SPARX - A Side-Channel Protected Processor for ARX-based Cryptography

Florian Bache*, Tobias Schneider†, Amir Moradi†, Tim Güneysu*‡
*University of Bremen, Germany
†Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
‡Cyber Physical Systems, DFKI GmbH, Bremen, Germany
Email: {florian.bache, tim.gueneysu}@uni-bremen.de, {tobias.schneider-a7a, amir.moradi}@rub.de

Abstract— ARX-based cryptographic algorithms are composed of only three elemental operations — addition, rotation and exclusive or — which are mixed to ensure adequate confusion and diffusion properties. While ARX-ciphers can easily be protected against timing attacks, special measures like masking have to be taken in order to prevent power and electromagnetic analysis. In this paper we present a processor architecture for ARX-based cryptography, that intrinsically guarantees first-order SCA resistance of any implemented algorithm. This is achieved by protecting the complete data path using a Boolean masking scheme with three shares.
We evaluate our security claims by mapping an ARX-algorithm to the proposed architecture and using the common leakage detection methodology based on Student's t-test to certify the side-channel resistance of our processor.

I. Introduction

Security plays a major role in many applications of the upcoming Internet of Things (IoT), in particular when security-critical digital devices are also potentially exposed to physical attacks, such as side-channel analysis (SCA) and intentional fault-injection. In this regard, the security system designer is often faced with two contradictory challenges. On the one hand, he is required to build and integrate a robust cryptographic subsystem that is efficient but resistant against any type of (physical) attack. On the other hand, he must create a lifetime-secure but agile system with included migration paths to allow updates of cryptographic components in the field, if necessary. While the first requirement indicates a hardware implementation that combines computational efficiency and physical protection, the second update criterion demands a software-like implementation. In this context Field-Programmable-Gate-Arrays (FPGA) can be one viable solution for some situations, but in many lightweight contexts (e.g., Smart Cards) they are not applicable for several reasons. Hence, for a conventionally combined setting (i.e, using hardware-software co-design) the implementation of a holistic security concept is far from trivial and requires particular care and security expertise from both hardware and software engineers.

A. Contribution

In this work, we present a novel co-processor subsystem that is designed as an Application-Specific Instruction-Set Processor (ASIP) for a specific class of cryptosystems with inherent hardware resistance against side-channel analysis. More precisely, our architecture and instruction set follows principles from the "Threshold Implementation" (TI) concept that is known to provide provable security against power side-channel analysis. As an ASIP it can be loaded with software implementations of different symmetric ARX-based cryptographic primitives, such as stream and block ciphers or hash-functions without the need for adaption of the hardware. We show that a prototype of our design including a software implementation of Speck is not only secure against first-order side-channel analysis and timing attacks but can be realized at moderate costs that are even comparable against pure (protected) hardware implementations. Note that the hardware is designed to completely counter the aforementioned side-channel attack which significantly relaxes the requirements for software engineers to handle complex constraints of physical side-channel security.

B. Related work

While we provide the first ASIP specifically built for ARX-based symmetric cryptosystems, the authors of [1] had previously reported on an approach to protect an existing microcontroller architecture with a power side-channel countermeasure - yet without practical evaluation. The work [2] provides a dedicated accelerator for ARX-based cryptography, which does not include a consideration on physical attacks. Finally, we refer to the proposals for side-channel resistant (hardware) implementations of ARX-based constructions as reported in [3], [4].

II. Preliminaries

A. ARX Algorithms

ARX-based cryptography is denoting a set of symmetric constructions that are purely based on Additions ($\bmod 2^n$), Rotations and XOR (ARX). In [5] the set of ARX operations was proven to be functionally complete over $\mathbb{Z}_{2^n}$. The main advantage of ARX cryptosystems is their compact and fast instantiation on most instruction-set architectures, including resistance against timing attacks by design. Examples for

ARX-based constructions are the block ciphers FEAL, Threefish or Speck; the stream ciphers Salsa20, ChaCha, HC-128 or the hash functions BLAKE and Skein. While there is a wealth of different ARX-constructions, we will focus in this work on the Speck block cipher and the Salsa20 stream cipher as case studies.

