Improving the Efficiency of Cryptography Algorithms on Resource-Constrained Embedded Systems via RISC-V Instruction Set Extensions

Carlos Gabriel de Araujo Gewehr*†, Fernando Gehm Moraes*
*School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
†EnSilica – Porto Alegre, Brazil
carlos.gewehr@edu.pucrs.br, fernando.moraes@pucrs.br

Abstract*—This work presents the design and evaluation of RISC-V implementations of AES-128, AES-256, SHA-256, and SHA-512, both with and without specialized instructions from the *Zkne* and *Zknh* ISA extensions. In our evaluation, we use the Ibex implementation of the RISC-V ISA, a simple low-area 2-stage pipeline design, and the TinyCrypt library, a collection of low-overhead C implementations of widely employed cryptography algorithms. Several criteria relevant to low-complexity embedded systems are measured and compared, such as area costs for the hardware side; stack usage and code density for the software side; illustrating the trade-offs emerging from using specialized RISC-V instructions in the aforementioned algorithms. Clock cycle count gains of 42.57x, 44.81x, 1.45x and 1.74x were observed, as well as 4.16x, 4.16x, 1.58x and 1.63x gains in memory usage efficiency and 27.81x, 28.91x, 1.45x and 1.79x gains in energy efficiency, with an overhead of 10% in die area cost. The extended TinyCrypt library with hardware accelerated implementations and extended Ibex processor RTL are available open-source at *https://github.com/cggewehr/RISCV-crypto*.

***Index Terms*—RISC-V; Instruction Set Extensions; Embedded Systems; Cryptography; Security; Hardware Acceleration.**

I. Introduction

With the increasing deployment of low-complexity embedded systems such as IoT sensor nodes, battery-powered and energy-harvesting devices, secure communication is a core requirement. Such devices often do not have the necessary resources for locally computing data processing demands, delegating intensive work to the cloud or network edge. Due to this emerging computing paradigm, a significant amount of data must be securely transmitted to and from several network endpoints. This is not an easy task to accomplish, seeing as the cryptographic algorithms that enable the confidentiality and integrity of the data being transmitted, such as AES [1] and SHA-256 [2], are associated with non-trivial overheads in application performance and memory usage, which are critical concerns in the context of resource-constrained devices.

Hardware acceleration can significantly enhance performance, memory, and energy efficiency in crucial operations of an algorithm. One way to achieve hardware acceleration is by implementing specialized instructions in a general-purpose processor. Compared to loosely-coupled hardware accelerators, these instructions provide benefits for low-complexity embedded systems: (i) significant resource sharing between general and specialized components, including register file and main memory interface; (ii) trivial data transfer within the processor's register file, avoiding costly loads/stores to peripherals or shared memory; (iii) no added complexity to the interconnect and interrupt controller.

The RISC-V ISA [3] is increasingly gaining popularity due to enabling customization to suit specific application requirements via Instruction Set Extensions (ISEs). Relevant to the context of security in low complexity embedded systems are the *Zkne* and *Zknh* [4] ISEs, defining specialized instructions for AES encryption; and SHA-256 and SHA-512 hashing, respectively. As an open ISA, no licensing fee is required for distributing implementations of a RISC-V processor. In this manner, several open-source RISC-V cores have been made publicly available at no cost [5, 6, 7]. Between such implementations, of special interest is Ibex [5]. Ibex is a small 2-stage pipeline design aimed at minimizing complexity, which has been used in other relevant works in literature [8, 9, 10, 11]. Additionally, several security-minded features, such as ECC-protected register file and dummy instruction insertion, can be easily obtained by simply setting appropriate parameters, making Ibex a prime choice for resource-constrained embedded systems where security is a core requirement.

The requirements for low resource usage in such contexts are also relevant in its software components. The TinyCrypt [12] library is designed specifically for embedded systems with limited resources. Due to its design philosophy of specifically targeting such devices, it has been chosen as a fair software context for this work, providing real-world applicability.

This work aims to present the implementation and evaluation of AES, SHA-256, and SHA-512, both with and without specialized instructions from the *Zkne* and *Zknh* ISA extensions. The original contributions of this work include:

- an open-source RTL implementation of the Ibex RISC-V processor with *Zkne* and *Zknh* extensions;
- an open-source implementation of hardware-accelerated AES, SHA-256 and SHA-512 in highly optimized assembly using RISC-V *Zkne* and *Zknh* extensions, transparent with existing deployments of the TinyCrypt library;
- a deep analysis of the trade-offs surrounding the implementation of the *Zkne* and *Zknh* extensions, evaluating gains in performance, energy and memory efficiency, and die area costs.

This work was financed in part by CNPq (grant 309605/2020-2) and FAPERGS (grant 21/2551-0002047-4).

II. Related Work

Previous efforts in open-source RISC-V cores have not sufficiently addressed the need for low-overhead security in resource-constrained embedded systems. The PULP project [7] provides several RISC-V cores and loosely coupled dedicated cryptography cores, easily integrated via an automated system generation tool. This does not provide an optimal solution for low-energy cryptography workloads due to the long logical distance between the main processor and hardware accelerators. Software productivity is also a negative factor, seeing as custom drivers must be developed for each accelerator.

