Concealing Secrets in Embedded Processors Designs

Hannes Gross, Manuel Jelinek, Stefan Mangard,
Thomas Unterluggauer, and Mario Werner

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

Abstract. Side-channel analysis (SCA) attacks pose a serious threat to embedded systems. So far, the research on masking as a countermeasure against SCA focuses merely on cryptographic algorithms, and has either been implemented for particular hardware or software implementations. However, the drawbacks of protecting specific implementations are the lack of flexibility in terms of used algorithms, the impossibility to update protected hardware implementations, and long development cycles for protecting new algorithms. Furthermore, cryptographic algorithms are usually just one part of an embedded system that operates on informational assets. Protecting only this part of a system is thus not sufficient for most security critical embedded applications.

In this work, we introduce a flexible, SCA-protected processor design based on the open-source V-scale RISC-V processor. The introduced processor design can be synthesized to defeat SCA attacks of arbitrary attack order. Once synthesized, the processor protects the computation on security-sensitive data against side-channel leakage. The benefits of our approach are (1) flexibility and updatability, (2) faster development of SCA-protected systems, (3) transparency for software developers, (4) arbitrary SCA protection level, (5) protection not only for cryptographic algorithms, but against leakage in general caused by processing sensitive data.

Keywords: protected CPU, domain-orientend masking, masking, side-channel protection, threshold implementations, RISC-V, V-scale.

1 Introduction

The resistance of security-critical systems against the broad field of passive physical attacks is a fundamental requirement of todays embedded devices and smart cards. If an attacker has direct or indirect physical access to an unprotected device, the observation of side-channel information (like power consumption [12] or electromagnetic emanation [16]) leaks information on the processed data. The security of such devices is then no longer guaranteed even if state-of-the-art cryptography is in place, because sensitive information like the used key material leaks through side-channel information.

The history of countermeasures against side-channel analysis attacks (SCA) is as old as the first paper targeting differential side-channel analysis by Kocher *et al.* [12]. Hereby, masking has become the first-choice measure to defeat SCA. The first masking approach was introduced by Goubin et al. [6], but many schemes followed like the Trichina gate [22] approach and the works of Ishai *et al.* [10], who introduced the concept of private circuits. However, many masking schemes have shown to be insecure in the presence of glitches that occur within the combinatorial logic of hardware implementations.

To overcome the inherent issue of glitches of these masking schemes, Nikova *et al.* [14] introduced the first-order secure threshold implementation (TI) masking scheme. However, in comparison with software masking schemes, the original TI requires a higher number of random shares to handle glitches. A higher demand for fresh random shares goes hand in hand with increased hardware costs and higher randomness requirements, especially for implementations secure against higher-order attacks.

Most recently many works were published on the implementation of masked hardware implementations with reduced number of shares [2, 3, 8, 15, 18]. The work of Gross *et al.* [8] introduced the so-called domain-oriented masking scheme that requires only $d + 1$ shares, $d(d + 1)/2$ fresh randomness, and allows easy generalization to arbitrary protection orders.

Even though the trend to reduce the amount of shares to $d + 1$ made protected hardware implementations more efficient and resulted in generic higher-order implementations, the efficient protection against SCA is still cumbersome, requires a lot of expertise for both implementation and evaluation, and is error-prone. Furthermore, the reduction of shares introduces additional register stages due to the decomposition of complex functions into a couple of algebraically simpler subfunctions [2]. This circumstance of additional delay cycles naturally brings implementations based on hardware masking schemes closer to software masking schemes in terms of throughput.

The aforementioned issues when implementing efficiently masked applications motivated our work. In particular, we investigate the interesting question: Is it possible to construct a general-purpose processor that is inherently secure against side-channel analysis without giving up the benefits and flexibility of software-driven design? As far as we know, there exist only a few works that targets the protection of processors against SCA [7, 17, 21] which, however, only focused on first-order protection.

Our Contribution In this work, we introduce a side-channel protected general-purpose CPU based on the RISC-V open instruction-set architecture [23] using the open-source V-scale [13] core. Therefore, we use the findings of domain-oriented masking [8] to modify the open-source V-scale CPU to be resistant against passive physical attacks.

The benefits of our approach compared to custom-made protected hardware implementations are, (1) more flexibility in terms of the selection of algorithms and updatability, (2) faster development of secure systems, (3) hardware-level protection that is transparent for both the running software and the designer, (4) the CPU can be synthesized for arbitrary protection orders by just changing one parameter, (5) a CPU is part of most security-critical systems and therefore requires SCA protection for security-sensitive data processed by the CPU anyway (which are not necessarily cryptographic operations).

2 Efficient Masking in Hardware

Side-channel attacks such as differential power analysis or chip probing attacks typically exploit data dependencies within the observed side-channel information. Therefore, the intuition behind masking is to make security-critical computations independent of the underlying data. Many masking schemes achieve this data independence by representing variables in a so-called shared representation which ensures independence up to a certain protection order d. One of the most popular formal models to investigate the security of masking schemes is the so-called d-probing model introduced by Isha *et al.* [10]. In this probing model, the protection order d equals the number of needles an attacker can utilize in parallel. A circuit that resists probing attacks with up to d needles is said to be d-secure.

