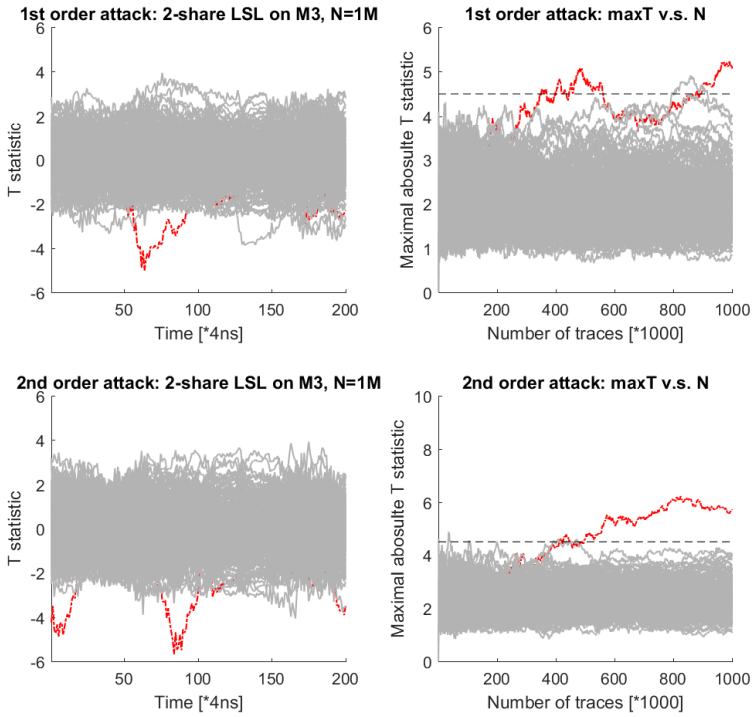


**Figure 9:** Attacking the 2-share version on M3

will have a complicated non-linear impact on the intermediate state. Before entering the target AND, both inputs are refreshed to make sure their shared form are independent. All traces were captured with a PicoScope 2206B scope running at 250 MSa/s, using the same M0/M3 cores in the last section. Unlike the previous work [LBS19], we did not try any advanced distinguisher here: as the target intermediate state is only 1 bit and the leakage sample is a univariate one, most distinguishers should be equivalent [MOS11]. In the following experiments, we simply use the T-test based 1-bit DPA [MOP07], as it can share exactly the same code as our TVLA test.

## 4.3 Attacks on a 2-share masked implementation

The T-statistic values in Section 3.3 seem promising, and thus unsurprisingly it is possible to reveal leaks also within the more realistic code snippet that we are now using. Unlike in the previous experiments, only the lowest 2 bits are valid shares which are contributing to exploitable leakage and thus we must deal with a low signal to noise ratio. In practice, the attacker may consider a number of options, such as attacking the first few AND-s within the Sbox or taking the following plaintext blocks into consideration. However, as the purpose of this paper is verifying the "bit-wise independent assumption", we are after a certificational attack rather than the "best strategy".

Figure 9 confirms the "leaking shift" does lead to successful attacks. More specifically, in the first order, the correct key byte becomes one of the front runners after 400k traces, although it has not got much distinguish margin until 850k. Meanwhile, the second order attack clearly shows the correct key byte after 500k and leaves an adequate margin ever since.

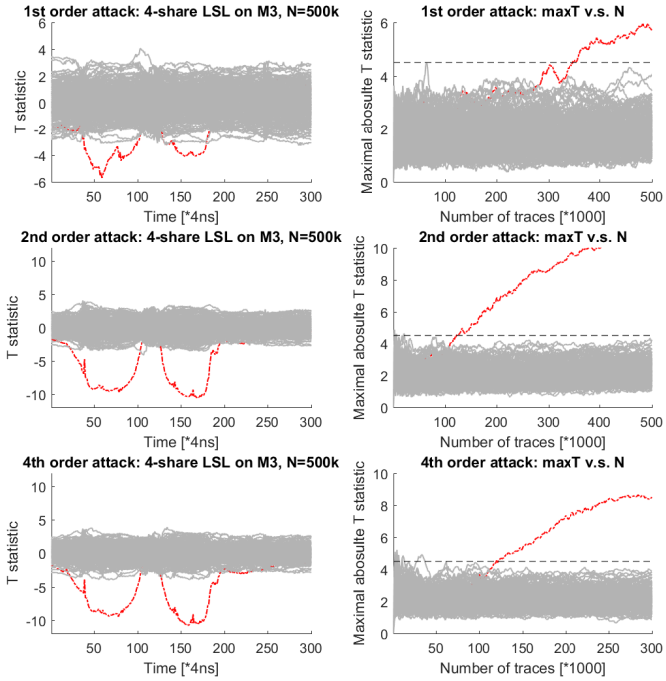


**Figure 10:** Attacking the 4-share version on M3: each trace repeat 1000 times

## 4.4 Attacks on a 4-share masked implementation

Attacking a 4-share version requires considerably more traces than a 2-share version. A well established workaround (that does not require improving the SNR of the setup) is to average over traces (with the same input) to reduce the environmental noise, see e.g. Journault and Standaert's work [JS17]. We adopt this practice for this attack.

