Sharing is Caring—On the Protection of Arithmetic Logic Units against Passive Physical Attacks

Hannes Gross

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
hannes.gross@iaik.tugraz.at

Abstract. Embedded systems are often used in security-critical scenarios where physical access of an adversary cannot be prevented. An attacker with unrestricted physical access to an embedded device could thus use observation-based attacks like power analysis or chip probing techniques to extract chip-internal secrets. In this work, we investigate how to counteract first-order passive physical attacks on an embedded microcontroller. In particular, we focus on the protection of the central point of data processing in the microcontroller design—the arithmetic logic unit (ALU)—with the provably secure threshold implementation (TI) masking scheme. Our results show that the amount of required fresh random bits—a problem many masked implementations suffer from—can be reduced to only one bit per ALU access and clock cycle. The total chip area overhead for implementing the whole microcontroller of our case study as a three-share TI is about a factor of 2.8.

Keywords: secure ALU, threshold implementation, masking, side-channel analysis, DPA, chip probing

1 Introduction

Throughout the last decades, our understanding of the term "security" became much broader. In the beginning of cryptographic research almost the whole security problematic was somehow narrowed down on finding mathematical problems that are hard to solve without the knowledge of some secret information or trapdoor function. As it turned out, physical systems are usually easier to attack through the back door than by attacking the mathematical approach that protects the front door.

In particular, when an attacker has unrestricted access to a device—for example, because she is the holder of the device or the device is operated in an area that can hardly be secured—physical attacks become a serious threat. An unprotected hardware implementation reveals its secrets through, e.g., its power consumption [8], the electromagnetic emanation [13], or an attacker could even use needles to eavesdrop on the chip internal data exchange [6].

Protecting cryptographic hardware against physical attacks is now for more than 15 years an ongoing research topic. Many different masking schemes were proposed, but because of the possible occurrence of glitches caused by the combinatorial logic in hardware, the security of the masking schemes were jeopardized. Since the introduction of threshold implementations (TI) by Nikova et al. [10] in 2006, a big step towards provable protection of hardware implementations was taken. However, until now this approach was only used to protect cryptographic hardware implementations against passive physical attacks.

In order to protect a whole system against these observation-based attacks, all components of the system that touch the assets of the system need to be considered. In the case of an embedded system, the data is usually processed by a microcontroller. More precisely, every information that is processed has to pass the arithmetic logic unit (ALU) of the microcontroller. The ALU is thus the most attractive and also the most vulnerable point to be attacked by an adversary that can physically observe a device. It is therefore crucial for the security of such a system to have an ALU protected against these kind of attacks.

Our contribution In this work, we show how a general purpose ALU design can be secured against first-order passive physical attacks. Therefore, the functionality of a typical ALU is considered and its functionality is transformed step-by-step in order to fulfill the requirements of the threshold implementation scheme. Considering each function separately is of course not enough. By carefully bringing these functions together, the chip area and the number of required fresh random bits per clock cycle can be reduced. The final ALU TI design requires about 1.1 kGE of chip area and only one random bit per clock cycle. The protected ALU is then integrated in a case study microcontroller and the required changes to the microcontroller design are discussed. Finally, the protected and unprotected hardware implementations are compared on three different levels: (1) on gate level—by comparing standard cell library gates to the TI of the gates required by the functionality of the ALU—, (2) the resulting costs of the standalone ALU TI are considered in relation to the unshared ALU, and (3) the comparison is done for the case study microcontroller. It is shown that the area overhead in the latter case is about a factor of 2.8, and the power consumption is increased by a factor of 2.3.

The rest of the paper is organized as follows: In Section 2, an overview of the threshold implementation scheme is given. In Section 3, the considered attack scenario is explained briefly. Afterwards, in Section 4, a look into the functionality provided by a general purpose ALU is given, and the functionality is then transformed in accordance with the requirements of the threshold implementation scheme. In a case study (Section 5), the integration of the protected ALU into a microcontroller design is described before the results are discussed in Section 6. Closing remarks are given in Section 7.

2 Threshold Implementations

The threshold implementation (TI) masking scheme was introduced by Nikova et al. in 2006 [10]. In contrast to many existing masking schemes, TIs are provably secure against first-order passive physical attacks—like differential power analysis attacks or chip probing—even in the presence of glitches. In the past, the TI scheme was mainly used to protect cryptographic circuits in order to make them side-channel resistant (e.g. Keccak [2], AES [9], Present [12]). The basis of TI is a technique well known from

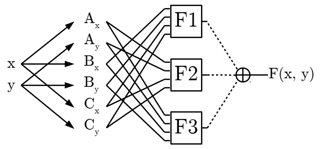

Fig. 1. Example for a shared function

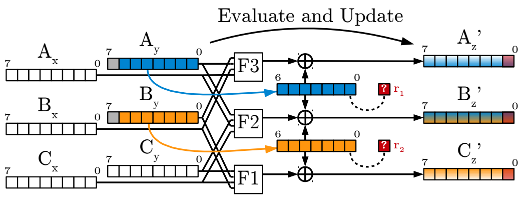

Fig. 2. Resharing of a nonuniform function

multi party computations named *secret sharing* which breaks the data dependency of the computation. As it was shown by Barak et al. [1] a calculation can never be totally independent from its underlying data, but the complexity—the order of the attack—can be increased. In the subsequent sections the term sharing and threshold implementations are used interchangeably. The secret sharing principle is illustrated in Figure 1 for a bivariate function based on the TI scheme with three shares.