The OpenTitan project [11] uses a similar approach with loosely-coupled accelerators, but uses a higher-complexity version of the Ibex core with *Zb** bit-manipulation extensions for its main processor. It does not intend to provide a platform for use in IoT applications, but in the same manner, as PULP [7], the use of loosely-coupled accelerators does not satisfy the constraints of the low-complexity context being explored.

The SCARV project [6] implements a 5-stage RISC-V core with cryptography ISEs and a software library using these ISEs to achieve hardware acceleration. The SCARV core is of significantly higher complexity than the simple 2-stage Ibex core used in this work. It does not provide a fair comparison, nor is it an adequate processor choice for the low-resource context being explored. Additionally, the software library is custom-made, again leading to challenges in software productivity in integrating it into other projects.

Zgheib et al. [8] propose a tightly-coupled AES accelerator integrated into an Ibex core. A 40% increase in FPGA slice usage when integrating the AES accelerator into the Ibex processor, improving performance by 46 times compared to a software implementation from OpenSSL. The authors also report their solution consumes 44.9 times less energy than OpenSSL AES. Note that this AES implementation is not ideal considering an embedded context. Seeing as the `SubBytes` operation is done in an online manner instead of via a lookup table, avoiding cache timing side-channel attacks. In a low-complexity embedded context, such attacks are not applicable as there is no cache to leak timing information. A table-based `SubBytes` such as the TinyCrypt implementation used in the present work is clearly both faster and more compact, but we offer no quantitative evidence to this claim.

Marshall et al. [13] present a comparative analysis of several RISC-V ISE proposals, where the Saarinen et al. [14] work is demonstrated to be the superior extension to the 32-bit base ISA. The ISE proposed by Saarinen et al. [14] was later standardized and ratified as the *Zkne* and *Zknd* extensions in the RISC-V scalar cryptography as the ISE for AES encryption and decryption, respectively. Synthesis results are provided in terms of NAND2 equivalent gates. In the context of a SCARV [6] core, the Saarinen et al. [14] ISE shows a 3% area increase, accelerating both encryption and decryption. Saarinen et al. [14] report a 5% increase in LUT usage in an FPGA-based evaluation considering a lower complexity single-cycle RISC-V core. No power or energy results are provided in [13, 14].

In the software side of the evaluation in [13], [14] shows 3.5x and 1.65x performance gains for AES-128 encryption and encryption key expansion, respectively. The software comparisons consider a higher-performance T-Table based implementation, trading-off memory usage for performance. It is significantly faster than the TinyCrypt implementation used in this work, at a much higher memory footprint.

Further software evaluations concerning the entire RISC-V cryptography ISE (not only *Zkne* and *Zknd* as discussed so far) are reported in [15, 16]. Nişancı [15] presents an analysis of program and static data memory usage for accelerated and non-accelerated implementations of cryptography algorithms, prioritizing execution speed in their implementations. The Authors show a reduction of 83% in program memory usage in AES-128, but do not seem to consider a reduction in static data memory usage in their accelerated implementation, nor stack usage in their evaluation. It is unclear if their AES implementations use the stack at all. In this work, we show the reduction in static data and stack memory usage to be of significant importance to the gains provided by an accelerated AES implementation. For SHA-256 and SHA-512, clock cycle count gains of 43% and 40%, as well as a reduction in program memory usage by 33% and 32% are reported, respectively. Similar results are reported in the earlier [16] work, evaluated in the context of the previously discussed SCARV [6] core.

Despite reporting performance in terms of clock cycles instead of instructions retired, the authors of [15] do not sufficiently describe their hardware platform, simply stating their processor core to be ”a 32-bit 5-stage pipelined RISC-V processor that only supports base integer instructions”. Crucial facts concerning the complexity of the RISC-V core used in their evaluation such as operand forwarding, branch prediction, and multiplier and shifter topology are omitted. This makes it difficult to compare results, seeing as the complexity of their RISC-V core cannot be correlated to the Ibex core we use.

In summary, the core contributions of the works reviewed in this Section are presented and compared in Table I. Note that all works reviewed present ASIC implementation results using open-source synthesis tools and generic cell libraries. Our work uses a commercial synthesis tool and cell library, providing results with a better correlation to physical implementation than previous evaluations. Deeper insights into the RISC-V cryptography ISEs, especially *Zknh*, in low-complexity embedded contexts are scarce in the literature.