Speck: Speck is a recently proposed and lightweight ARX-based block cipher optimized for software implementations presented in [6]. It is specified for block sizes between 32 and 128 bit and uses key sizes between 64 and 256 bit. The operand length for the elemental ARX-operations is half of the block size. In each round, the state is updated using a very simple Feistel-like function, defined as

$$x_{i+1} = ((x_i \ggg \alpha) + y_i) \oplus k_i,$$
$$y_{i+1} = x_{i+1} \oplus (y_i \lll \beta)$$

where (x_i, y_i) is the current state, k_i is the round key, and α and β are constants. The Speck key schedule is very similar to its round function and is computed as

$$l_{i+m-1} = (k_i + l_i \ggg \alpha) \oplus i,$$
$$k_{i+1} = k_i \lll \beta \oplus l_{i+m-1}.$$

where m is the chosen number of key words, $(l_{m-2}, \ldots, l_0, k_0)$ is the key and k_i is the ith round key.

Salsa20: Salsa20 is a lightweight ARX-based stream cipher presented in [7] and a final candidate in the eSTREAM hardware profile. The keystream is generated by calculating a hash of the 256-bit key, a 128-bit constant, a 64-bit nonce and a 64-bit block number using the Salsa20 hash function. The initial 512-bit state is composed by arranging these inputs in a 4-by-4-word matrix containing 32 bit in each entry. After 20 iterations of the round function the result can be used as a keystream. In each round every column is treated separately while each word is updated once, starting with the below-diagonal words. Let x_i be the word in column i that is to be updated, then the update process is defined as:

$$x_i \leftarrow \left(\left(x_{(i-1) \bmod 4} + x_{(i-2) \bmod 4}\right) \lll 7\right) \oplus x_i.$$

After each round the state is transposed. A study analyzing the susceptibility of the phase three eSTREAM candidates towards SCA attests Salsa20 exploitable SPA and DPA vulnerabilities (especially in its rotate function), while assessing the cost of countermeasures as high [8]. A successful (simulated) Correlation Power Analysis (CPA) attack on Salsa20 is presented in [9].

B. Side-Channel Countermeasures

As explained in the introduction, SCA is a severe threat to any cryptographic implementation making the integration of appropriate countermeasures essential. Most of the commonly known approaches are either *masking* or *hiding* schemes. While hiding countermeasures try to reduce the signal-to-noise ratio [10], masking relies on the randomization of intermediate values to make the leakage independent of the secret values up to a certain degree [11]. Masking schemes are based on a sound theoretical foundation and can provide provable security up to a certain order. In particular, the intermediate values are split up into multiple shares and an adversary needs to combine leakages of multiple of these shares to recover the unshared value. This notion of attack is denoted as *higher-order attacks*. A masking scheme of order d provides provable security against all attacks of orders lower than $d + 1$ and it requires a $d+1$-order attack to break it. Given a sufficient level of noise in the measurements, the complexity of an attack increases exponentially in its order, up to the point that it becomes unfeasible in practice.
However, to provide this level of security masking schemes rely on certain assumptions. If these assumptions are violated, the level of security can be severely decreased. Therefore, it is important to implement a masked algorithm with particular care. Notably, glitches, which are temporary faulty states, are a problem for masked hardware circuits and render a straight-forward implementation of a masking scheme insecure [12].