The implementation costs for masking schemes, like chip area and randomness requirements, are strongly related to the number of used shares. In the domain-oriented masking scheme (DOM), the primary goal is to minimize the number of required shares to $d + 1$ to reduce the implementation costs. Hereby, a variable x is represented as the sum of $d + 1$ shares in $GF(2)$. Each of these shares is associated with a specific share domain that we denote with capital letters (see Equation 1) with the associated variable in the index. If the sharing itself is referenced we use a bold capital letter as abbreviation for writing each share of x explicitly.

$$x = \underbrace{A_x + B_x + C_x + \dots}_{d+1} = \mathbf{X} \tag{1}$$

The intuition behind DOM is to prevent a protected circuit from combining shares associated with different domains in the same signal path. Therefore, any function that is intended to be performed on the unshared variable x is instead applied on the shares of x following the same principle of domain separation. As a result, any linear function $\mathrm{F}(x)$ is split up in $d + 1$ domain functions as shown in Equation 2 and Figure 1, respectively.

$$\mathrm{F}(x) = \underbrace{\mathrm{F_A} + \mathrm{F_B} + \mathrm{F_C} + \dots}_{d+1} \tag{2}$$

The realization of any non-linear functions—$\mathrm{G}(x, y, \dots)$ in Figure 1—, however, requires the shares to cross the domain borders. A share that is used in a different domain therefore needs to be blinded before it can be safely integrated in the target domain. The blinding is performed by adding a randomly picked share Z to the cross-domain share. To keep the hardware cost low, more complex functions are decomposed into a cascade of simpler linear and non-linear functions.

In the original DOM paper [8], different designs of $GF(2^n)$ multipliers were introduced which serve as the basis for realizing protected logic functions. In particular, two different variants of DOM multipliers were introduced. The first multiplier (DOM-*dep*) does not have any restrictions on the shared inputs regarding the independence of their sharings. As a consequence, it is even allowed to use the same sharing of the same variable x for both inputs. Because the assumption of share independence is trivially given in some cases, a more efficient implementation of the multiplier (DOM-*indep*) was introduced, which requires less randomness and standard cells than the DOM-*dep* realization. The main difference between these two multipliers is that the DOM-*indep* variant does not require one input to be blinded before the multiplication is performed. Instead, only the partial products of the multiplier are remasked before the terms are summed up (for more detailed information please see [8]).

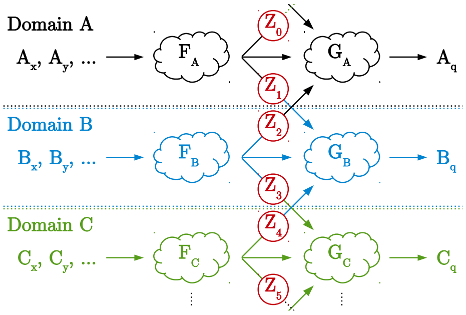

Fig. 1. DOM concept for protection order $d = 2$ and two shared functions

Besides efficiency, one main advantage of DOM is its genericity. This allows any hardware design being implemented according to the DOM scheme to be realized for any protection order d without any redesigning effort. In particular, it leads to hardware designs that use a security parameter d to automatically generate protected circuits for arbitrary protection order without touching the design.

3 Targeted Processor Platform

This work builds upon the V-scale processor that implements the RISC-V instruction-set architecture (ISA), which was originally developed at the University of California, Berkely. RISC-V is a customizable, modular, free and open RISC ISA which suits research perfectly. The architecture is highly flexible, meaning that the register size (32, 64, or 128 bit), their number (16 or 32), the number of privilege levels (1 to 4), and the supported instructions can be chosen according to the desired use case.

The ISA defines the mandatory base integer instruction set (I or E) which contains the most basic memory, arithmetic, logic, and control-flow instructions. Optionally, more complex instructions can be implemented and are defined via various standard extensions. These extensions include, for example, instructions for integer multiplication/division (M), atomic (A) operations, as well as single- (F) and double-precision (D) floating-point computations. The instructions in the base instruction set and the mentioned extensions are all encoded in 32 bits. However, both shorter and longer instructions are supported too. The extension for compressed instructions (C), for example, defines 16-bit instructions, which map to the base instruction set, to increase code density. Furthermore, RISC-V also supports the addition of fully-custom instructions as so called non-standard extensions (X).

The fact that RISC-V, unlike for example the AVR, x86, and the ARM ISA, has no status flags (carry, overflow, zero, ...) is noteworthy too, given that it simplifies the masking efforts. Carry propagation as well as comparisons are performed with dedicated instructions instead.

