2023 Design, Automation & Test in Europe Conference (DATE 2023) – Best Paper Award Candidate

Efficient Software Masking of AES through Instruction Set Extensions

Songqiao Cui and Josep Balasch
e-Media Research Lab, STADIUS, KU Leuven, Leuven, Belgium
Email: {songqiao.cui, josep.balasch}@kuleuven.be

***Abstract*—Masking is a well-studied countermeasure to protect software implementations against side-channel attacks. For the case of AES, incorporating masking often requires to implement internal transformations using finite field arithmetic. This results in significant performance overheads, mostly due to finite field multiplications, which are even worsened when no lookup tables are used. In this work, we extend a RISC-V core with custom instructions to accelerate AES finite field arithmetic. With a 3.3% area increase, we measure 7.2x and 5.4x speed up over software-only implementations of first-order Boolean Masking and Inner Product Masking, respectively. We also investigate vectorized instructions capable of exploiting the intra-block and inter-block parallelism in the implementation. Our implementations avoid the use of lookup tables, run in constant time, and show no evidence of first-order leakage when evaluated on an FPGA.**

***Index Terms*—Side-channel attacks, masking, instruction set extension, RISC-V**

I. Introduction

Side-channel attacks are a popular method to extract sensitive data from cryptographic devices. They involve an adversary capable of collecting and analyzing side-channel leakage information, the most notable example being the instantaneous power consumption [1]. A myriad of literature has appeared in the last decades on how to prevent side-channel attacks. Among the proposed techniques, masking [2] is a very popular countermeasure due to its soundness and its suitability to protect block cipher implementations.

In its simplest form, masking consists in splitting every sensitive variable s in the implementation into two shares (s_1,s_2) which are then used to perform computations. The encoding function $s = f(s_1, s_2)$ determines the type of masking. The most classical is Boolean masking, where $f = s_1 \oplus s_2$, but other constructions exist that leverage on more complex encodings. A masking scheme provides a set of computational procedures to operate in the masked domain. Besides functional equivalence, such procedures must also ensure all sensitive intermediates values remain masked. When the computational procedures of a scheme support an arbitrary number of shares n, e.g. when $f = (s_1, \ldots, s_n)$, one talks about higher-order masking.

Masking can be implemented either at gate or at algorithm level. The former method is popular for hardware implementations and has a generic nature. It often involves replacing atomic gates in an (unprotected) circuit with functional-equivalent gadgets that operate in the masked domain. The latter method is more common for software implementations, and is linked towards specific cryptographic algorithms. It typically exploits specifities of the algorithm to obtain more efficient implementations.

In this work we focus on software implementations of the Advanced Encryption Standard (AES), where both the encoding and the masked functions are defined over the field GF(2^8). Addition in GF(2^8) is equivalent to bitwise exclusive-or, which is natively available in a processor's Instruction Set Architecture (ISA) and therefore is very efficient. Multiplication in GF(2^8) is however more involved, and demands dedicated software routines which bring significant overheads. This is particularly the case when they are implemented without neither lookup tables nor conditional statements, which is necessary to avoid timing side-channel leakages. As multiplications in GF(2^8) represent the lowest level of arithmetic operations in masking of the AES, any overhead here has a major impact to the overall efficiency of the implementation. To give an example, a first-order Inner Product masking [3] of the AES requires roughly 19,000 multiplications in GF(2^8) Hence it is clear that any improvement in the execution of this low-level function results in significant gains.

Motivated by this observation, in this work we set out to design, implement and validate a small Instruction Set Extension (ISE) tailored to accelerate field arithmetic in GF(2^8). By focusing on the lowest level arithmetic layer, we ensure that most algorithmic level masking schemes applied to the AES benefit from it. For the purposes of demonstration and evaluation, we opt to prototype our work and our side-channel resistant implementations on a compact RISC-V core.