Threshold Implementation: A commonly used concept to achieve secure masking in the presence of glitches is Threshold Implementation (TI). It provides provably first- and higher-order security for arbitrary functions and can be very efficient depending on the masked algorithm. Therefore, it has been (statically) applied to a wide variety of ciphers (e.g., [13]–[15]). In the following, we only briefly introduce the general concept of TI and refer the interested reader to the original publications [16], [17] for a more detailed description of the masking scheme.
In TI, every sensible intermediate state x is split up into n shares with the property $x = \bigoplus_{i=1}^{n} x_i$. This relates to an $n-1$-order Boolean masking scheme which is advantageous for linear functions $L(\cdot)$ as they can be applied to each share separately to compute $L(x)$. However, most cryptographic algorithms contain non-linear functions, e.g., an Sbox or modular addition for ARX schemes. To this end, a TI representation of these non-linear functions has to be constructed in compliance with the three properties *correctness*, *non-completeness*, and *uniformity*. The first property indicates that the shared representation of the non-linear function computes the correct results of the non-linear functions in a shared form. *Non-completeness* (for higher-order TI dth-order *non-completeness*) requires that every component function is independent of at least one input share. One assumption of TI is uniformly shared inputs for each shared function. Since the outputs of these functions are usually inputs to another shared function, it is desirable that the output of such a function is also uniform. This can be achieved by carefully constructing the component functions or with the addition of fresh randomness.

III. Design Considerations and Technical Description

Our system is designed as a generic and intrinsically SCA-protected co-processor platform for ARX ciphers. It supports arbitrary ARX algorithms by updating the program code, without the need for adapting the hardware design. In particular, we merge the concept of the TI countermeasure into our SPARX microarchitecture instead of applying it to a dedicated cipher implementation. For this purpose, we designed an application-specific CPU (ASIP) with a TI-protected ARX-ALU that is provably secure against first-order attacks (using $d = 3$ shares). Certainly, higher-order attacks can also be prevented by increasing d at higher hardware costs. Our system is designed to separate any (SCA-critical) data flow from the control flow.

The fourfold pipelined architecture is based on a RISC approach and incorporates two separate ALUs. The side-channel protected ALU performs all elementary ARX operations on a protected register file with direct access to a source of randomness that is required for the addition operation. We further identified that a dedicated unprotected ALU, which operates on a dedicated register file, is beneficial to increase the overall performance at reasonable costs. Load and store instructions are available for moving data between the RAM and the register files. The high-level structure of SPARX is provided in Fig. 1.

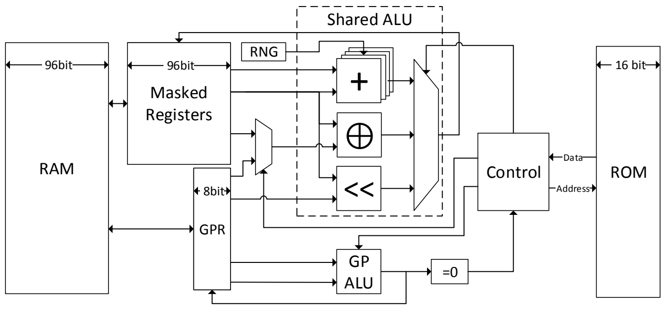


Fig. 1. High-Level Block Diagram of SPARX

A. Protected ARX-Specific ALU

The main ALU operates on triple-shared 32-bit words and is used for all calculations on the sensitive state of the implemented ciphers. It consists of a TI-protected adder, an `xor` and a rotation unit and is connected to a dedicated register file. Because both, the `xor` and rotation operations are linear, ensuring that each share is processed independently is sufficient for the masked implementation. However, addition in $\mathbb{Z}_{2^{32}}$ is non-linear. The construction of a TI-conform shared representation of the addition is non-trivial.