Like the ISA, also the V-scale processor core has been developed in Berkely. V-scale is a Verilog implementation of the RV32IM instruction set, i.e., it is a RISC-V processor with 32 registers with 32 bit width featuring the base integer instruction set and the

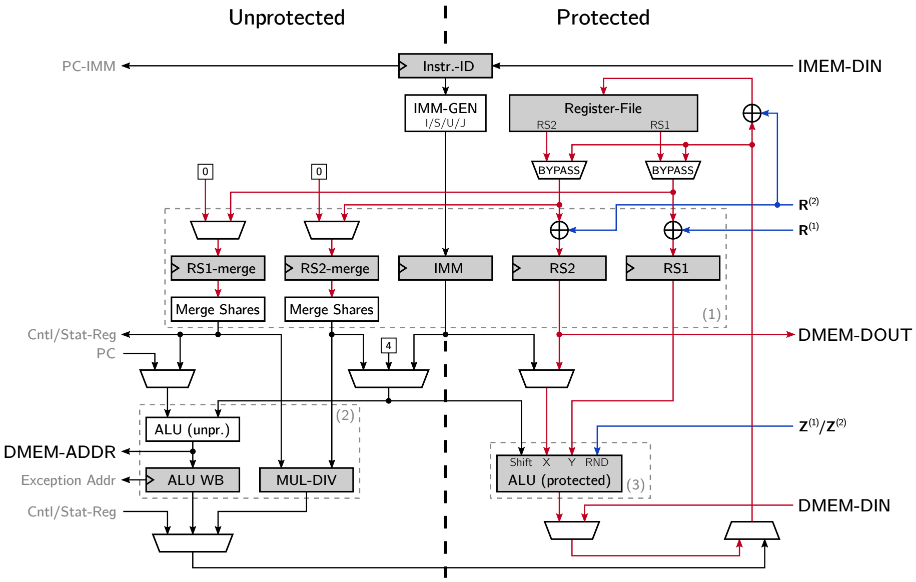

Fig. 2. Overview of the V-scale core. Grey blocks are registers or use a register stage internal. Shared data connections are illustrated in red, unshared in black and the randomness in blue.

integer multiplication extension. The core itself relies on a single-issue in-order 3-stage pipeline comprising a fetch, a combined decode+execute, and a write back stage. Additionally, the data dependencies between consecutive instructions can be resolved using a bypass of the write back stage which permits to maximize the utilization of the core. Communication with memory relies on separated AHB-Lite memory interfaces for instructions and data, permitting to build Harvard and von Neumann architectures.

4 Protected Implementation of V-scale

Our protected implementation of V-scale addresses the problem that data processed by the processor is subject to side-channel attacks. In this work we solely protect the instructions of the base RV32I instruction set as it is the most versatile. Nevertheless, the multiplication/division (M) extension of the original V-scale processor has been kept to maintain compatibility but is still unprotected.

Therefore, the register file, the majority of the ALU and the data memory interface of the V-scale processor have been protected using the DOM scheme. Other parts, like the instruction memory interface and the decoder have been left unprotected. The reason for this split is that in any case the implemented code must be written such that it does not leak information about the processed data over the instruction sequence because different instructions show different power signatures in leakage traces as also mentioned in [7]. Otherwise, even on a fully shared processor, timing attacks would for example be possible.

The resulting processor's architecture is depicted in Fig. 2. One major difference to the original V-scale processor is that the protected core now has four pipeline stages. The additional pipeline stage (see (1) in Fig. 2) splits the previously combined decode+execute stage and is necessary to prevent leakage due to glitches when data shares are merged. This aspect is described in more detail in Section 4.1.

From another perspective, the processor is split into a part that operates on DOM-shared data and a part operating with merged data shares. Accordingly, the ALU itself has been split into a protected and an unprotected part. The unprotected ALU (see (2) in Figure 2) implements multiplication/division, address calculation, and data comparison for conditional jumps. Performing comparisons for conditional jumps in an unprotected way is legitimate as code is not allowed to branch on secure data anyway to avoid timing attacks. More details on the logic to securely merge the different DOM shares and on the unprotected ALU itself can be found in Section 4.2. All the remaining functionality being part of the base instruction set (*e.g.* AND, OR, XOR, ADD, ...) is implemented in the protected ALU in a DOM-protected way. The protected ALU is visualized in Figure 2 at (3) and is thoroughly described in Section 4.3.

4.1 Additional Pipeline Stage

The major change to the unprotected processor are the additional source registers shown in Figure 2 at (1). The main purpose of these buffer registers is to prevent glitches in the merging units connected to *RS1-merge* and *RS2-merge*. These merging units recombine the shares to the original value as shown in Equation 1.

Without the registers *RS1-merge* and *RS2-merge*, (de-)activation of the merging units can result in data dependent glitches. This is illustrated using two basic scenarios. First, the output of the register file switches to sensitive data. This requires the merging units to be disabled by detaching their inputs from the source register. However, if the sensitive data is selected faster than the merging unit is disabled, sensitive data propagates into the merging unit and results in the leakage of sensitive data. Second, the output of the register file switches from sensitive data to data to be merged. This enables the merging unit by switching the multiplexer to the output of the register file. Here, if the multiplexer switches faster than the register file output is selected, the sensitive data from before glitches into the merging units which leaks information. Both scenarios are prevented by the additional buffer registers *RS1-merge* and *RS2-merge*. These effectively decouple the merging units from the register file selector by setting the input to the merging units to zero if not required. To adapt the delay of the protected to the unprotected data path, further buffer registers *RS1* and *RS2* are needed.