A. Related Works

A few works have tackled the issue of bringing masking-based side-channel countermeasures to RISC-V cores. For instance, De Mulder *et al.* [4] propose a RISC-V architecture where internal components are inherently hardended against attacks. These include both the register bank and the ALU, as well as transfers to/from memory. The work by Kiaei and Schaumont [5] introduces a separated and secure datapath that executes a set of instructions protected against side-channels attacks. The solutions provided in these works integrate the countermeasures directly into the processor architecture, making them somewhat independent of the software running on them. Our objective is instead to accelerate certain operations

The authors would like to thank B. De Cock and Q. Vansina for their initial explorations in the context of this work. This research has been funded in part by KU Leuven Startfinanciering STG/20/047. Songqiao Cui is funded by a PhD grant strategic basic research from the FWO Research Foundation Flanders (grant number 1SF8223N).

978-3-9819263-7-8/DATE23/© 2023 EDAA

with minimal additional execution units, which are used from the software side.

Following a similar reasoning, the work by Gao *et al.* [6] proposes an ISE for masking tailored to software developers. In particular, a total of 22 instructions are proposed that support Boolean masking operations as well as mask conversion. While similar in spirit, our work differs from [6] in that we aim for a small subset of instructions for arithmetic in GF(2^8), which can be leveraged by different masking schemes. Examples include higher-order masking schemes based on Boolean masking [7], [8], or Inner Product masking [3]. Other solutions such as Affine masking [9] can also benefit from it to pre-compute lookup tables. Note that bitsliced implementations such as [10] operate at bit level (AND, XOR) and therefore cannot benefit directly from such an ISA which assumes variables defined in GF(2^8). To account for this, we explore in the work vectorized implementations which, similar to bitslicing scenarios, assume multiple data blocks are processed in parallel.

B. Contributions

Our contributions are as follows:

1) We create dedicated hardware instructions to accelerate multiplication in the AES field GF(2^8), the major bottleneck for masked software AES implementations.
2) We incorporate singular and vectorized instructions as extension to the popular `VexRiscv` core and analyze their area costs by synthesizing them on an FPGA.
3) We benchmark assembly implementations of unprotected and protected AES implementations using Boolean and Inner Product Masking schemes, both for the singular and vectorized cores.
4) We evaluate the resistance of our assembly implementations using non-specific leakage assessment to validate their security against first-order attacks

C. Paper organization

In Sect. II we review the masking schemes implemented in this work and their application to AES in software. In Sect. III we introduce our proposed instruction set extensions, describe the modifications performed to the original RISC-V core and associated tooling, and quantify the area costs of the extended cores. In Sect. IV we benchmark our hardware-accelerated implementations w.r.t. the baseline case and provide the results of the side-channel leakage assessment when synthesizing the core on an FPGA. In Sect. V we conclude the paper.

II. Background and Related Works

In this section we present in more detail the masking schemes considered in this work and their application to protect AES in software. In particular, we focus on the higher-order schemes proposed by Rivain and Prouff [7] (based on Boolean masking) and by Balasch *et al.* [3] (based on Inner Product masking). For each of them, we summarize the cost of their masked operations.

A. Boolean Masking and Inner Product Masking

Boolean masking [2] is one of the most studied and implemented masking schemes. Its encoding function is based on the exclusive-or operator. For an arbitrary number of shares n, variables are split as: $x = s_1 \oplus \ldots \oplus s_n$.

In [7] Rivain and Prouff show how to leverage on this encoding function to secure operations in any finite field. With the goal of protecting AES, they provide a set of algorithms to mask the necessary GF(2^8) operations at any order n. The cost of each masked algorithm in terms of basic finite field operations (addition $\oplus$ and multiplication $\otimes$) is listed in Table I (upper rows). Linear operations (addition, squaring) are typically efficient in the masked domain, as they can be applied individually per share without need of recombining them. This is also the case for the refreshing algorithm, which is used to remove data dependencies between operations by adding fresh randomness. In contrast, multiplication has quadratic complexity due to the calculation of product terms of all shares. Note that the table omits randomness costs, as the focus of our work is on computational costs. For details on the exact algorithms we refer the reader to [7].