We therefore refer to the discussion of this issue in [18]. Schneider *et al.* proposed two different types of addition circuits for Boolean-masked values which follow the TI principle. For our purpose, we use a similar variant of their proposal based on the ripple-carry adder as it is far more area-efficient for lightweight application than their presented Kogge-Stone adder, while only requiring four bits of fresh randomness per operation. In addition, we found that a slight modification of the original blinded addition circuit can enhance the versatility of our processor. More precisely, in [18] the additions only take the two summands as input and no initial carry. However, it is easily possible to tweak the original design to include this capability without breaching the security assumptions. To this end, we require that the input carry is uniformly shared, which is implicitly given if it is the output carry of a previous addition. As the output carry shares are not independent of their related sum bits they must not be used as joint inputs to a shared function. Our design accounts for that by only using the carry bits for adding multiple 32-bit blocks. With this tweaked adder, our processor can support multi-precision addition of inputs larger than 32 bits (e.g., for Speck128/256, Blake2b or Threefish) without negatively affecting the performance of the single-limb 32-bit addition.

Note that the adder is the slowest element in the overall design, requiring 32 cycles to complete one 32-bit addition. In order to still allow for high throughput the module is provided with a separate clock running at double the frequency of the main clock. This reduces the latency of one addition to 16 cycles. In order to increase the CPU's resource utilization, the adder is connected asynchronously to the rest of the processor using two instructions: one for starting the addition and one for retrieving the result. The retrieve instruction returns the current state of the addition, regardless of whether the operation has finished or not. It is the responsibility of the programmer or compiler to ensure that a retrieve operation is only issued after at least 16 cycles have passed since the addition has started. This task is trivial and can easily be automated because in every cycle exactly one new instruction is fetched.

In order to further increase the throughput of SPARX four parallel addition units were instantiated. This reduces the average number of required clock cycles per addition from $16 + 2 = 18$ to $(16 + 8)/4 = 6$ including the time needed for the add- and retrieve-instructions. Incorporating more than four adders provides diminishing returns in performance and could not be efficiently exploited by most ARX algorithms.

B. Auxiliary General-Purpose ALU

For increased efficiency, SPARX contains an unprotected 8-bit auxiliary ALU that supports general purpose arithmetic and logical operations in particular for control flow operations. It provides single-cycle addition, subtraction, `and`, `or` and `xor` operations, each with either two register operands or one register and one immediate value. Furthermore, the auxiliary ALU can be used to compute counters and flags to control the key-independent program flow, without occupying the main TI-adders. It can also safely calculate round constants and other inputs to the cryptographic algorithm. Sensitive data cannot be leaked by the auxiliary ALU, because there is no connection between the protected register file and the unprotected ALU. Unmasked data can be loaded from and stored in the RAM to enable interaction with the control flow from outside of the CPU. This is useful to dynamically select a cipher algorithm or to generate an "encryption-done" flag for an external main CPU.

C. Data and Control Flow

SPARX is designed based on a standard RISC architecture. The pipeline is composed of four stages, namely *Fetch*, *Decode*, *Execute* and *Writeback*. The design relies on single-cycle read-latency memory and therefore uses neither data- nor instruction-caching. The architecture does not incorporate a stack or `call` and `return` operations to enable function calls, as cryptographic primitives do not benefit from this in general. It does, however, support branches and loops which can reduce the code size for round-based algorithms – like ARX ciphers – significantly. In order to control the program flow, two branching operations are implemented: an unconditional `jump` and a branch-not-zero (`bnz`) instruction. The condition for the `bnz` instruction is generated by comparing the result of the general purpose ALU operations to zero. Masked data cannot be used as input to this process, which ensures that the execution times of all programs running on the proposed architecture are data independent, rendering the design resistant to timing-attacks. In conclusion, there are no operations that stall or flush the pipeline, which results in a throughput of one instruction per cycle.