Another change to the processor design is the addition of fresh randomness to the processed values before the ALU result is written back to the register file and before the registers *RS1* and *RS2* are used as the operands for the protected ALU. This allows to restore the independence of the sharings after unprotected operations and shifts operations which generate zeros or duplicate the most significant bit, respectively. Furthermore, the addition of fresh randomness is required right before operating on identical operand registers for protected ALU operations.

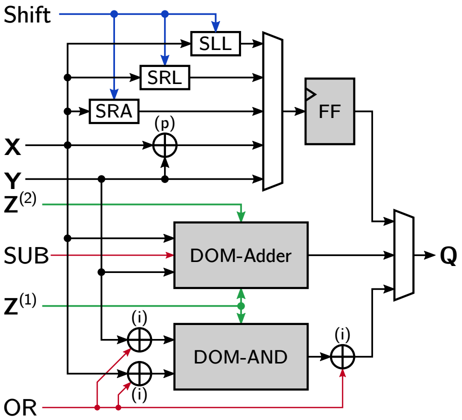

Fig. 3. Protected ALU using a single DOM-AND for AND and OR operation. Shown XOR operations used in different manner: (p)airwise XOR operation of inputs shares (e.g. $A_x + A_y; B_x + B_y; \ldots$); (i)nverting the operand XORing the signal OR with every element of the corresponding first share;

4.2 Unprotected Operations

Figure 2 shows at (2) the modules *MUL-DIV* and *ALU (unpr.)* providing the unprotected operations of our core. These modules operate natively with 32-bit word size and use the merged data as described in Section 4.1. The *MUL-DIV*-module is the unprotected hardware multiplication and division unit from the original V-scale processor design and kept to maintain compatibility.

The unprotected ALU implements different compare operations, i.a., for branch instructions. However, the comparison results can also be written back to a register. While all branch instructions use two source register inputs, instructions storing the comparison result allow to alternatively use an immediate value as the second source. Note that the compare functionality could have been implemented without merging the data, but branching on protected data must anyway be avoided due to possible timing attacks [11]. This design decision should be kept in mind as it makes it necessary to avoid compare operations on protected data.

Furthermore, the unprotected ALU provides an adder to perform address calculations within load and store operations. Note however that the required merging of source register before the actual address computation does not reduce security. As the second operand is constant and determined by a known software implementation, the value of the source register can always be reconstructed, also if a masked adder was used and the shares of the memory address were merged afterwards. Besides, the unprotected adder is also used within two further instructions. First, the adder is used in the jump and link instruction to increment the program counter in the computation of the address of the following instruction. Second, in the add upper immediate to program counter instruction both the program counter and the immediate input are publicly known making a masked adder obsolete.

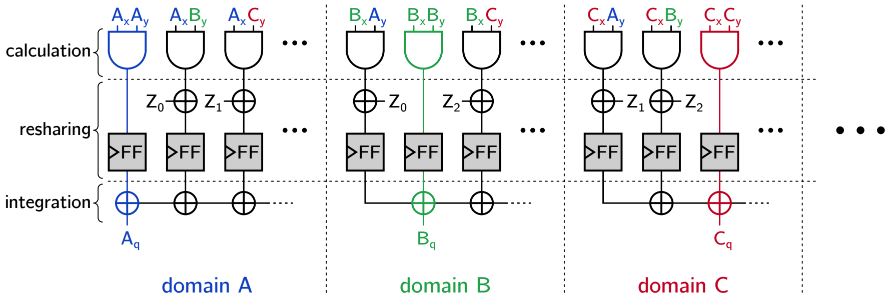

Fig. 4. Overview of the DOM-AND.

4.3 Protected ALU

The protected ALU is shown in Figure 3 which provides the masked functionality for bit-wise logic operations and arithmetic operations. Both input sharings $\mathbf{X}$ and $\mathbf{Y}$ are composed of $d+1$ independent shares (see Equation 3), where d is the protection order of the DOM implementation. For resharing purposes, the protected ALU has two additional inputs $\mathbf{Z}^{(1)}$ and $\mathbf{Z}^{(2)}$ holding the required fresh random shares. The data width of the input shares and the fresh random Z shares is 32 bits each.

$$\mathbf{X} = \underbrace{(A_x, B_x, C_x, \dots)}_{d+1} \qquad \mathbf{Y} = \underbrace{(A_y, B_y, C_y, \dots)}_{d+1} \tag{3}$$

$$\mathbf{Z}^{(1)} = \underbrace{(Z_0^{(1)}, Z_1^{(1)}, Z_2^{(1)}, \dots)}_{d(d+1)/2} \qquad \mathbf{Z}^{(2)} = \underbrace{(Z_0^{(2)}, Z_1^{(2)}, Z_2^{(2)}, \dots)}_{d(d+1)/2} \tag{4}$$