In [11] Balasch *et al.* present a masking scheme that uses the inner product of two vectors (denoted $\boldsymbol{L}$ and $\boldsymbol{R}$) as encoding function. For an arbitrary number of shares n, variables are split as: $s = r_1 \oplus (l_2 \otimes r_2) \oplus \ldots \oplus (l_n \otimes r_n)$.

For efficiency reasons the first element of $\boldsymbol{L}$ (which is assumed to be public) is set to one, i.e. $l_1 = 1$. The main advantage of Inner Product Masking is that its encoding function brings improved security properties and better resistance against transition leakages [3]. The price to pay is an increase in complexity of its masked constructs. These are summarized in Table I (bottom rows). Besides addition, the remaining masked operations require more field operations, in particular costly multiplications. Note that some of these multiplications (also in Boolean masking) are often tabulated in implementations, i.e. to perform squarings or multiplications by constants. We however consider consider them as normal multiplications as we aim for implementations without lookup tables. For details on the exact algorithms we refer the reader to [3].

TABLE I
Cost of masked algorithms in terms of finite field operations and number of shares n.

Operation	$\mathtt{GF}_{ADD}$ ($\oplus$)	$\mathtt{GF}_{MUL}$ ($\otimes$)
$\mathtt{BM}_{ADD}$	n	-
$\mathtt{BM}_{MUL}$	$2n^2$ - 2n	n^2
$\mathtt{BM}_{SQ}$	-	n
$\mathtt{BM}_{REF}$	2n-2	-
$\mathtt{IP}_{ADD}$	n	-
$\mathtt{IP}_{MUL}$	$2n^2$	$2n^2$+n-1
$\mathtt{IP}_{SQ}$	-	2n
$\mathtt{IP}_{REF}$	2n	n-1

B. Application to the AES

The AES is a block cipher that operates on a 128-bit state, internally arranged as a 4 x 4 array of bytes in GF(2^8). In

this work we focus on AES-128 encryption, where the AES states goes iteratively through a series of linear and non-linear transformations. We incorporate masking at algorithm level by adapting each transformation to operate on masked data.

Table II summarizes the number of masked operations required by each AES transformation. Non-linear transformations are in general very efficient. This is particularly the case of `AddRoundKey`, which only involves 4 x 4 element additions, and `ShiftRows`, which simply reorders array elements without additional calculations. For `MixColumns` we use the `xtime` equations, which require four doubling operations (i.e. multiplications by 2). The most costly transformation is `SubBytes`, which internally calculates a field inverse followed by an affine transformation. For the field inverse, we leverage on the approach originally proposed in [7] and compute it with the power function x^{254}, which can be evaluated with a monomial that requires 4 multiplications and 7 squarings. The refreshing of 2 intermediate computations is needed to prevent leakages due to dependent variables. For the affine transformation, we evaluate it with an equation over GF(2^8), which requires 8 additions, 7 squarings, 7 multiplications with a constant and 2 refreshings. Following the reasoning given before, we consider multiplications by a constant as normal multiplications.

TABLE II
NUMBER OF MASKED OPERATIONS PER AES TRANSFORMATION.

AES Transformation	Masked Operation			
	ADD	MUL	SQ	REF
`AddRoundKey`	16	-	-	-
`SubBytes` (inverse)	-	4	7	2
`SubBytes` (affine)	8	7	7	2
`ShiftRows`	-	-	-	-
`MixColumns`	15	4	-	-

III. INSTRUCTION SET EXTENSION

In this section, we elaborate on the instruction set extensions that we have developed to accelerate multiplication in GF(2^8). We discuss how these have been integrated to the baseline RISC-V processor used in this work (`VexRiscv`) and the adaptations made to the RISC-V tool-chain and the SoC wrapper (LiteX) necessary to obtain a functional and synthesized design for FPGA environments.

A. Hardware Multiplier in GF(2^8)

Multiplication in GF(2^8) takes two variables as input and generates one variable as output. All input/output variables are elements in GF(2^8) and consequently encoded as bytes. A multiplication in GF(2^8) involves two steps: the multiplication of the two inputs followed by a reduction by an irreducible polynomial. In context of AES, the irreducible polynomial is given by $x^8 + x^4 + x^3 + x^1 + x + 1$.