D. Memory Configuration

Our SPARX processor is based on a Harvard architecture, i.e., it has as separate program and data memory. This enables increased instruction throughput without requiring another memory port and cleanly supports different widths for data and instruction words.
Each instruction word is encoded in 16 bits so a program memory of the same width can easily provide one instruction per cycle. The program memory size is limited to 4096 words. For comparison, our implementations of Speck and Salsa20 need 113 and 310 instruction words respectively.
The data port width is 96 bit to natively support access to 32-bit masked values. Only direct addressing of RAM data is supported. The amount of RAM is not fixed but the instruction-word width limits its size to 512 96-bit words. Besides for buffering data, the RAM is also used as IO interface.
In order to improve the throughput while keeping the number of pipeline stage reasonably low, the processor has access to dedicated registers. However, it is imperative to isolate the masked sensitive data from the unmasked general purpose data in order to prevent information leakage. To this end, two separate register files were implemented in the proposed architecture. The eight general purpose registers are 8 bit wide and can be used for storing rotation offsets, round constants, counters or flags. The second register file is used as a working memory for the sensitive data words SPARX is operating on such as the key, the plaintext and the ciphertext. The number of shared registers had to be considered carefully because each register is 96 bit wide and therefore costly in terms of hardware resources. In order to maximize the utilization of the four adders, up to eight operands / registers are optimal. While some ARX-algorithms could benefit from more than eight registers, the extra hardware cost does not pay off for most scenarios. As an example, Salsa20's throughput could be increased by 12% when 12 registers are available but, according to our benchmarks as described in Section V, at an additional area cost of more than 16%

E. Data Separation

In order to ensure SCA-protection for arbitrary ARX-implementations, information flow from the protected data to the unprotected auxiliary data must be prohibited. Otherwise, information could be leaked although the ARX-primitives have been securely implemented. This is ensured by instantiating separate registers for the critical masked data and the auxiliary non-masked data.
The protected `xor` module operates on either two masked values or one masked and one non-masked input. In both cases masked output values are generated. This feature can significantly improve performance for algorithms relying on round constants: Because the constants are public and do not need to be protected they can be computed using the general purpose ALU instead of the slower, more restricted ARX ALU. The `xor` of the non-masked value B and the shared value $A = A_1 \oplus A_2 \oplus A_3$, which is internally represented as the triple $\tilde{A} = (A_1, A_2, A_3)$, is computed as $\tilde{A} \oplus B = (A_1, A_2, A_3 \oplus B)$. The resistance of SPARX against SCA is not harmed during this operation because only one share of the masked value is modified by the linear and invertible `xor` function.
For the rotation, the value to rotate is always masked while the rotation offset is non-masked. This is not an issue as long as the offset does not depend on secret information. Preventing information flow from the masked to the non-masked data through main memory access is not enforced in hardware. Hence, the compiler must ensure that sensitive data is never loaded into non-masked registers.

IV. Implementation

In this section we describe the technical hardware instantiation of the SPARX processor and a set of ARX-based ciphers in software.

A. Hardware Instantiation of the SPARX Processor

The instruction set architecture of SPARX was specified using the Language for Instruction Set Architecture (LISA) in version 2.0. In order to develop a working prototype of the processor, Synopsis Processor Designer was used to generate HDL code, an assembler and a simulator for the proposed architecture. The rotate-module, the TI-adder and the required RNG-component were hand-coded in VHDL because LISA does not directly support these structures or the generated code was inefficient.

B. SPARX-Compliant Cipher Implementations

Two ciphers were implemented for SPARX as case studies.

Speck: The Speck variant with a 64 bit state and 128 bit key that uses 27 rounds when mapped to the platform. In a straightforward implementation of Speck only two additions, one for the round function and one for the key schedule, are computed per round such that two of SPARX' four TI-adders remain unused. For improved resource utilization, block parallelism with Speck can be used that performs three encryptions and computes the key schedule at once. In our implementation, the rotation of x_i is performed in each round first, after which the additions are started. While the adders are operational, the y_i are bitwise rotated, the round counter is incremented, l_i is stored in the main memory and l_{i+1} is loaded into a register. While load/store-operations are necessary due to SPARX' limited register count, they do not reduce the throughput because the following computations have to wait for the result of the addition either way. After retrieving the addition result, the rest of each round is computed. Our Speck implementation consists of 113 16-bit instruction words and can encrypt three 64-bit words in 1057 cycles.