DOM-AND The basis for all implemented non-linear operations is the so-called DOM-*indep* $GF(2)$ multiplier variant (see [8]) which corresponds to a logic *AND* gate with two one-bit inputs. The DOM-*indep AND* gate is illustrated in Figure 4. A basic requirement of the DOM-*indep* multipliers is that the two inputs $\mathbf{X}$ $(A_x, B_x, C_x, \dots)$ and $\mathbf{Y}$ $(A_y, B_y, C_y, \dots)$ are independently shared which is ensured by design of the protected core.

The construction of the *DOM-AND* is generic and can thus be extended to arbitrary protection orders by adding additional shares. For the protection order d, $d+1$ shares per variable are required giving $d+1$ independent share domains. Every domain consists of $d+1$ AND gates and flip-flops which results in a quadratical growth of the chip area accordign to the protection order. The three steps (calculation, resharing, and integration) of the DOM implementation are applied independently for every bit position of the 32-bit shares. Therefore, a 32-bit AND gate consists of 32 *DOM-AND* gates.

In the *calculation* step the terms resulting from the calculation of $\mathbf{X} \times \mathbf{Y}$ (A_xA_y, A_xB_y, A_xC_y, B_xA_y $\dots$) are calculated separately. In the next step (*resharing*) all terms

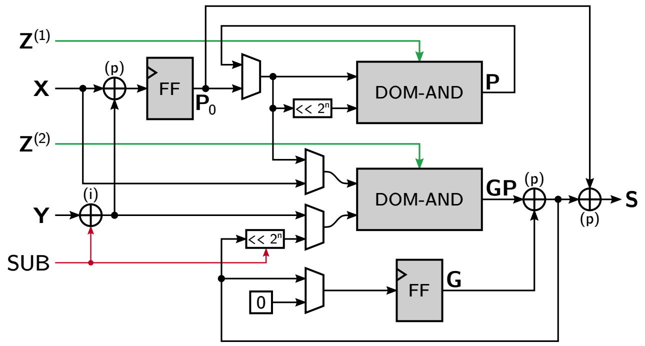

Fig. 5. Masked adder using two DOM-AND. Shown XOR operations used in different manner: (p)airwise XOR operation of inputs shares (e.g. $A_x + A_y; B_x + B_y; \ldots$); (i)nverting the operand by XORing the signal SUB with every element of the first share of $\mathbf{Y}$ (A_y).

that contain shares which are not associated with the respective domain are reshared by using a fresh random Z share. The subsequent register ensure that no early propagation effects occur which could result in glitches that would effect the SCA resistants of the gate. To keep the timing of the masked *AND* synchronous the register is also inserted in inner-domain paths of the domains (e.g., A_xA_y or B_xB_y). The last step reduces the number of shares again to $d+1$ by integrating the freshly masked cross-domain terms into the inner-domain terms and hence generates the output $\mathbf{Q}$ ($A_q, B_q, C_q \ldots$).

DOM-Adder The protected adder is based on a Kogge-Stone similar to the construction of Schneider *et al.* [20]. The adder is a carry lookahead type adder using a tree-like structure separating the addition into propagation and carry generation. Figure 5 shows the secure DOM adder. It is composed of two *DOM-ANDs*, two bit shifts, and multiple XORs. The XOR as well as the shift operations can be performed independently for each share domain and each input. The nonlinear parts of the adder are formed by two *DOM-AND* gates. To make the illustration of the adder in Figure 5 more concise, the three steps for calculating the DOM-AND are only indicated by the respective function (see Figure 4 for more details). The *DOM-AND*'s internal registers together with the $\mathbf{G}$ are used as the working registers for the iterative calculation of the sum. The *DOM-AND*'s internal registers are indicated by $\mathbf{GP}$ which belongs to the carry generation path and $\mathbf{P}$ which belongs to the propagation path.

For the carry generation path the register $\mathbf{G}$ is used to store the previous value of the generation step as it is required in the next iteration. An important requirement of the used *DOM-AND* gate is an independent sharing both inputs. This independence is ensured for both AND gates because the bit position of one operand is always shifted by at least one position. With the same argument the random Z shares in each cycle are applied for both AND gates without violating the independence requirement.

The subtraction operation can easily be performed by calculating the twos-complement of the subtrahend. The subtraction is controlled by the *SUB* input. Therefore, the input signal *SUB* is XORed with every bit of the first share of $\mathbf{Y}$ (A_y). Incrementing the result by one is done by connecting the carry-in of the adder with *SUB* which is active on a subtraction. This is done in the shifter of the generation path by appending the carry bit below the least significant bit of the first share and shifting it into the carry generation path. The following equations uses the $\ll$ operation to indicate a left shift performed independently on every input share supporting only shifts with 2^n where $n \geq 0$. The calculation of the sum is performed in three steps called preprocessing, processing, and postprocessing.