Before the shared function F is applied to the two variables, the variables themselves need to be shared. In this paper we use the same notations as Bilgin et al. [2]. Accordingly, a variable *x* is shared with three shares A, B, and C so that Equation 1 is valid at any time.

$$x = A_x \oplus B_x \oplus C_x \tag{1}$$

In order to have a shared implementation that is called a valid TI, the sharing must fulfill the following three requirements: *correctness*, *non-completeness*, and *uniformity*. A sharing of a function F is said to be correct if the sum of the output shares equals F applied to the sum of the input shares (see Equation 2).

$$F_{(A_x \oplus B_x \oplus C_x)} \Leftrightarrow F1_{(B_x, C_x)} \oplus F2_{(A_x, C_x)} \oplus F3_{(A_x, B_x)} \tag{2}$$

The non-completeness property requires that each of the component functions of a function F is independent from at least one input share like it is shown in Figure 1 and Equation 2. Fulfilling the uniformity property is usually the hardest task, because a uniform sharing of F requires that under the assumption of a uniformly distributed input sharing, the resulting output shares of the component functions are again uniformly distributed.

Even for a simple boolean function, like *AND* or *OR*, no uniform sharing exists for a three-share TI as it was shown by Nikova et al. In this case, additional random bits are required to recover the uniformity of the output shares. As an alternative, a higher amount of shares could be used, but this has the drawback of a higher resource footprint. However, the requirement of many high-quality fresh random bits with a high entropy at each clock cycle is also not quite practical. In 2013, Bilgin et al. [2] showed that the majority of the required random bits for the Keccak TI can be taken from independent shares by using Lemma 1.

Lemma 1. ***(Simplified version from [2]).*** *Let (A, B, C) be n-bit shares (not necessarily uniform). Let (D, E, F) be uniform n-bit shares statistically independent of (A, B, C). Then, (A+D, B+E, C+F) are uniform n-bit shares.*

Figure 2 shows how the uniformity of a nonuniformly shared function F—where each of the component functions are not uniform—can be repaired according to Lemma 1, when F is a bit-wise function. Since each of the bits of the shares A, B, and C are independent from the bits with a different index, the bits from the input shares of F can be used to reshare the output of the component functions of F. This is done by adding uniformly distributed and statistically independent bits—from the input shares and the two random bits—to the resulting output shares. The random bits are necessary to avoid dependencies between the bits. The result is a correct and uniform sharing at the output. When only one or two of the component functions are not uniform, it is enough to apply Lemma 1 to only two of the component functions which then requires just one high-quality random bit.

3 Attack Scenario

Figure 3 shows a typical embedded system scenario—considered to be integrated on a single microchip. The illustrated system could be, e.g., part of a technical process, and interacts with its direct environment through sensors and actuators. The core of the embedded system is a microcontroller that exchanges data with other modules connected to the I/O bus.

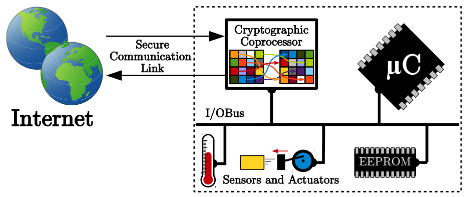

Fig. 3. Embedded system scenario

When a message is received or needs to be sent—e.g. to a server on the Internet—the cryptographic coprocessor handles security services like authentication, confidentiality protection, message integrity, et cetera. Therefore, the communication channel is considered to be secure. However, in this scenario we assume a powerful attacker that can not only spy on the network communication of the embedded system, but can also physically observer the device. Such an attacker could measure the power consumption of the system, place an electromagnetic probe on the device, or even put a needle on the I/O bus or on any other of the system's component in order to reveal any data flowing inside the system. Nevertheless, the adversary is assumed to be constraint in its observations and attacks to first-order scenarios. The attacker could thus either probe a single chip wire or execute a first-order power analysis attack. This is a typical assumption one has to make, because the costs for protecting the device, the attack costs, and the value of the protected assets should be considered to find adequate countermeasures.

The information an attacker is interested in could be, e.g., the communicated message between the system and the outside world, configuration parameters or cryptographic keys stored inside the non-volatile memory, et cetera. Regardless of the origin of the data, as soon as the data is processed inside the system it needs to pass the ALU of the microcontroller.

Protecting the system with the TI scheme hinders an attacker—limited to first-order attacks—to succeed. In the following it is assumed that security-critical data is either stored in the system or transported inside the system—e.g. over the I/O bus—in shared form. The targeted representation of the data input and output signals for the TI of the ALU is thus a uniform sharing. The number of used shares thereby determine the overhead of the TI relative to the unshared implementation, because the number of registers increases linearly with the number of required shares. Therefore, it is desirable to have an implementation with only three shares, which is the minimum number for first-order secure TIs. For a more sophisticated attacker the approach could be extended to higher-order threshold implementations like it was shown by Bilgin et al. [3] on the costs of higher hardware requirements. As a precondition, it needs to be considered that at no point of the system the data signal sharing is violated (demasking of the data), because an attacker could exploit this signal, e.g., for a simple probing attack. This condition leads to the requirement that all data input signals are already uniformly shared before they are handed to the ALU.