Salsa20: As the Salsa20 round function operates on four independent columns, its implementation can directly utilize the four addition units of SPARX. Except for the first four words updated in each round, every word depends on the previous computations, which limits the possibilities for re-ordering the operations with respect to latency of our TI-adder. The 16-word state of Salsa20 is too large to fit into the register file but virtually all necessary load/store-operations can be performed while waiting for the adders. Because every other round operates on rows instead of columns, two rounds of Salsa20 were unrolled in order to avoid the overhead associated with transposing the state. The resulting program contains 310 instructions and takes 2937 cycles to compute $512\,\mathrm{bit}$ of key-stream.

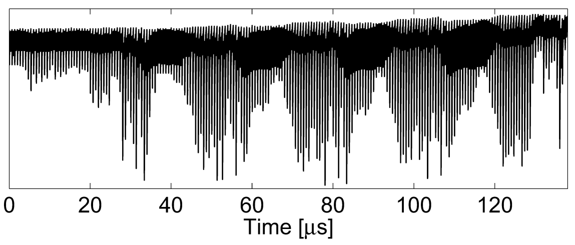
V. Evaluation

In this section we evaluate the SCA protection of the proposed architecture, provide size and performance figures and compare them to previous publications.

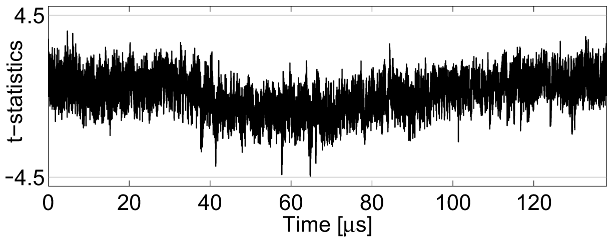
A. Leakage

As stated in the preliminaries, correctly implemented TI-based masking provides provable resistance against power- and EM-attacks. In practice however, ensuring that the TI requirements are actually met in the synthesized design can be difficult, especially for more complex architectures. In order to practically assess SPARX' resistance against SCA attacks, we made use of a SAKURA-G board [19] as an SCA evaluation platform. The design was mapped to a Xilinx Spartan6 XC6SLX75 FPGA, and a 4-round version of our Speck implementation was realized as the target ARX algorithm. Such a reduced-round implementation has been chosen to shorten the power traces in order to accelerate the measurement and evaluation processes, while still utilizing all the processor's components. The power traces have been collected by a digital oscilloscope at a sampling rate of 500 MS/s while the design was clocked at a frequency of 3 MHz thereby obtaining clear traces with minimal level of noise where the power peaks of adjacent clock cycles are not overlapping.

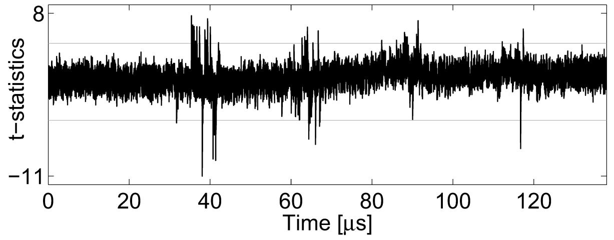
As the evaluation metric, we applied a leakage detection scheme instead of an arbitrary attack vector. The non-specific t-test – known as *fixed versus random* t-test – has been employed to examine the ability of the design to prevent SCA leakages. For more detailed information we refer to the original articles [20], [21]. For such an evaluation, we collected 10 million power traces, one of which is shown in Figure 2a. The t-test results at first and second orders are shown by Figure 2, which confirm the robustness of our construction against first-order attacks. Since first-order TI is the underlying masking scheme, the design as expected exhibits (although small) higher-order leakages, which should in practice complicate the feasibility of higher-order SCA key-recovery attacks.