Figure 10 shows the attack results for the 4-share version on our M3 core, where each trace is averaged over 1000 acquisitions. The red dash-dot line represents the correct key guess while the grey lines represent 255 wrong key guesses. Clearly, the second order attack becomes the most efficient one, which suggests for this specific shift (LSL on M3), bit-interaction leakages reduce the security order.

Perhaps the most interesting point is that we can still observe exploitable 1st order leakage. The existence of such leakage is clear evidence for bit-interaction leakage because it cannot be explained via transitions between consecutive values (see also the "order reduction theorem" [BGG+14], no transition in the 4-share version should produce first order leakage). Thus, the first order leakage here must come from other logic re-combining (Section 3.2) or physical re-combing ("coupling"). The fact that such leakage does not exist for the "non-leaking shift" (LSR on M3) suggests it is unlikely that coupling is the dominant factor here, as coupling should affect similarly all shift instructions. As we can see in Figure 10, the divisor 2 might be an overly-optimistic expectation for the security order reduction and therefore invalidates that theorem for share-slicing implementations,.

**Remarks.** It seems important to clarify the differences between our results and the results reported in Journault and Standaert's work [JS17]. Multiple factors can explain the differences. Firstly, although we only show the leaking shifts in this section, the only shift instruction may behave differently. Secondly, the implementation of the M0/M3

architecture differs across different manufacturers, thus it is *a priori* unclear if a specific M0/M3 has a leaking shift (or not). Thirdly, although both experiments use repetition and averaging, we use 500k traces, where each trace is average among 1000 repetitions—Journault and Standaert's work used 120k traces where each one repeated 50 times. The total number of traces in our experiment is significantly larger. Last but not least, the experiments were conducted with different setups (i.e. different boards, measurement equipment and configurations), thus it is possible that our setup captured the existing leakage whereas the other setup did not.

We would also like to emphasise that our point with Figure 10 is about the comparison of these trends, not on any concrete signal-to-noise (SNR) level. Considering we used 1000 repetitions in our experiments, if the attacker builds a similar measure setup as ours, he or she cannot successfully attack the masked implementation until he or she uses a challenging amount of traces (say 1000 times of the number of traces in Figure 10). The goal of the repetition here is merely getting power traces with higher SNR so that the evaluation can be finished in a reasonable time period. One can still argue a weaker version of the "independent assumption" may still hold, i.e. "although there are bit-interactions, such leakage is way too noisy to explore". Nonetheless, new SNR-dependent security assumptions are certainly out of the scope.

# 5 Share-slicing: pros and cons

Our work clearly demonstrates that the "bit-wise independence assumption" does not necessarily hold in practice and perhaps even worse we have observed a more severe security reduction than the "order-reduction theorem" suggests. To this end, a natural question to ask would be, is "share-slicing" still a valid option on software implementations, or is it something we should always avoid?

## 5.1 Advantages of Share slicing

Despite the possible security flaws, "share-slicing" is still a tempting option for efficient implementations. Previous works [JS17, GJRS18] already demonstrated that Barthe et al.'s multiplication [BDF+17] is relatively efficient, in terms of both the required randomness and executing cycles. Its advantage keeps increasing with the security order: if the number of shares equals the bit-width of the processors, the $ROT$ in Algorithm 1 can possibly be replaced with a rotation instruction, which further saves some executing cycles. Of course, this will not be applied to instruction sets like 16-bit Thumb-1, as the rotation instruction in Thumb-1 forces the source and destination register to be the same (i.e. causes transition-based leakage). For more advanced instruction sets, this will not be a problem anymore.

An interesting property of "share-slicing" is that it also creates this "barrier of threats": as any linear transformation should "faithfully" perform only the bit-wise XOR-s, in theory, they should be free from the transition-based leakage. In other words, if we assume the " bit-wise independent assumption" is well respected in XOR, a share-slicing implementation would bound most threats within the masked multiplications. In a bit-only-slicing implementation, however, as different shares are stored in different registers, the linear transformation (especially the loading data share part) could also produce order-reduced leakage . Although protecting any non-linear component is still a tricky task, such "barrier of threats" might be desirable for professional programmers and engineers.