4 Protection of the Arithmetic Logic Unit

Since the arithmetic logic unit (ALU) is the central point of computation in all processor designs—like embedded microcontrollers—it is also the most vulnerable point for attacks on security-critical data. In the following, the typical functionality of a general purpose ALU is described in detail. The remainder of this section is then spent on answering the question how the ALU functionality can be implemented according to the threshold implementation (TI) masking scheme, and how these functions can be efficiently put together.

4.1 ALU Description

Figure 4 shows an ALU like it is used in similar form in many common processor architectures. The ALU basically consists of three stages. At first, the operands are selected from the ALU inputs. Then a preprocessing stage optionally inverts, shifts, or rotates an operand. Additionally, individual bits of the operands can be picked by using *bit-select* masks. Finally, the selected operands are applied to a so-called *functions array* that then executes the selected arithmetic or logic function.

In general there are two types of signals the ALU has to deal with: the control signals (red) and the data signals (black). The majority of signals are originated from data. The operands of the ALU can either be one or two selected registers, constants (from the program memory), the internally generated bit-select mask value, or the carry bit. For unary operations there is also a *"0"* operand selectable by the first operand multiplexer. Additionally, the first operand can also be inverted before it is applied to

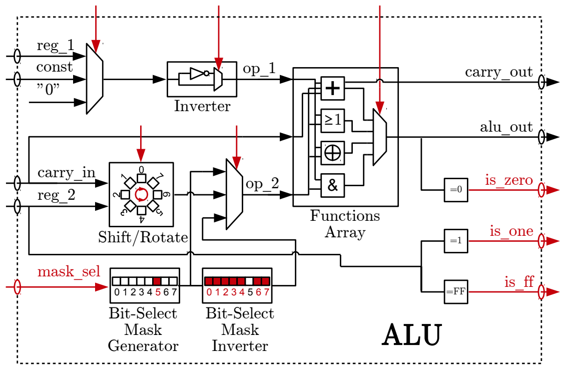

Fig. 4. Overview of the considered ALU (signals colored in red are control signals)

the functions array of the ALU. The second operand can be either shifted or rotated before the two operands are combined. In order to test or set a single bit of a register value, the *bit-select mask generator* provides a one-hot or one-cold encoded bit vector. The core functionality of the ALU is built by the functions array which consists of an arithmetic adder, and the three logic operations *AND*, *OR*, and *XOR*. The computed value is then available at the output of the ALU. Control signals, on the other hand, are either used by the controller to select the desired functionality of the ALU or are generated by the ALU to inform the controller about certain events, like data overflows or comparison results.

4.2 Sharing of the ALU

In the following the sharing of the ALU functions is explained in detail. Therefore, it is assumed that all data input signals of the ALU are uniformly shared. This requirement is decisive for the correctness of the TI implementation.

Functions Array, Inverter, and Shift/Rotate Sharing of the bit inverter and the *XOR* function of the *functions array* is straight-forward and was already shown before, e.g., by Nikova et al. [10]. For the inverter, it is sufficient to invert one of the shares of the selected operand. The logic *XOR* of two shared operands can be calculated by applying a logic *XOR* operation pairwise on the shares, e.g., $x \oplus y = (A_x \oplus A_y) \oplus (B_x \oplus B_y) \oplus (C_x \oplus C_y)$. The uniformity in both cases is not violated, and a resharing of the result is thus not required. Since, shifting and rotating also does not produce any new data the uniformity is guaranteed as long as the input is shared uniformly.

Sharing of the logic *AND* and *OR* operation is somewhat trickier. To the best of our knowledge, there exists no work that lists or analyzes all possibilities for sharing the logic *AND* and *OR* operations with three shares. In order to find the sharings, either a mathematical approach can be taken, like it was shown by Bilgin et al. [4], or an exhaustive search can be performed to search through all possibilities. It was shown by Nikova et al. [11] that for nonlinear functions, no uniform sharing with three shares exists. However, if a nonuniform sharing is found, it can be repaired by using so-called virtual variables. The equations for the shared functions is then extended by terms that contain random bits in such a manner that the uniformity is given again.

The exhaustive search approach used for this work to find the sharing with the lowest hardware costs for the logic *AND* and *OR* is described in Appendix A. In total eight possible (non equivalent) sharings for the logic *AND* and *OR* functions were found as it is listed in Table 1 and Table 2, respectively.