An addition is started with the initial preprocessing step initializing the registers $\mathbf{G}$, $\mathbf{P}_0$ and $\mathbf{GP}$ according to Equation 5.

$$\mathbf{G}_0 = 0 \qquad \mathbf{P}_0 = \mathbf{X} + \mathbf{Y} \qquad \mathbf{GP}_0 = \mathbf{XY} \tag{5}$$

The processing step is performed five times in a row (n =1 $\ldots 5$).The first and last steps are diverging form the normal processing operation. In the first step the input register $\mathbf{P}$ is replaced by $\mathbf{P}_0$. In the last processing step the register update of $\mathbf{P}$ is omitted (see Equations 6-8).

$$\mathbf{G}_n = \mathbf{G}_{n-1} + \mathbf{GP}_{n-1} \qquad n = 1 \ldots N \tag{6}$$

$$\mathbf{P}_n = \mathbf{P}_{n-1}\,(\mathbf{P}_{n-1} \ll 2^{n-1}) \qquad n = 1 \ldots N-1 \tag{7}$$

$$\mathbf{GP}_n = \mathbf{P}_{n-1}(\mathbf{G}_n \ll 2^{n-1}) \qquad n = 2 \ldots N \tag{8}$$

In the final postprocessing step the resulting sum is simply computed by a single XOR operation as shown in Equation 9.

$$\mathbf{S} = \mathbf{P}_0 + (\mathbf{G}_N \ll 1) \tag{9}$$

Resharing of ALU Inputs and Outputs To reduce the required fresh randomness the two resharing values $\mathbf{R}^{(1)}$ and $\mathbf{R}^{(2)}$ in Figure 2 are generated from the random Z shares. Furthermore, the merged value of both $\mathbf{R}$ shares is always zero so that an addition of the shares with a sharing of the register file input or output always result in a resharing without changing the underlying value. For first-order protection the resharing value is generated by duplicating a single random share as shown in Equation 10.

$$\mathbf{R}^{(1)} = (Z_0^{(1)}, Z_0^{(1)}) \qquad \mathbf{R}^{(2)} = (Z_0^{(2)}, Z_0^{(2)}) \tag{10}$$

For other protection orders, the randomness is composed as shown in Equation 11-12.

$$\mathbf{R}^{(1)} = (Z_0^{(1)}, Z_0^{(1)} + Z_1^{(2)}, Z_2^{(1)} + Z_1^{(2)}, Z_2^{(1)} + Z_3^{(2)}, Z_4^{(1)} + Z_3^{(2)}, \ldots) \tag{11}$$

$$\mathbf{R}^{(2)} = (Z_0^{(2)}, Z_1^{(1)} + Z_0^{(2)}, Z_1^{(1)} + Z_2^{(2)}, Z_3^{(1)} + Z_2^{(2)}, Z_3^{(1)} + Z_4^{(2)}, \ldots) \tag{12}$$

To guarantee the independence of both resharing values, the first sharing $\mathbf{R}^{(1)}$ uses the shares of $\mathbf{Z}^{(1)}$ with even and shares of $\mathbf{Z}^{(2)}$ with odd indexes, whereas the second sharing $\mathbf{R}^{(2)}$ uses the remaining shares of $\mathbf{Z}^{(1)}$ and $\mathbf{Z}^{(2)}$. This combination of both Z shares is necessary to prevent adding of two shares which are also used in the *DOM-AND* for the integration step. For example, if the second term of $\mathbf{R}^{(1)}$ uses the same random Z share ($Z_0^{(1)} + Z_1^{(1)}$) it could be used to eliminate two random values in domain A as shown in Figure 4. This reduces the number of signals an attacker has to probe to reveal an unshared intermediate.

Other ALU Operations The remaining operations of the protected ALU (see Figure 3) are the shift operations, the logic operations XOR and OR, and the pass-through path. The shift operations are represented by the blocks SLL, SRL and SRA, which perform logical left or right shift or an arithmetic right shift. The $Shift$ operand uses a separate unshared input for selecting the shift width which is generated outside the module as shown in Figure 2. This is necessary to prevent an unwanted merging of the default used shift operand $\mathbf{Y}$. The shifts are performed independently on every share of $\mathbf{X}$. For the arithmetic right shift the most significant bit of every share is duplicated. The logical shift operations add zeros to the shares. Therefore, the shares must be refreshed which is done before writing back the result into the register file or the buffer registers adding fresh randomness (see Figure 2).

The XOR operation is done in a straight-forward way by adding the input shares of $\mathbf{X}$ and $\mathbf{Y}$ share wise. This leads to a zero result using the same input values. Again the results are reshared using fresh randomness before storing them in the buffer registers *RS1* and *RS2* to guarantee independence of the shares.

The pass-through applies the second input $\mathbf{Y}$ unmodified to the output. To prevent a duplication of the sharing of $\mathbf{Y}$ in different registers, the sharing is again refreshed before writing it to a register.