Implementing this operation in hardware can be done with basic logic gates, namely AND and XOR gates. On the first step, 64 AND gates and 49 XOR gates are used to calculate the 16-bit result of the multiplication. On the second step, 29 XOR gates are used to perform the reduction back to an element in GF(2^8). The reduction is implicitly hardcoded in our multiplier, as we focus on implementations of the AES and therefore consider the irreducible polynomial as fixed. If one were to design a generic multiplier in GF(2^8), a different approach would be required to perform the reduction. We opt to implement the multiplier as a fully combinational circuit, such that it can be efficiently integrated into the execution stage of the processor. By doing so, we incorporate it as an atomic operation that executes with the same latency as native instructions such as AND or XOR. In other words, we essentially make the cost of a field multiplication equivalent to a field addition. Naturally, the insertion of this field multiplier brings certain area overheads for the core, which are discussed later in Sect. III-D.

B. Proposed Instructions in RISC-V Context

The support for encoding space for opcode extensions, regulated in the RISC-V Instruction Set Manual, enables to develop custom instructions while maintaining compatibility with the baseline ISA. Our proposed instructions are summarized in Table III. We propose two sets of instructions: *singular* (first row) and *vectorized* (second row). As the name indicates, instructions in each set differ in the amount of multiplications in GF(2^8) that can be performed in parallel. This enables different implementations as will be detailed in Section IV, exploiting inter-block and intra-block parallelism options in the implementation.

Focusing first on the *singular* case, two instructions are proposed: `GFMul` and `GFMulImm`. The instruction `GFMul` is the most generic: it reads two operands from source registers, and outputs the result to a destination register. The instruction is of type R-format, and performs a single multiplication in GF(2^8). Since registers are 32 bits, only the lowest byte is used by this operation. The instruction `GFMulImm` is a specific I-format instruction that employs an immediate rather than a second source register. As mentioned in Sect. II, several masking operations involve multiplying variables with a constant. Having this instruction allows to save one instruction compared to using `GFMul`, as the constant does not need to be loaded into a register. Immediates in RISC-V are 12 bits long, which is enough to hold the 8 bit value required by the instruction. Since it reuses the same multiplier circuit of `GFMul`, this instruction comes at virtually no cost other than some extra decoding logic.

The second set of instructions, for the *vectorized* case, are simply extensions of the singular operations. Instead of a single multiplication, they perform four parallel multiplications treating the 32-bit registers as four virtual 8-bit registers. Thus, `GFMulVec` is a vectorized version of `GFMul` operating on registers, and `GFMulVecImm` is an extension of `GFMulImm`. The latter still employs an 8-bit immediate, which is replicated to each of the four multiplier instances. Note that both singular and vectorized instructions are assigned with the same instruction pattern. The vectorized instructions are essentially reduced to singular ones if 24 MSBs are set to 0.

In order to use and validate our custom instructions, we need to perform some modifications to the RISC-V GNU toolchain.

TABLE III
OVERVIEW OF PROPOSED RISC-V INSTRUCTIONS.

Instruction ABI	GFMul GFMulVec	GFMulImm GFMulVecImm	← **singular** ← **vectorized**
Format	R	I	
Opcode	0001011	0001011	
funct3	000	111	
funct7	0000011	-	
Operand	d,s,t	d,s,j	
MATCH	0x600000b	0x700b	
MASK	0xfe00707f	0x707f	

This is required so that the compiler can recognize the instruction ABI and translate it to the right machine code. Table III accommodates a summary of the added instructions, with ABI name, format, pattern, operands, *MATCH* and *MASK*, in lines with RISC-V instruction naming conventions and formatting.

C. Integration in VexRiscv

VexRiscv is an open-source RISC-V implementation written in SpinalHDL, a Scala-based Hardware Description Language (HDL). A nice feature of VexRiscv is that it can be highly customized by means of the so-called *plugin* system. Architectural aspects such as pipeline stages, supported RV32I instruction sets, MUL/DIV extensions, memory caches, etc. can be defined prior to generating the final HDL. The integration of customized instruction extensions in VexRiscv is supported by means of the same plugin system.