Table 1. Sharings for a logic *AND* function: $F_{(A_x \oplus B_x \oplus C_x,\, A_y \oplus B_y \oplus C_y)}$

	$F1_{(B_x,B_y,C_x,C_y)}$	$F2_{(A_x,A_y,C_x,C_y)}$	$F3_{(A_x,A_y,B_x,B_y)}$
AND-Sharing #1	0000001101010110	1001010100110000	**0001110110111000**
AND-Sharing #2	0000001101010110	**1101000101110100**	0101100111111100
AND-Sharing #3	0000001101011001	1101000101111011	0101100111111100
AND-Sharing #4	0000001110101001	**1101000110001011**	0101100111111100
AND-Sharing #5	0000110010101001	1101111010001011	0101100111111100
AND-Sharing #6	**0001110110111000**	1101111010001011	0101100111110011
AND-Sharing #7	0011111110011010	1101111010001011	0101100100000011
AND-Sharing #8	0111101111011110	1101111010001011	0101011000000011

Table 2. Sharings for a logic *OR* function: $F_{(A_x \oplus B_x \oplus C_x,\, A_y \oplus B_y \oplus C_y)}$

	$F1_{(B_x,B_y,C_x,C_y)}$	$F2_{(A_x,A_y,C_x,C_y)}$	$F3_{(A_x,A_y,B_x,B_y)}$
OR-Sharing #1	0000001101010110	1001101011000000	**0111010000101110**
OR-Sharing #2	0000001101010110	**1101111010000100**	0011000001101010
OR-Sharing #3	0000001101011001	1101111010001011	0011000001101010
OR-Sharing #4	0000001110101001	1101111001111011	0011000001101010
OR-Sharing #5	0000110010101001	1101000101111011	0011000001101010
OR-Sharing #6	**0001110110111000**	1101000101111011	0011000001100101
OR-Sharing #7	0011111110011010	1101000101111011	0011000010010101
OR-Sharing #8	0111101111011110	1101000101111011	0011111110010101

Each entry in these tables shows the output bit combination that appears when iterated over the input variables. Some of the component functions are already uniform (emphasized in the table). In this case only the two non-uniform component functions need to be repaired by simply adding one random bit as a virtual variable to the output of the nonuniform functions (see Figure 2). For the case where all three component functions are non-uniform, either two virtual variables could be used, or a nonlinear combination of one virtual variable with other variables leads to a uniform sharing. Anyway, as it turns out in our hardware implementation evaluation (see Section 6), these sharings are always more hardware demanding than the sharings with already one uniform component function. The extension of the 1-bit shared logic operations to the required n-bit is trivial, since the bits are independent and hence the shared gates can be put in parallel.

Adders and the "0" Operand A simple half-adder takes per definition two input bits and calculates one sum bit and further on a carry bit. The implementation of a half-adder can be realized by using an *XOR* gate for the resulting sum bit and an *AND* gate to calculate the carry. Full-adders—as they are required for the ALU design—which also take a carry bit as an input, can be built of two half-adders and an *OR* gate (see Figure 5).

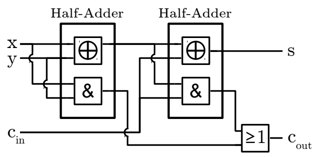

Fig. 5. Schematic of a full-adder built from two half-adders and one logic *OR*

A straightforward way of implementing a shared full-adder would be to use the already obtained shared logic gates to implement the half-adders and the logic *OR*. However, this would require fresh random bits for each of the full-adders in the design to guarantee uniformity. Nevertheless, Figure 5 shows that for the calculation of the sum bit "s" only *XOR* gates are required, and thus the corresponding s-bit output sharing is already uniform if the inputs are uniformly shared. The carry bit on the other hand can be realized by Equation 3.

$$\begin{aligned} c_{out} &= (x \wedge y) \vee (x \oplus y \wedge c) \\ &= (x \wedge y) \vee (x \wedge c) \vee (y \wedge c) \end{aligned} \tag{3}$$

The shared form of this equation is presented in Equation 4, and is obtained by using a direct sharing approach.

$$\begin{aligned} (A_{Cout} \oplus B_{Cout} \oplus C_{Cout}) = &\big((A_x \oplus B_x \oplus C_x) \wedge (A_y \oplus B_y \oplus C_y) \big) \vee \\ &\big((A_x \oplus B_x \oplus C_x) \wedge (A_c \oplus B_c \oplus C_c) \big) \vee \\ &\big((A_y \oplus B_y \oplus C_y) \wedge (A_c \oplus B_c \oplus C_c) \big) \end{aligned} \tag{4}$$

By dissolving the brackets and by grouping the terms according to the non-completeness rule, the sharing of the carry results in Equations 5-7.

$$\begin{aligned} A_{Cout} = &(B_x \wedge B_y) \oplus (B_x \wedge C_y) \oplus (B_y \wedge C_x) \oplus \\ &(B_x \wedge B_c) \oplus (B_x \wedge C_c) \oplus (B_c \wedge C_x) \oplus \\ &(B_c \wedge B_y) \oplus (B_y \wedge C_c) \oplus (B_c \wedge C_y) \end{aligned} \tag{5}$$

$$\begin{aligned} B_{Cout} = &(C_x \wedge C_y) \oplus (A_x \wedge C_y) \oplus (A_y \wedge C_x) \oplus \\ &(C_x \wedge C_c) \oplus (A_x \wedge C_c) \oplus (A_c \wedge C_x) \oplus \\ &(C_c \wedge C_y) \oplus (A_y \wedge C_c) \oplus (A_c \wedge C_y) \end{aligned} \tag{6}$$

$$\begin{aligned} C_{Cout} = &(A_x \wedge A_y) \oplus (A_x \wedge B_y) \oplus (A_y \wedge B_x) \oplus \\ &(A_x \wedge A_c) \oplus (A_x \wedge B_c) \oplus (A_c \wedge B_x) \oplus \\ &(A_c \wedge A_y) \oplus (A_y \wedge B_c) \oplus (A_c \wedge B_y) \end{aligned} \tag{7}$$

The sharing of the carry bit is again uniform, and does not require additional random bits. The question that still remains is whether or not the joint distribution of the shared sum and carry bits are uniform. A nonuniform joint distribution would leak information about the operands and could thus be considered in a power model. By iterating over all input share combinations for each pair of the component functions of the sum and carry bits, it can be shown that also the joint distribution of these bits are also uniform. The shared implementation of the full-adder can therefore be realized without using any additional random bits at all if the input shares are uniform.