### 5.2 Disadvantages of Share slicing

All our experiments have proved that the "bit-wise independence assumption" does NOT always comply with current commercial processors. This puts a question mark on the security of such schemes, as the security proofs rely heavily on this specific assumption. Perhaps the most devastating effect is, unlike the transition-based leakage, for logic recombing (or coupling [LBS19]), that such security order reduction is not bounded by the divisor 2 [BGG+14].

Although our experiments showed that the first-order leakage is much smaller, the existence of such leakage makes the theoretical guarantees void and the practical security level estimation a much harder task. To our knowledge, there is little quantitative comparison between bit-interaction leakage and transition-based leakage, especially in higher orders. If we simply assume the "order reduction theorem" has already captured all possible transition-based leakage, for a 4-share version, using 4 registers to store 4 shares seems a better idea. Of course, one can always argue that Barthe et al.'s multiplication can also be implemented in a bit-only-slicing manner. However, that is contradicted with what the parallel bounded moment model suggested. Besides, most efficiency benefit will also be lost if it fell back to the traditional bit-only-slicing.

## 6 Conclusion

Micro-architectural effects have become a concern for even medium complexity processors such as the ARM Cortex M family. There are a variety of micro-architectural events happening internally that are not exposed to the programmer, which contribute significantly to the leakage behaviour of the processor and which do not align with assumptions made masking proofs. The "trivial" software implementation of Barthe et al.'s multiplication [BDF+17] is a good example to highlight this mismatch: on a commercial processor, such gadget requires all secret shares to be stored within one register (a.k.a. "share-slicing") but the bits must not interact with each other through the bit-wise instructions (eg. AND, XOR, LSL, etc.). On the contrary, we argue that in the presence of barrel shifters, we must expect bit-interaction leaks. Although the exact cause of such bit-interaction leakage might be hard to pinpoint, we show that they are definitely not negligible on the NXP LPC Cortex-M0/M3 processors. We further verify that such leakage can be exploited to recover the secret key in a masked implementation, reducing the security order effectively in practice. Unlike the transition-based leakage, such a reduction is not bounded by a factor of 2 [BGG+14], which makes it difficult to provide any practical security estimation.

Intriguingly, our experiments accidentally reveal some internal features of the NXP LPC series Cortex-M0 and M3 processors, as only one of the two shift instructions (LSL and LSR) has significantly bit-interaction leakage. We leave the further investigation of this phenomenal as an interesting open problem. As the bit-interaction leakage strongly depends on the concrete implementation details of the target processor, examining more processors leakage behaviour in this regard would also be an interesting topic for the future study (although we doubt there will be any "golden rule" in the end).

Our work serves as (another) cautionary tale for software developers who have to implement masking securely: it is imperative to verify the leakage characteristics of instructions before considering implementation options for masked implementations. For the M0 (as manufactured by STM and the LPC variation by NXP), to some extent the M3 (LPC) there exists at least one attempt of such a characterisation [MOW17] that includes information about transition and bit-wise interactions.

## Acknowledgements

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work has been supported in part by the European Commission through the H2020 project 731591 (acronym REASSURE) and EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme.

## References

[ARMa] ARM. Arm v6-m architecture reference manual. URL: https://static.docs.arm.com/ddi0419/d/DDI0419D_armv6m_arm.pdf.

[ARMb] ARM. Arm v7-m architecture reference manual. URL: https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 616–648, 2016.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 535–566, 2017.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, pages 64–81, 2014.

[Bih97] Eli Biham. A fast new des implementation in software. In Eli Biham, editor, *Fast Software Encryption*, pages 260–272, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[CBG+17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, pages 1–18, 2017.

[CEM18] Thomas De Cnudde, Maik Ender, and Amir Moradi. Hardware masking, revisited. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):123–148, 2018.

[CPRR14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *Fast Software Encryption*, pages 410–424, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[dGPdLP+17] Wouter de Groot, Kostas Papagiannopoulos, Antonio de La Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and arm: Friends or foes? In Andrey Bogdanov, editor, *Lightweight Cryptography for Security and Privacy*, pages 91–109, Cham, 2017. Springer International Publishing.

[Fur00] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GJRS18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, pages 3–22, 2018.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 567–597, Cham, 2017. Springer International Publishing.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 623–643, 2017.

[LBS19] Itamar Levi, Davide Bellizia, and François-Xavier Standaert. Reducing a masked implementation's effective security order with setup manipulations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):293–317, Feb. 2019.