We start from a baseline VexRiscv core configuration, useful for area comparison and benchmarking. The baseline core features a 5-stage pipeline and support for RV32IM instruction sets. We opt to disable both the instruction cache and data cache to obtain a compact baseline core. Note that our implementations are aimed to run in constant time and to avoid the use of lookup tables, hence caches could be enabled without affecting the security evaluation in Sect. IV.

We implement our instructions by means of a customized plugin (*GFMulPlugin*), which extends the plugin interface within VexRiscv. It notifies the decoding stage about the new instruction pattern and the control unit about register bypassing and operand usage. The hardware multiplier in GF(2^8) is directly integrated in the execution stage, using the operands as required by the instruction, i.e. with or without immediate, and pushing the results to next pipeline stages. We generate two independent designs, one with support for singular instructions and one with support for vectorized instructions, in order to compare their area overheads.

D. FPGA Validation

LiteX [12] is a convenient and efficient framework, written in Migen (a Python based HDL) that allows to create System-on-Chip (SoC) designs that act as interface between a softcore and the outside system. The resulting SoC designs can be ported to different FPGA boards by adapting configuration and constraint files. Interesting for our purposes is that LiteX has already support for VexRiscv cores. We leverage on this to create an SoC extended with a serial communication interface, so that we can both communicate with the core as well as load custom firmware directly to the FPGA.

We incorporate our VexRiscv core flavours (baseline, singular and vectorized) in the LiteX platform, and synthesize the SoC design for Xilinx Artix7 FPGA using Vivado 2020.1. Table IV summarizes the implementation results of all cores. Our VexRiscv with singular extensions requires around 60 additional LookUp Tables (LUT), which corresponds to a 3.3% area increase. The core for vectorized extensions requires 202 extra LUTs, hence an area increase of 7.5%. Note that the increase from singular to vectorized is not an exact three-fold. This is explained by the fact only combined LUTs are contained in the report, e.g. a 6-input LUTs in Artix7 is counted whether one or two of the internal 5-input LUTs are used, plus potential optimizations performed by Vivado. Although the relative area increase may seem large, we stress that our baseline core is intentionally compact and would be reduced if using a more complex core. For both our designs, the flip-flop count increases only by 2 due to the extra instructions.

TABLE IV
OVERVIEW OF RESOURCE UTILIZATION.

Core	LUT	Flip-Flops
VesRiscv (baseline)	1814	1072
VesRiscv (singular)	1874	1074
VexRiscv (vectorized)	2016	1074

IV. EVALUATION

In this section, we present our masked implementations of the AES. We focus on first-order Boolean masking and Inner Product masking, and benchmark the gains obtained with the different cores that we developed (baseline, singular, and vectorized). We also provide a Test Vector Leakage Assessment (TVLA) evaluation [13] on FPGA to confirm our accelerated implementations do not show evidence of leakage.

A. Implementations and Performance Results

In order to have better control of the processor's resources and to force the usage of our customized instructions, we have developed all our implementations in RISC-V assembly. An important remark here is that, in our current instantiation of the VexRiscv core, instructions take a minimum of 2 clock cycles to execute. This is due to a limitation of the Wishbone bus implemented by the LiteX platform when composing the SoC, which limits the processor throughput to one instruction per two cycles and one memory access per six cycles. As a consequence the *absolute* cycle counts are quite bloated with respect to related works, even though our assembly code has been heavily optimized. We note that this is an architectural effect of our SoC generation and has limited impact on our main contributions. Indeed, the key point in our performance analysis is to determine the *relative* cycle count gains across the different implementations, which is not strongly affected by this effect.

The timing results of our implementations are summarized in Table V. The first implementation (AES) corresponds to a software-only reference implementation in the

baseline core. It is a classical AES implementation, unprotected against side-channel attacks which uses lookup tables to perform the SubBytes transformation and the doublings in MixColumns. The overhead caused by masking can be measured with respect to this implementation, which takes roughly 22 k clock cycles.

TABLE V
PERFORMANCE OF IMPLEMENTATIONS (ROUNDED).