From 1-bit Adders to n-bit Adders The circumstance that an addition—in contrast to the boolean operations—needs to consider all bit positions together to calculate the correct sum, leads to the observation that the calculations cannot be performed in a single clock cycle—like it was shown recently by Schneider et al. [14]. Putting a series of shared full-adders together to from an n-bit adder violates the independence requirement of the TI, because the calculations then no longer depend on only two shares. The consequences could be data dependent power glitches that lead to a vulnerability to first-order attacks. Consequently, the calculations of the component functions must be separated by register stores.

While there exist many possible adder designs with different benefits and drawbacks—Schneider et al. [14] showed two possible realizations—, we decided on a straight-forward iterative approach, because of the rather low data width of the targeted ALU design. The shared 8-bit adder design is shown in Figure 6.

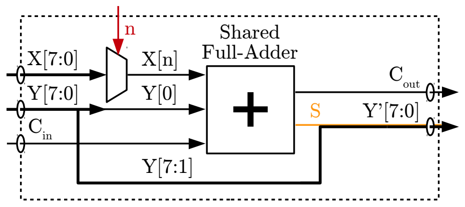

Fig. 6. Iterative Shared 8-bit Adder (upper case variables are shared)

At the input, the multiplexer selects one bit of the shared operand X beginning with the least-significant bit. The source of the Y operand is always a register which also servers as the destination for the result. The two shared operand bits and the carry are then applied to the shared full-adder. The calculated carry is stored and then reused in the subsequent iteration. For the new register content, only the current bit position of the shared operand is overwritten with the resulting sum bit S and the bits are reordered in each iteration. The number of required clock cycles thus increase linearly with the number of used bits. However, the benefits of this approach are the very low area footprint and that no additional random bits are required.

Increment and Decrement A special case of the addition operation are the increment and the decrement instructions. In this case the first operand is set to zero and the inverter control signal and the carry input bit are set accordingly. Setting the first operand and the carry to a fixed value seems to be a violation against the uniformity requirement at first sight. Nevertheless, by doing this the right way the uniformity property can be saved.

For a uniform implementation of the increment instruction, the shares of the first operand (A_x, B_x, and C_x) are set to zero, the inverter control is set to "by-pass", and all bits of the *carry_in* shares (A_c, B_c, and C_c) are set to one. The uniformity of the sum bit is given, since for its calculation only the shares of the second operand (A_y, B_y, and C_y) are inverted because of the carry bit shares. The carry output bit, however, only depends on the value of the second operand which can be demonstrated by inserting the afore mentioned changes into Equations 5-7. Since the output of the first full-adder is uniform, so is the output of the next full-adder, et cetera.

For the decrementation instruction, the first operand's shares (A_x, B_x, and C_x) are again set to zero but then inverted by setting all bits of the A_x share to one through the inverter. Furthermore, the A_c share of the *carry_in* is set to zero and all other shares to one.

Bit Set/Bit Clear and the Bit-Select Mask Generator In the unshared design, the *Bit-Select Mask Generator* basically performs a one-hot encoding of the *mask_sel* value. This mask is then used to set or clear one specific bit. Setting one bit high is then performed by selecting the logic *OR* operation from the functions array. For clearing one bit, the inverse mask is used. The selected register is finally combined with the mask by using the *AND* operation.

Applying the same operations on shared values requires some changes in the ALU. One possibility for implementing shared n-bit select masks is shown in Equation 8.

$$\begin{aligned} M_1 &= 000 \cdots (A_x \oplus B_x) \cdots 000_n \\ M_2 &= 000 \cdots (A_x \oplus [1]) \cdots 000_n \\ M_3 &= 000 \cdots B_x \qquad\quad \cdots 000_n \end{aligned} \tag{8}$$

First of all, the bit-select mask value generation already needs to take the desired operation into account. This is done by adding an additional "1" to the desired bit position for M_2 when the bit needs to be set. All other bits of the shares are set to zero. The resulting bit-select mask is then applied to the operand by using the *XOR* operation of the functions array in either case. For the output sharing, only the targeted bit position changes.