The OR operation is combined with the AND operation formed by the *DOM-AND* to reduce the logic overhead. This is done by transforming the logical OR into an AND by inverting both inputs and the output. If the OR operation is used, the input *OR* is set which inverts the first share of both input operands as well as the resulting output of the *DOM-AND* by adding to all bits the *OR* signal.

5 Hardware Results

The hardware results are gathered for a Xilinx Kintex-7 FPGA with the Xilinx Vivado Design Suite 2014.3. Therefore, the synthesis was done for the unprotected core as well as for the protected V-scale core with protection orders from 1 up to 4. Figure 6 shows the evolution of required look up tables (LUTs) (left) as well as the required registers (right) for increasing protection order. The overall area seems to grow only linearly with the protection order. The design of the DOM-AND gates which are part of the nonlinear modules of the protected ALU increase quadratically which, however, contribute only marginally to the overall size for lower protection orders. Table 1 shows the area result in numbers. Additional the required randomness is shown which increases quadratically with the protection order. In particular the randomness required for the protected ALU is $32 \times d(d+1)$ bits in each cycle. The last column shows the maximum clock frequency which is higher for the protection orders 1 up to 3 as for the unprotected implementation. This results from the additional pipeline stage of the protected implementation which reduces the critical path but increases the delay on the other hand.

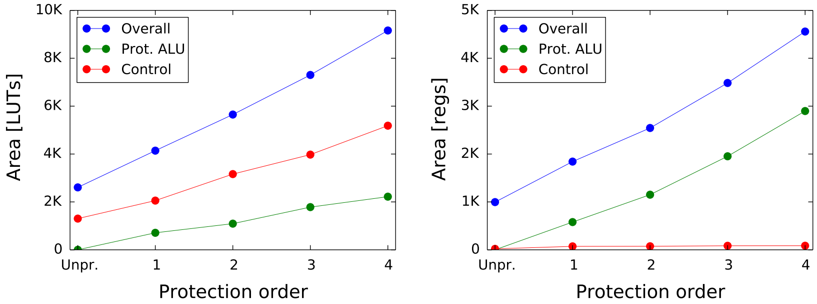

Fig. 6. Required LUT (left) and registers (right) on an FPGA.

Table 1. V-scale core implementation results.

Prot. Order	FPGA Logic		Randomness	Max. Clock
d	*[LUTs]*	*[regs]*	*[Bits]*	*[MHz]*
Unpr.	2,607	996	0	45.6
1	4,143	1,842	64	61.0
2	5,626	2,551	192	59.5
3	7,259	3,484	384	58.3
4	9,244	4,561	640	41.0

6 Side-Channel Evaluation

We have discussed the security of our DOM implementation of the V-scale core in the d-probing model in Section 4. In this section we practically evaluate the resistance of our implementation. To show the first-order resistance of our protected V-scale design, the Welch's t-test is used according to the recommendations of Goodwill et al. [5]. The idea of this test is to collect two sets of traces. One set with completely random inputs and another set with constant inputs—the shares and random input bits for the non-linear are of course still random. The null-hypothesis is that both sets cannot be distinguished from each other, meaning they have identical means.

To make our leakage assessment as reproducible as possible, a SASEBO-GIII [9] based FPGA board, the SAKURA-X is used. The board is especially designed for side-channel evaluation and provides special measurement connectors for measuring the power consumption. The SAKURA-X board consists of a Xilinx Spartan-6 FPGA device working as controller connected to the measurement PC and the Xilinx Kintex-7 FPGA implements the device under attack (DUA)—the protected V-scale core in our case. The leakage traces are collected by a Picoscope 6404C oscilloscope at 312.5 Ms sampling rate for a 8 MHz DUA clock. As the targeted software implementation we implemented the round transformations of an authenticate encryption scheme (ASCON) together with additional code that triggers particular instructions and instruction sequences that were considered critical.

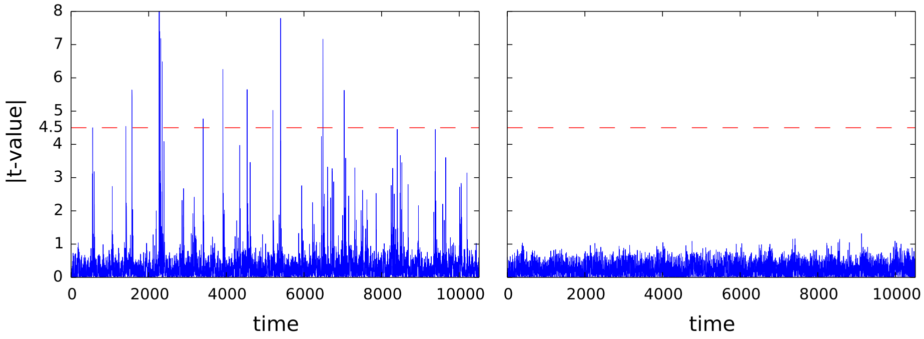

Fig. 7. T-test value of protected ALU operations with inactive random generator with $2M$ traces (left) and active random generator with $100M$ traces (right)