Implementation	Clock Cycles	VexRiscv Core
AES	22 372	**baseline**
AES-BM$_{SW}$	1 291 317	
AES-IP$_{SW}$	2 165 339	
AES-BM$_{HW}$	177 645	**singular**
AES-IP$_{HW}$	399 507	
AES-BM$_{HW,INTRA}$	54 809	**vectorized**
AES-IP$_{HW,INTRA}$	124 317	
AES-BM$_{HW,INTER}$	209 079	
AES-IP$_{HW,INTER}$	422 694	

The second set of implementations correspond to the masked versions of the AES with Boolean Masking (AES-BM$_{SW}$) and IP Masking (AES-IP$_{SW}$) in the baseline core, i.e. without instruction acceleration. Both implementations use $n = 2$ shares and perform multiplication in GF(2^8) in software. The routine uses log-alog tables without conditional statements (i.e. constant time) and takes 98 clock cycles to execute. Compared to the reference AES, the masked implementations incur 57x and 96x times overhead, respectively for BM and IP.

The implementations on the singular core (AES-BM$_{HW}$ and AES-IP$_{HW}$) are virtually the same as the previous ones. The only difference is that multiplication in GF(2^8) is accelerated with the dedicated instructions. This change results in 7.2x and 5.4x times speed up for BM and IP, respectively, visualizing the usefulness of our hardware-accelerated instructions.

The implementations on the vectorized core exploit different parallelism options depending on the number of blocks (i.e. number of encryptions) that need to be processed by the application. We first explore *intra-block* parallelism, where certain operations within a single AES encryption are performed in parallel. This case is useful when a single block is encrypted and latency is important. The SubBytes transformation, representing the bottleneck of the implementation, is performed on four AES state elements in parallel. Additionally, the AES state is rearranged to facilitate the calculations of ShiftRows and MixColumns. The resulting implementations (AES-BM$_{HW,INTRA}$ and AES-IP$_{HW,INTRA}$) achieve an additional 3.24x and 3.2x times speed up with respect to the singular core.

Lastly, we also explore *inter-block* parallelism where multiple (independent) AES encryptions are performed together. This case is similar to exploited in bitsliced implementations, e.g. [10], where multiple data blocks are encrypted and throughput is important. In this case, we observe slightly larger gains of 3.4x and 3.76x respectively for BM and IP. Note that for this case, the numbers reported in Table V correspond to four blocks.

In both parallelization options, the gains are very close to the optimal 4x speed up that could theoretically be obtained. The reason why gains are slightly lower is simply due to the overhead caused by extra operations (e.g. pack/unpack) as well as the limited parallelization options of memory accesses.

B. Side-Channel Leakage Evaluation

We have used the CW305 Artix7 FPGA Target by NewAE Technology as platform to evaluate the side-channel security of our implementations. For each VexRiscv core, we have ported and synthesized the LiteX SoC for the Artix7 FPGA in the CW305 and enabled serial communication to interface with an external PC. Since our core does not have an internal random number generator, we have implemented an internal cryptographic pseudo-random number generator that provides (before each encryption) the randomness required by the implementation. The seed is provided by the PC at the beginning of each measurement campaign. We have configured the VexRiscv cores to operate at a clock frequency of 12.5 MHz. This value is selected to avoid overlaps between the power patterns of consecutive clock cycles. To measure the power consumption, we have used an external Picoscope PS6404 with a sampling interval of 3.2 ns and captured the first three rounds of the algorithm.

For the analysis of measurements, we have used the TVLA described in [13]. For each measurement campaign, we have collected two sets of measurements while performing AES encryption on the evaluation board: one where the (unmasked) plaintext is fixed, and one where the plaintext is randomly generated. The (unmasked) key is the same for both measurement sets and it is refreshed before each algorithm execution. The TVLA test calculates the Welch's (two-tailed) t-test for each sample in the measurement sets, and returns a vector of *t-scores* as result. As common in the literature, a threshold value of the t-score is set at $\pm$ 4.5 to determine if the implementation exhibits leakage in the first-order moment.