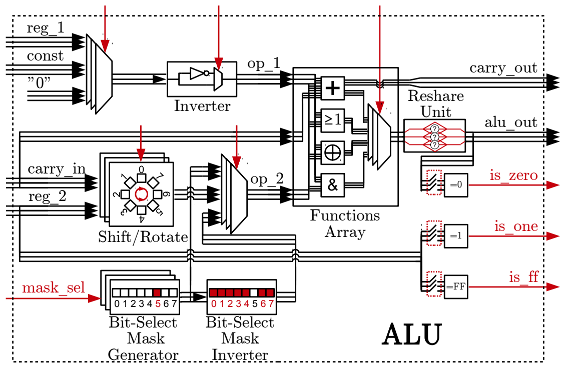

Fig. 7. Overview of the TI of the ALU

Sharing of Control Signals The ALU usually generates some control signals required for the control unit of a processor to handle, e.g., conditional branch instructions. These instructions compare the output of the ALU or the content of one register to a fixed value (e.g., the zero and overflow flags). This could be realized by bringing the comparison equation to a disjunctive normal form. In so doing, this results in a cascaded network of logic *AND* gates with a gate count of $n - 1$ and a delay path depth of $log_2(n)$—for a comparison equation with n (= power of two) variables. Even though it is possible to share these control signals, it was shown in the past that branches can be detected quite easily by an attacker. Branches can be detected through changes in the program flow structure which affect the power traces or the execution time of the program. It is thus essential to avoid branches or conditional code execution which depend on the data one wants to protect. Cryptographic software implementations, e.g. for the AES, could use table lookups either in the program memory or the data memory to speed up their implementation. These issues are well know and are therefore usually avoided by programmers aware of side-channel attacks (see e.g., Käsper et al. [7]). Sharing of control signals will thus not lead to a better protection and can thus be excluded from the TI of the ALU. The software developers are responsible to program the software in a way that it is free of branches which rely on security critical data. However, it needs to be ensured that the creation of data dependent control signals is suppressed when critical data is processed, e.g., by using operand isolation techniques in case a branch instruction is fetched.

Bringing the Findings Together Besides the described shared implementation of the functions and the tripled signal paths, some new components are introduced in the final design of the ALU (see Figure 7). A new module called *reshare unit* is added which does the repairing of the non-uniform boolean functions by adding the virtual variables to the output of the functions array. Since, only one of the functions from the functions array is executed at a time, the reshare unit needs to be implemented only once. Therefore, only one fresh random bit is required—for the reparation of the *OR* and *AND* function—, and the rest of the virtual variables are taken from independent shares similar to Figure 2. So, the TI of the ALU requires one fresh random bit in each clock cycle to guarantee uniformity at the output. Furthermore, in order to thwart an unintended unmasking of the shared data, the comparison units are isolated from the operands in any case other than a branch instruction is fetched. This is done by gating the input signal of the comparison units which is controllable by a single input control signal. In the next section, the protected ALU is integrated into a case study microcontroller design and the required implications are discussed.

5 Case Study: Microcontroller

The microcontroller used in the case study of this paper is illustrated in Figure 8. It is an 8-bit microcontroller with a Harvard architecture design and has a separate storage for program code and data, accordingly. Instructions from the program memory are decoded in the *control FSM* unit that generates the control signals for the ALU, the *register file*—consisting of up to 64 registers—, and the *program counter* (PC). Overall, there are just about 30 instructions available in the standard instruction set. However, the microcontroller is completely configurable in terms of program memory size, size of the register file, and also the instruction set can be extended or reduced easily to the needs of certain applications. The register file has three special purpose registers, and a configurable size of general purpose (GPR) and I/O registers. I/O registers have an additional external feedback path that is used to extend the microcontroller with peripherals, or an additional RAM, or a cryptographic coprocessor.

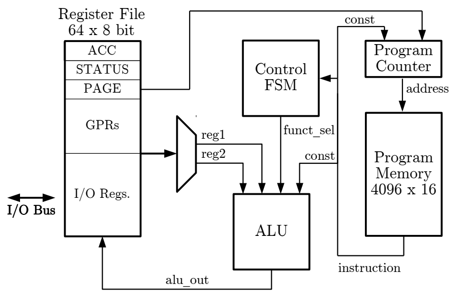

Fig. 8. Overview of the used 8-bit microcontroller

Figure 8 also shows how the ALU from Section 4 is integrated into the microcontroller design. All data inputs of the ALU need to be shared before the processing in the ALU is performed. For data that needs to be protected, it is assumed that this data is either stored (and updated) on-chip in a shared representation—e.g. in a non-volatile memory—or that the data is transported into the microcontroller in a uniformly shared way.

The origin of the data input signals of the ALU can either be the internal registers or the I/O registers, or the program memory. The internal registers are successively filled with the shared data output of the ALU or with data provided by the I/O modules. The register file size is therefore tripled, and I/O modules connected to the microcontroller need to be prepared for handling shares. For example, if the microcontroller is connected to a cryptographic coprocessor that secures the external communication, the design of the coprocessor also needs to be a TI. The data between the microcontroller and the cryptographic coprocessor is then exchanged in a shared form over the I/O bus.

For the sake of simplicity it is assumed that data inside program memory is shared and updated accordingly. The program memory therefore needs to be writable and non volatile. One possible realization is the usage of ferroelectric RAM (FRAM) like it is used in the MSP430F microcontroller family. FRAM is extremely fast—50 ns for writing, according to [5]—and has a very low power consumption. How the program memory sharing and the updating is managed is out of scope for this work. In this case study we are mainly interested in estimating the overhead for implementing the core of the microcontroller as a threshold implementation, since the program memory requirements vary from application to application.