[Mar13] Trevor Martin. *The Designer's Guide to the Cortex-m Processor Family*. Newnes, Oxford, 2013.

[Mes01] Thomas S. Messerges. Securing the AES DPAContestFinalists Against Power Analysis Attacks. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Fast Software Encryption: 7th International Workshop, FSE 2000 New York, NY, USA, April 10-12, 2000 Proceedings*, pages 150–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, Berlin, Heidelberg, 2007.

[MOS11] Stefan Mangard, Elisabeth Oswald, and François-Xavier Standaert. One for all - all for one: unifying standard differential power analysis attacks. *IET Information Security*, 5(2):100–110, 2011.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 199–216, 2017.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked cmos gates. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 351–365, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, pages 529–545, 2006.

[Pag] D. Page. SCALE: Side-Channel Attack Lab. Exercises. URL: http://www.github.com/danpage/scale.

[PM05] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 172–186, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[PR07] Emmanuel Prouff and Matthieu Rivain. A Generic Method for Secure SBox Implementation. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications: 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers*, pages 227–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 413–427, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[SBM18] Pascal Sasdrich, René Bock, and Amir Moradi. Threshold implementation in software - case study of PRESENT. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, pages 227–244, 2018.

[Tri03] Elena Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.

[TV04] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10246–, Washington, DC, USA, 2004. IEEE Computer Society.

[YYP+18] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018, Washington, DC, USA, April 30 - May 4, 2018*, pages 57–64, 2018.

# A "De-coupling" experiment

In order to show that coupling effect is not the main contributor in our experiments, we have conducted the same experiment in Section 3.3, without any external shunt resistor. Removing resistor is in general a difficult task on a taped-out board: fortunately, the design of the SCALE board leaves a jumper which can bypass the shunt resistor. According to the authors' theory, when there is no external resistor presented, the external coupling effect should be minimised [LBS19]. As we can see in Figure 11, since the SCALE project was designed with that resistor, the signal-to-noise ratio clearly favors the shunt resistor[6]. Nonetheless, the leakage trends still resembles Figure 6: the 1st order leakage did not vanish here, despite the fact that it took much more traces to detect it. We do **NOT** claim coupling effort does not exist in our experiments: however, we believe it is safe to say the coupling effort described in [LBS19] is hardly the one and only reason that we observe such leakage in our experiments.

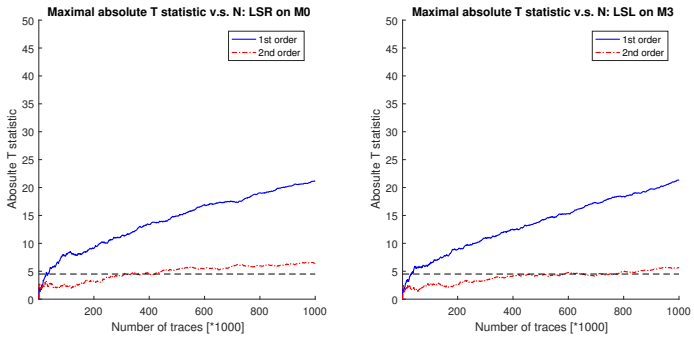


**Figure 11:** Instruction-wise leakage analysis without external resistor: 2-share LSL on M3/LSR on M0

# B Leakage analysis on other instructions

In the following, we list the other 8 instructions we tested on M0/M3. The acquisition setup is exactly the same as Section 3.3: the target data is 16 pair of 2-shares with the same unshared bit, using 16-bit Thumb-1 instructions. The significant leakage in rotation instruction "ROR" is expected: as Thumb-1 forces rotation to use the same source and destination register, the bit-flips of this register gives the unshared value. We do not claim we have a reasonable explanation for every phenomenon in Figure 12: they could be caused by the measurement setup, physical effects or more subtle micro-architectural effects. Further investigation of those effects would be an interesting topic for future work.