Fig. 1 shows the outcome of the t-test evaluation of our masked implementations in the singular core: AES-BM$_{HW}$ and AES-IP$_{HW}$. In these experiments, masking is effectively disabled by setting all random values to zero. It is a sanity check with the purpose of confirming the experimental setup is sound. Consequently we expect it to indicate the presence of leakage as confirmed in the plot, with many samples crossing the ± 4.5 boundary. We have collected 10,000 measurements for these experiments.

Fig. 2 shows the outcome of the t-test evaluation when randomness is enabled. We have collected 1 M measurements for these experiments. Note that the upper plot, corresponding to the implementation of Boolean masking with $n = 2$ shares, exhibits leakage. As explained in [3], this effect is caused due to transition based leakages in the implementation and/or processor architecture which can undermine the security of the implementation. This effect is stronger in Boolean masking, whose encoding function is based on the exclusive-or operator. Increasing the number of shares to $n = 3$, as shown in the middle plot, effectively yields a secure first-order implementation regardless of the existence of transition based leakages [14].

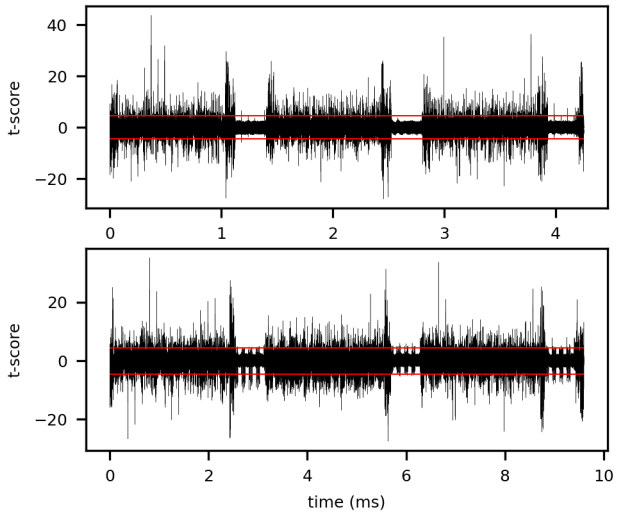

Fig. 1. T-test results for AES-BM$_{HW}$ (top) and AES-IP$_{HW}$ (bottom) with RNG deactivated. The red lines mark the ±4.5 threshold.

Note however that the execution time increases considerably. For Inner Product masking, as shown in the bottom plot, there is no evidence of leakage when using $n = 2$. The experimental results demonstrate the soundness of our implementations and confirm the insertion of the GF(2^8) multiplier has no adverse effects on their side-channel security.

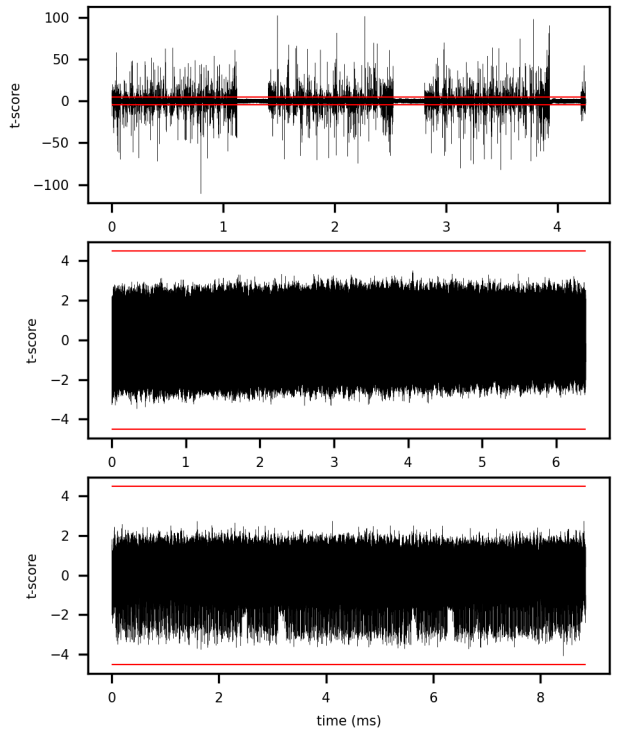

Fig. 2. T-test results for AES-BM$_{HW}$ (top, $n = 2$), AES-BM$_{HW}$ (top, $n = 3$), and AES-IP$_{HW}$ (bottom, $n = 2$) with RNG activated. The red lines mark the ±4.5 threshold.