6 Results

In this section, the impact of the shared components of the threshold implementation is investigated on three different abstraction levels. At first the influence of the TI on gate level is considered which shows how the sharing changes the requirements regarding chip area, power consumption, and timing constraints. In a more macroscopic juxtaposition the costs for the ALU described in Section 4 are then considered. Finally, the microcontroller of the case study in Section 5 is compared with and without shared components. All results are collected for a 90 nm UMC Standard Performance Low-K ASIC library from Faraday for a global clock of 10 MHz and a 1 V power supply. For the synthesis step, the Cadence Encounter RTL Compiler Version v08.10-s28 and for the routing the Cadence NanoRoute v08.10-s155 are used.

Comparison on Gate Level Table 3 shows the costs for a sharing of the *XOR*, *AND*, *OR* and *full-adder* cells relative to the costs of the standard-cell library gates. The required chip area is measured in multiples of a gate equivalent (= the size of a single two-input NAND gate). The maximum clock frequency is calculated on the basis of the remaining slack for the longest combinatorial path in the circuit at the targeted clock period. Therefore, it should not be seen as a total maximum but only with regard to the used gates. Gates with a higher driving strength would of course increase the maximum possible clock frequency, but this would also influences the chip area and power consumption. The power consumption is estimated with Cadence Encounter which only allows a rough estimation of the power consumption, because no transistor-level power information is considered. However, since these values are only compared in relation to each other, it should be sufficient and accurate enough to show how a sharing influences the power consumption.

Since the shared *XOR* gate requires only two additional XOR gates in parallel, the overall size triples but the propagation delay stays unchanged. The effect on the consumed power is with an overhead factor (OHF) of almost 36 more noticeable. In the case

Table 3. Comparison of standard library cells and their shared implementations

Function	Implementation	Area *[GE]*	Area OHF	max. Frequency *[GHz]*	Delay OHF	Power [nW]	Power OHF
XOR	std-cell	2.50	3.00	22.73	1.00	22.73	35.94
	shared	7.50		22.73		816.77	
AND	std-cell	1.25	13.40	27.03	3.54	57.50	25.93
	shared	16.75		7.63		1,491.21	
OR	std-cell	1.25	12.80	23.26	2.88	140.14	11.64
	shared	16.00		8.06		1,631.40	
Full-Adder	std-cell	8.25	10.67	6.58	1.94	777.00	11.93
	shared	88.00		3.39		9,268.83	

of a logic *AND* the best results in all categories are obtained from implementation #1 of Table 1. The area overhead factor is in this case 13.4 which means that a standard cell library *AND* gate is 13.4 times smaller. Due to the cascaded combination of the gates in the shared *AND* gate, the delay path increases only to a factor of 3.54, and therefore also the maximum possible clock frequency is just lowered according to this factor. In terms of power, the shared variant consumes almost 26 times more power because of the increased gate count on one side but also due to the higher capacitances at the load side of the gates. Compared to the shared logic *AND*, the shared logic *OR* implementation is a bit cheaper with an area OHF of 12.8, but also the propagation delay and the power increase is comparably smaller. Whats also very interesting, is the fact that the variants of the shared logic gates where already one component function is uniform result in a smaller design than the variants with three nonuniform component functions[1]. The area overhead factor of the logic *AND* variant #1 and variant #3 of Table 1 is 13.4 and 17.4, respectively.

The most complex shared function is the full-adder. Nevertheless, the sharing requires “only” 10.7 times more area than the standard cell, and lowers the maximum clock frequency to a factor of 1.9. Also the effect on the power is relatively small compared to other shared gates.

Comparison of the ALU Designs Table 4 shows a comparison between the shared and unshared ALU. It can be seen that the size of the ALU changes from 249 GE to 1.14 kGE. The area OHF in this case is about 4.56. The longest combinatorial path, however, is reduced because of the iterative adder design. Please note that additions take now eight cycles instead of one. Furthermore, the power increases to a factor of almost five.

Table 4. Comparison of the shared and unshared ALU

Component	Implementation	Size *[GE]*	Area OHF	max. Frequency *[MHz]*	Delay OHF	Power *[*µW]	Power OHF
ALU	unshared	249	4.56	725.69	0.99	34.60	4.98
	shared	1,135		731.53		172.40	

[1] Please note that only the best candidates for each shared gate are considered in the respective tables for brevity and clarity reasons.

Comparison of the Microcontroller Designs In this comparison the costs for the shared microcontroller core including the ALU, the register file, the control FSM, and the program counter are evaluated. This comparison interestingly shows how the influence of the TI to the resource costs is relativized compared to Tables 3 and 4. The influence of the program memory and peripherals connected to the microcontroller are left out for this considerations, because they strongly vary from application to application. Table 5 shows that the relative area increase in the shared case sinks below a factor of three. The reason for this is the still relatively small size of the ALU compared to the rest of the microcontroller. A bigger influence to the overall size has the shared register file which is almost twelve times bigger than the shared ALU. The maximum working frequency of the shared microcontroller is still 191 MHz. Also the relative power consumption overhead is not as highly influenced as in the previous comparison with a factor of about 2.3.