Random Number Generators Turned Off The first-order t-test is performed according to Equation 13 for the two trace sets A and B. In particular, for the two trace sets with random and constant inputs the difference of the mean traces X_A and X_B is calculated. The result is then scaled according to the estimated standard deviations S_A and S_B with respect to the size of the trace sets denoted by N_A and N_B, respectively.

$$t = \frac{X_A - X_B}{\sqrt{\frac{S_A^2}{N_A} + \frac{S_B^2}{N_B}}} \tag{13}$$

If the t-value exceeds the confidence interval of ± 4.5 the null-hypothesis is rejected with confidence greater than 99.99% for large sizes of N.

For validating the functionality of our measurement setup, we first deactivated the used random number generator and performed the t-test which is shown in Figure 7 (left). As expected the t-test showed significant peaks over the ± 4.5 border which indicates first-order side-channel leakage for 2 million traces.

Random Number Generator Turned On We repeated the t-test with the random number generators turned on and collected 100 million traces. Even with 50 times more traces compared to the first t-test the leakage evaluation does not show any significant peaks any more. We thus consider the side-channel countermeasures to work as expected.

7 Conclusions

In this work implemented a side-channel protected V-scale core following the DOM scheme. The implemented core is fully scalable in terms of protection order, and allows to protect informational assets that are processed by the protected V-scale core. As our results show the overhead for the side-channel protection of the core is only a factor of roughly 1.5 for the first-order implementation. We synthesized the processor up to protection order four. Up to this point the size of the core seems to grow only linearly. This results from the fact that the protected ALU (that contains non-linear modules which grow quadratically) is relatively small compared to other parts of the processor.

To show the resistance of our implementation against side-channel analysis attacks, we performed a first-order t-test of our core on a SAKURA-X FPGA evaluation board. The practical evaluation even with 100 million leakage traces does not show any statistical significance. However, a practical evaluation is of course never complete nor a complete argument for the security of an implementation. The formal analysis of our implementation is thus considered as part of future work. Furthermore, the security of our design is in general only given for software that does not introduce any control flow changes based on the asset one tries to protect (timing attacks). However, we do not consider this much of a drawback since constant runtime implementations are a basic requirement of protected software and hardware.

Acknowledgements.

This work has been supported by the Austrian Research Promotion Agency (FFG) under grant number 845589 (SCALAS). The HECTOR project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 644052. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

European Research Council
Established by the European Commission

References

1. S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. Wiener, editor, *Advances in Cryptology CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer Berlin Heidelberg, 1999.
2. C. Chen, M. Farmani, and T. Eisenbarth. A Tale of Two Shares: Why Two-Share Threshold Implementation Seems Worthwhile-and Why it is Not. Cryptology ePrint Archive, Report 2016/434, 2016. `http://eprint.iacr.org/2016/434`.
3. T. D. Cnudde, O. Reparaz, B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen. Masking AES with d+1 Shares in Hardware. Cryptology ePrint Archive, Report 2016/631, 2016. `http://eprint.iacr.org/2016/631`.
4. S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan. Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases. In H. Gilbert, editor, *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 135–156. Springer Berlin Heidelberg, 2010.
5. G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A Testing Methodology for Side-Channel Resistance Validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

6. L. Goubin and J. Patarin. DES and Differential Power Analysis The Duplication Method. In *Cryptographic Hardware and Embedded Systems*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer Berlin Heidelberg, 1999.
7. H. Gross. Sharing is caringon the protection of arithmetic logic units against passive physical attacks. In S. Mangard and P. Schaumont, editors, *Radio Frequency Identification*, volume 9440 of *Lecture Notes in Computer Science*, pages 68–84. Springer International Publishing, 2015.
8. H. Gross, S. Mangard, and T. Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. Cryptology ePrint Archive, Report 2016/486, 2016. `http://eprint.iacr.org/2016/486`.
9. Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *The 1st IEEE Global Conference on Consumer Electronics 2012*, pages 657–660, Oct 2012.
10. 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.
11. P. C. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, pages 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.
12. 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.
13. Y. Lee, A. Ou, and A. Magyar. V-scale. `https://github.com/ucb-bar/vscale`, 2013.
14. 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.
15. S. M. D. Pozo and F.-X. Standaert. A note on the security of threshold implementations with $d + 1$ input shares. Cryptology ePrint Archive, Report 2016/420, 2016. `http://eprint.iacr.org/2016/420`.
16. 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.
17. F. Regazzoni, A. Cevrero, F. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne. A design flow and evaluation framework for dpa-resistant instruction set extensions. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2009.
18. O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating Masking Schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 764–783, 2015.
19. M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer Berlin Heidelberg, 2010.
20. 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.

21. S. Tillich, M. Kirschbaum, and A. Szekely. Implementation and Evaluation of an SCA-Resistant Embedded Processor. In E. Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2011.
22. E. Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.
23. A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.