(a) sample trace



(b) 1st-order t-test



(c) 2nd-order t-test

Fig. 2. A sample power trace, and the result of “fixed versus random” t-test using 10 million traces

B. Performance and Size

The performance of SPARX was evaluated for two different target architectures: a Xilinx Spartan-6 FPGA and the Faraday 180nm ASIC process. The bitstream for the FPGA was generated using Xilinx ISE 14.7 with the optimization goal speed. In order to prevent the mapping tool from using the same LUT to compute functions on bits of independent shares, and thereby violating the TI requirements, the design hierarchy was

TABLE I
RELATIVE AREA COST OF SPARX' MODULES

Module	Size on FPGA (Slices / %)	Size on ASIC (kGE / %)
Adders	471 / 31.0	18.1 / 44.8
Register File	424 / 27.9	9.5 / 23.7
Shifter	107 / 7.0	4.7 / 11.7
XOR	48 / 3.2	0.4 / 1.0
Total	1529	40.4

maintained during synthesis. This resulted in a total number of 1519 used slices and a maximum clock-frequency of 112 MHz. The ASIC netlist was created with Synopsis Design Compiler 2010.12. To ensure that the security assumptions are not violated, ungrouping, flattening and register re-timing were disabled for the synthesis. The total number of cells required by the design amounts to 40358 NAND-gate equivalents. The sizes of SPARX' main modules for the ASIC as well as the FPGA process are provided in Table I. In the FPGA results, slices that are used in more than one module, are counted multiple times.
The throughput of our Speck implementation is $20.3\,\mathrm{Mbit/s}$ on the FPGA and $75.7\,\mathrm{Mbit/s}$ on the ASIC, while the Salsa20 core can encrypt $19.5\,\mathrm{Mbit/s}$ or $69.7\,\mathrm{Mbit/s}$, respectively.

Comparison: While, to our knowledge, the proposed design is the first side-channel resistant, flexible ARX accelerator, several hardware implementations of ARX ciphers have been introduced in the literature. In [4] a 99-slice SCA-resistant Speck implementation for FPGAs with a throughput of $9.7\,\mathrm{Mbit/s}$ is presented. The unprotected Speck implementation proposed in the same contribution achieves a performance of $10.1\,\mathrm{Mbit/s}$ using 42 slices.
The authors of [22] present several unprotected $180\,\mathrm{nm}$-implementations of Salsa20 that are optimized towards either area or throughput. Their balanced iterative design can generate $255\,\mathrm{Mbit/s}$ at $23.4\,\mathrm{kGE}$.
The unprotected high-performance ARX-accelerator CoARX [2] can generate a Salsa20-keystream at a rate of of $1.93\,\mathrm{GBit/s}$ running at $700\,\mathrm{Mhz}$. CoARX achieves this speed when synthesized in a $90\,\mathrm{nm}$ process costing $95\,\mathrm{kGE}$. While the algorithm running on CoARX can be selected via software, the supported algorithms must already be known at (hardware-) design time in contrast to our approach.

VI. CONCLUSION

In this paper we propose a flexible ARX-ASIP that intrinsically protects all implemented algorithms against timing and first-order side-channel attacks. The resistance of our implementation was verified practically by applying a well-established leakage detection scheme. The proposed architecture enables support for multiple ARX algorithms such as block cipher, stream ciphers and hash functions at the same time and permits updating of cryptographic algorithms in the field, while keeping the cost for securely adapting the software to changing requirements minimal.

REFERENCES

[1] H. Gross, "Sharing is caring - on the protection of arithmetic logic units against passive physical attacks," in *RFIDSec*, vol. 9440 of *Lecture Notes in Computer Science*, pp. 68–84, Springer, 2015.

[2] K. Shahzad, A. Khalid, Z. E. Rákossy, G. Paul, and A. Chattopadhyay, "Coarx: a coprocessor for arx-based cryptographic algorithms," in *DAC*, pp. 133:1–133:10, ACM, 2013.

[3] A. Shahverdi, M. Taha, and T. Eisenbarth, "Silent simon: A threshold implementation under 100 slices," in *HOST*, pp. 1–6, IEEE Computer Society, 2015.