Table 5. Comparison of the shared and unshared Microcontroller

Component	Implementation	Size *[GE]*	Area OHF	max. Frequency *[MHz]*	Delay OHF	Power [μW]	Power OHF
Microcontroller	unshared	5,216	2.80	198.14	1.04	246.88	2.27
	shared	14,605		191.09		560.65	

7 Conclusions

In this work, we showed how a general purpose arithmetic logic unit (ALU) can be implemented in a way that it resists first-order passive physical attacks. Therefore, we looked at the functionality of a typical ALU and discussed then how this functionality can be realized in accordance to the provably secure threshold implementation (TI) scheme. While the protection of cryptographic implementations with the TI scheme was already addressed in the literature, the shared implementation of the more versatile functionality provided by an microcontroller has not yet been researched. A problem that often limits the usability of masked implementations is the required amount of fresh random bits. It was shown that by combining the ALU functions in an efficient way, the number of required random bits can be reduced to only a single bit per clock cycle. The protected ALU was finally integrated into a case-study microcontroller design and the impact of the sharing on other modules of the microcontroller investigated. Nevertheless, the shared implementation of the microcontroller should not be considered as an all-round carefree package for software designers. There are some pitfalls one can run into, like branches depending on security sensitive data, table lookups, et cetera. The whole spectrum of implications that result from a shared microcontroller design as well as a side-channel evaluation is part of future research.

Acknowledgements. This work has been supported by the Austrian Science Fund (FWF) under the grant number TRP251-N23 (Realizing a Secure Internet of Things - ReSIT) and the FFG research program SeCoS (project number 836628).

References

1. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)Possibility of Obfuscating Programs. *J. ACM*, 59(2):6:1–6:48, May 2012.
2. B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. V. Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, pages 187–199, 2013.
3. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-Order Threshold Implementations. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343. Springer Berlin Heidelberg, 2014.
4. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold Implementations of All 3x3 and 4x4 S-Boxes. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin Heidelberg, 2012.
5. T. Instruments. *FRAM FAQs*, 2014 (accessed February 10, 2015).
6. Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer Berlin Heidelberg, 2003.
7. E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2009.
8. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, 1999. Springer-Verlag.
9. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT'11, pages 69–88, Berlin, Heidelberg, 2011. Springer-Verlag.
10. S. Nikova, C. Rechberger, and V. Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In P. Ning, S. Qing, and N. Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer Berlin Heidelberg, 2006.
11. S. Nikova, V. Rijmen, and M. Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *Journal of Cryptology*, 24(2):292–321, 2011.
12. A. Poschmann, A. Moradi, K. Khoo, C.-W. Lim, H. Wang, and S. Ling. Side-Channel Resistant Crypto for Less than 2, 300 GE. *J. Cryptology*, 24:322–345, 2011.
13. J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer Berlin Heidelberg, 2001.
14. T. Schneider, A. Moradi, and T. Gneysu. Arithmetic Addition over Boolean Masking - Towards First- and Second-Order Resistance in Hardware. Cryptology ePrint Archive, Report 2015/066, 2015.

A Searching for Efficient TI Realizations of Boolean Functions

In Section 4.2, we mentioned our approach for finding possible realizations of shared boolean functions based on an exhaustive search. Since, the results in Table 1 and Table 2 can be interpreted more easily if the underlying approach is clarified, we briefly step through the search strategy in the following.

The exhaustive search approach used for this paper to find the sharing with the lowest hardware costs for the logic *AND* and *OR* is based on the simple observation that the structure of the component functions is defined by the requirements for TIs: (1) each of the three component functions of F for a three-share TI can have at most four input bits from two independent sharings (non-completeness), (2) the output of the component functions is always one bit, and (3) added together these functions must output the same value as the original function (correctness). A shared boolean function with three shares can therefore be fully described by the following truth tables.

B_x	B_y	C_x	C_y	$F1$
0	0	0	0	b_0
0	0	0	1	b_1
0	0	1	0	b_2
0	0	1	1	b_3
0	1	0	0	b_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	0	1	1	b_{11}
1	1	0	0	b_{12}
1	1	0	1	b_{13}
1	1	1	0	b_{14}
1	1	1	1	b_{15}

A_x	A_y	C_x	C_y	$F2$
0	0	0	0	b_0
0	0	0	1	b_1
0	0	1	0	b_2
0	0	1	1	b_3
0	1	0	0	b_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	0	1	1	b_{11}
1	1	0	0	b_{12}
1	1	0	1	b_{13}
1	1	1	0	b_{14}
1	1	1	1	b_{15}

A_x	A_y	B_x	B_y	$F3$
0	0	0	0	b_0
0	0	0	1	b_1
0	0	1	0	b_2
0	0	1	1	b_3
0	1	0	0	b_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	0	1	1	b_{11}
1	1	0	0	b_{12}
1	1	0	1	b_{13}
1	1	1	0	b_{14}
1	1	1	1	b_{15}

In particular, it is sufficient to describe each realization of the component function by its output-bit combination $b_{15\ldots0}$, which is used for Table 1 and Table 2. We can thus search through all possible realizations of the function F by iterating over all 2^{16} possible output-bit combinations for $F1$ and $F2$. The output bit combination of $F3$ then automatically results from $F1$ and $F2$. For the iteration over the realizations for $F2$, we can save half of the iterations because they are already covered by iterating over $F1$. The overall search effort for all possibilities to implement a three-share TI of F is hence 2^{31}.