# C ELMO evaluation

Figure 13 demonstrates the evaluation results using the power simulation ELMO [MOW17]. Considering the algorithm noise, here we have to use a tweaked version of ELMO: only the bit-interaction term is kept in the power model; all the other terms are set to 0. Using the original ELMO model should be able to detect the same effect, although the evaluation

[6]The SCALE project uses the typical SCA measurement setup in [CEM18], where the probe is connected AFTER the resistor. [LBS19], on the other hand, places the probe BEFORE the resistor. They are the same setups where there is no shunt resistor: however, where there is a shunt resistor, it is essential to remember that the SNR in our setup is completely different than [LBS19]

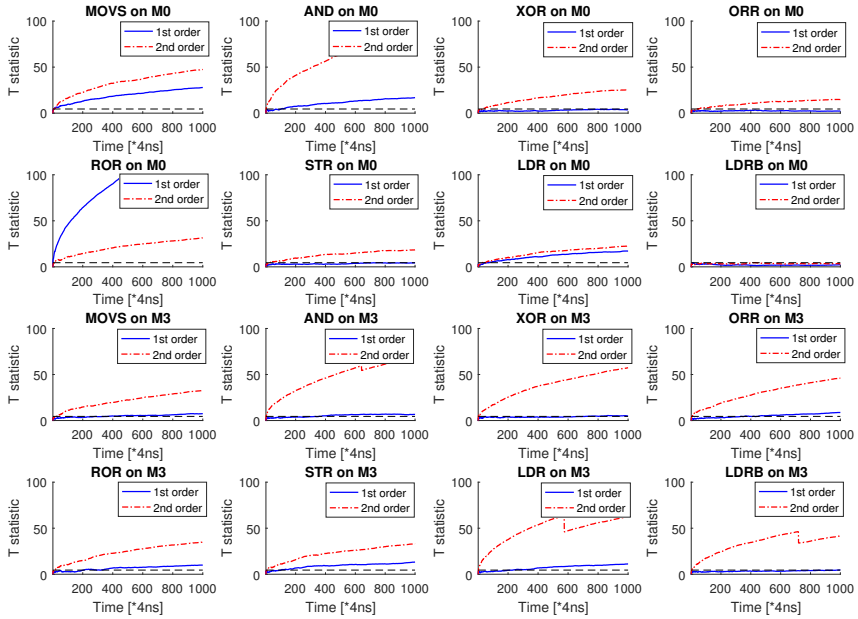


**Figure 12:** Instruction-wise leakage analysis: other instructions

can hardly be done in a reasonable time frame. Besides, setting all other terms to 0 also erases whatever transition-based leakage there could be: although this is not reflecting what is happening in practice, it certainly helps us to concentrate on the bit-interaction effect. As we can see in Figure 13, the bit-interaction effect becomes detectable within 1 million traces.

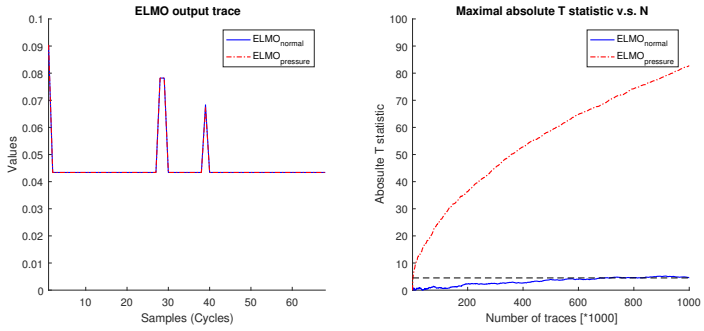


**Figure 13:** ELMO evaluation on the LSR-based 2-share multiplication

Meanwhile, Figure 13 also highlights the importance of the bit-slice strategy: the "ELMO_normal" line represents the same setup as Section 3.3: if the unshared secret bit is $q$, the 32-bit register should be filled with $(a^{(1)}, a^{(2)}, ..., p^{(1)}, p^{(2)})$, where $a^{(1)} \oplus a^{(2)} = ... = p^{(1)} \oplus p^{(2)} = q$. "ELMO_pressure", on the other hand, stands for a pressure test where the register filled with $(a^{(1)}, a^{(2)}, ..., a^{(1)}, a^{(2)})$ , $a^{(1)} \oplus a^{(2)} = q$. Clearly, Figure 13 suggests that "the pressure test" has much less algorithmic noise, which leads to more efficient detections.

One can argue whether such "pressure test" might capture certain bit-interactions

that do not affect security in practice (say the first $a^{(1)}$ and second $a^{(2)}$ might come from different blocks, therefore their joint leakage is not really a threat.). Nonetheless, even if the detected threat is not realistic, there is no doubt that this is still a violation of the "independent assumption". We believe when testing whether one can use share-slicing on a processor, it makes more sense to consider this "pressure test". The "2/4-share + other bits randomness" option in Section 4.3 and the "2/4-share + other bits constant" option [JS17] are perhaps, more realistic in terms of attacks. However, the low signal-to-noise ratio makes it difficult to detect such violation, even in a simulation-based context. When verifying the "independent assumption" on a new processor, we recommend to use such "pressure test" instead of any realistic attack oriented setup (definitely not the one we used in Section 4.3).