[4] C. Chen, M. S. Inci, M. Taha, and T. Eisenbarth, "Spectre: A tiny side-channel resistant speck core for fpgas," *IACR Cryptology ePrint Archive*, vol. 2015, p. 691, 2015.

[5] D. Khovratovich and I. Nikolić, "Rotational Cryptanalysis of ARX," in *Fast Software Encryption*, no. 6147 in Lecture Notes in Computer Science, pp. 333–346, Springer, 2010.

[6] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK families of lightweight block ciphers," *IACR Cryptology ePrint Archive*, vol. 2013, p. 404, 2013.

[7] D. J. Bernstein, "The salsa20 family of stream ciphers," in *The eS-TREAM Finalists*, vol. 4986 of *Lecture Notes in Computer Science*, pp. 84–97, Springer, 2008.

[8] B. Gierlichs, L. Batina, C. Clavier, T. Eisenbarth, A. Gouget, H. Handschuh, T. Kasper, K. Lemke-Rust, S. Mangard, A. Moradi, and E. Oswald, "Susceptibility of eSTREAM Candidates towards Side Channel Analysis," in *SASC - The State Of The Art Of Stream Ciphers*, pp. 123–150, 2008.

[9] B. Mazumdar, S. S. Ali, and O. Sinanoglu, "Power analysis attacks on ARX: an application to salsa20," in *IOLTS*, pp. 40–43, IEEE, 2015.

[10] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *ASIACRYPT*, vol. 7658 of *Lecture Notes in Computer Science*, pp. 740–757, Springer, 2012.

[11] E. Prouff and M. Rivain, "Masking against side-channel attacks: A formal security proof," in *EUROCRYPT*, vol. 7881 of *Lecture Notes in Computer Science*, pp. 142–159, Springer, 2013.

[12] S. Mangard, N. Pramstaller, and E. Oswald, "Successfully attacking masked AES hardware implementations," in *CHES*, vol. 3659 of *Lecture Notes in Computer Science*, pp. 157–171, Springer, 2005.

[13] A. Poschmann, A. Moradi, K. Khoo, C. Lim, H. Wang, and S. Ling, "Side-channel resistant crypto for less than 2, 300 GE," *J. Cryptology*, vol. 24, no. 2, pp. 322–345, 2011.

[14] T. D. Cnudde, B. Bilgin, O. Reparaz, V. Nikov, and S. Nikova, "Higher-order threshold implementation of the AES s-box," in *CARDIS*, vol. 9514 of *Lecture Notes in Computer Science*, pp. 259–272, Springer, 2015.

[15] B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. V. Assche, "Efficient and first-order DPA resistant implementations of keccak," in *CARDIS*, vol. 8419 of *Lecture Notes in Computer Science*, pp. 187–199, Springer, 2013.

[16] B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, N. Tokareva, and V. Vitkup, "Threshold implementations of small s-boxes," *Cryptography and Communications*, vol. 7, no. 1, pp. 3–33, 2015.

[17] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, "Consolidating masking schemes," in *CRYPTO (1)*, vol. 9215 of *Lecture Notes in Computer Science*, pp. 764–783, Springer, 2015.

[18] T. Schneider, A. Moradi, and T. Güneysu, "Arithmetic addition over boolean masking - towards first- and second-order resistance in hardware," in *ACNS*, vol. 9092 of *Lecture Notes in Computer Science*, pp. 559–578, Springer, 2015.

[19] "Side-channel AttacK User Reference Architecture." http://satoh.cs.uec.ac.jp/SAKURA/index.html.

[20] 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.

[21] T. Schneider and A. Moradi, "Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations," in *CHES*, vol. 9293 of *Lecture Notes in Computer Science*, pp. 495–513, Springer, 2015.

[22] J. Yan and H. M. Heys, "Hardware Implementation of the Salsa20 and Phelix Stream Ciphers," in *Canadian Conference on Electrical and Computer Engineering*, pp. 1125–1128, 2007.