V. Conclusion

In this work we have proposed a compact ISE for RISC-V to support field multiplications in GF(2^8). By accelerating the lowest level of arithmetic computation, our ISE can be leveraged on by different masking schemes with application to the AES. We have proposed two possible sets of extensions (singular vs. vectorized), aiming at different scenarios and with a different area vs. execution time tradeoff. We have developed assembly implementations based on Boolean Masking and Inner Product Masking, benchmarked their overhead, and evaluated their side-channel security on an FPGA, demonstrating that the added hardware does not compromise their security.

References

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99*, ser. LNCS, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 388–397.

[2] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Advances in Cryptology - CRYPTO '99*, ser. LNCS, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 398–412.

[3] J. Balasch, S. Faust, B. Gierlichs, C. Paglialonga, and F. Standaert, "Consolidating inner product masking," in *Advances in Cryptology - ASIACRYPT 2017*, ser. LNCS, T. Takagi and T. Peyrin, Eds., vol. 10624. Springer, 2017, pp. 724–754.

[4] E. D. Mulder, S. Gummalla, and M. Hutter, "Protecting RISC-V against side-channel attacks," in *Design Automation Conference 2019 - DAC 2019*. ACM, 2019, p. 45.

[5] P. Kiaei and P. Schaumont, "Domain-oriented masked instruction set architecture for RISC-V," *IACR Cryptol. ePrint Arch.*, p. 465, 2020.

[6] S. Gao, J. Großschädl, B. Marshall, D. Page, T. H. Pham, and F. Regazzoni, "An instruction set extension to support software-based masking," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 4, pp. 283–325, 2021.

[7] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *Cryptographic Hardware and Embedded Systems - CHES 2010*, ser. LNCS, S. Mangard and F. Standaert, Eds., vol. 6225. Springer, 2010, pp. 413–427.

[8] J. Coron, A. Greuet, and R. Zeitoun, "Side-channel masking with pseudo-random generator," in *Advances in Cryptology - EUROCRYPT 2020*, ser. LNCS, A. Canteaut and Y. Ishai, Eds., vol. 12107. Springer, 2020, pp. 342–375.

[9] G. Fumaroli, A. Martinelli, E. Prouff, and M. Rivain, "Affine masking against higher-order side channel analysis," in *Selected Areas in Cryptography - SAC 2010*, ser. LNCS, A. Biryukov, G. Gong, and D. R. Stinson, Eds., vol. 6544. Springer, 2010, pp. 262–280.

[10] P. Schwabe and K. Stoffelen, "All the AES You Need on Cortex-M3 and M4," in *Selected Areas in Cryptography - SAC 2016*, ser. LNCS, R. Avanzi and H. M. Heys, Eds., vol. 10532. Springer, 2016, pp. 180–194.

[11] J. Balasch, S. Faust, B. Gierlichs, and I. Verbauwhede, "Theory and practice of a leakage resilient masking scheme," in *Advances in Cryptology - ASIACRYPT 2012*, ser. LNCS, X. Wang and K. Sako, Eds., vol. 7658. Springer, 2012, pp. 758–775.

[12] F. Kermarrec, S. Bourdeauducq, J. L. Lann, and H. Badier, "LiteX: an open-source SoC builder and library based on Migen Python DSL," *CoRR*, vol. abs/2005.02506, 2020.

[13] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side channel resistance validation," http://csrc.nist.gov/news events/non-invasive-attack-testing-workshop/ 08 Goodwill.pdf, 2011.

[14] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert, "On the cost of lazy engineering for masked software implementations," in *Smart Card Research and Advanced Applications - CARDIS 2014*, ser. LNCS, M. Joye and A. Moradi, Eds., vol. 8968. Springer, 2014, pp. 64–81.