TABLE I
Related Work Comparison

Reference → Feature ↓	[8]	[13]	[14]	[15]	[16]	Our Work
Open-source RTL		X	X		X	X
Area cost evaluation	X	X	X	X	X	X
Energy cost evaluation	X					X
Commercial EDA tools						X
Manufacturable cell library						X
Open-source software		X	X		X	X
SHA-2 extensions evaluation				X	X	X
Established software library						X
RISC-V performance counters						X

III. RISC-V Scalar Cryptography Extensions

A. *Zkne extension for AES encryption*

AES [1] is a block cipher standard, widely used for confidentially transmitting information. AES iterates through its internal state in 4 fundamental operations in the following order: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`[1]. Note that the `SubBytes` operation involves a non-trivial inversion in GF(2^8), along with the affine transformation which follows. The entirety of the compound operation in `SubBytes` is commonly known as a Substitution Box, or SBOX. An optimization technique employed in the TinyCrypt library used in this work is to pre-compute each SBOX value and store them in a Look-Up Table (LUT). The SBOX LUT comprises 256 bytes, and is equivalent in size to 64 RV32I instructions. Assuming that online computing `SubBytes` requires more than 64 RV32I instructions, this can be seen as an optimization for performance and code size.

Seeing as the `ShiftRows` and `MixColumns` operations perform shifts and multiplications by constants, this reasoning can be taken one step further by pre-computing not the result of `SubBytes`, but of `MixColumns`. In contrast to an S-Table implementation, which used pre-computed values only for `SubBytes`, this approach is commonly known as a T-Table implementation, named following T_0, T_1, T_2, T_3 as defined below, where $d_{i,j}$, $a_{i,j}$ and S denote the output of `MixColumns`, the round's initial state and the SBOX LUT at row *i* and column *j*, respectively:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S[a_0] \\ S[a_{1,j-1}] \\ S[a_{2,j-2}] \\ S[a_{3,j-3}] \end{bmatrix} = T_0[a_0] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}]$$

$$T_0[x] = \begin{bmatrix} S[x]\cdot 2 \\ S[x] \\ S[x] \\ S[x]\cdot 3 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x]\cdot 3 \\ S[x]\cdot 2 \\ S[x] \\ S[x] \end{bmatrix} \quad T_2[x] = \begin{bmatrix} S[x] \\ S[x]\cdot 3 \\ S[x]\cdot 2 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x]\cdot 3 \\ S[x]\cdot 2 \end{bmatrix}$$

Note that each T-Table is a 1-to-4 byte mapping, instead of the previous 1-to-1 byte mapping using S-Tables. This increases memory usage from 256 to 4K bytes, but simplifies an entire AES round to 4 LUT lookups and 8 bitwise XOR operations. The performance to code size trade-off can be balanced by pre-computing only 1 T-Table and computing the remaining table entries online via rotations.

The *Zkne* extension defines two new specialized instructions: `aes32esmi` and `aes32esi`. `aes32esmi` can be interpreted as computing T-Table entries online in hardware, as well as performing `AddRoundKey` on the relevant byte i.e. an entire byte of the next AES state can be computed entirely with a single instruction. This provides the performance benefits from a T-Table implementation without the need for storing LUTs in memory, leading to gains in performance and code size, as well as energy efficiency gains via the reduction in memory accesses. Finally, `aes32esi` performs only a single SBOX lookup and XOR, used in computing round keys and the last encryption round.

[1]We kindly assume that the reader is already familiar with the inner workings of AES. For those who are inexperienced in this area, we recommend [17] for an introduction to AES.

B. *Zknh extension for SHA-256 and SHA-512 hashing*

The SHA-2 [2] standard defines the SHA-256 and SHA-512 hash algorithms, reducing a message of arbitrary size into 256 or 512 bits, respectively. A message M is padded and split into [256, 512] bit sized message blocks m. An initial h_0 value is set, and the following message blocks are computed with $h_i = f(h_{i-1}, m_i)$. The output is taken as the final h value.

For SHA-256, 4 instructions are defined: `sha256sum0`, `sha256sum1`, `sha256sig0` and `sha256sig1`. These instructions directly implement the Σ_0^{256}, Σ_1^{256}, σ_0^{256} and σ_1^{256} functions defined in the SHA-2 standard [2], used in the computation of $f(h_{i-1}, m_i)$, shown below. The *ROTR*$_n$ and *SHR*$_n$ functions denote rotation right shift and logical right shift by *n* bits, respectively.

$$\Sigma_0^{256}(x) = ROTR_2(x) \oplus ROTR_{13}(x) \oplus ROTR_{22}(x)$$
$$\Sigma_1^{256}(x) = ROTR_6(x) \oplus ROTR_{11}(x) \oplus ROTR_{25}(x)$$
$$\sigma_0^{256}(x) = ROTR_7(x) \oplus ROTR_{18}(x) \oplus SHR_3(x)$$
$$\sigma_1^{256}(x) = ROTR_{17}(x) \oplus ROTR_{19}(x) \oplus SHR_{10}(x)$$

The availability of *Zknh* instructions allows for an implementation that is not only faster, but more compact and energy-efficient. Implementing these 4 functions in plain RV32I is very costly due to the usage of rotation shifts, for which no RV32I instruction can directly compute. Computing a rotation shift by *n* bits in RV32I instructions requires a logical shift by *n* bits in the forward direction; a logical shift by $32 - n$ bits in the reverse direction; and OR-ing both these intermediary shifts together. This requires the use of additional temporary registers, which may lead to the compiler storing intermediary values in the stack due to the lack of registers available to the register allocator.

Similar functions are defined for SHA-512:

$$\Sigma_0^{512}(x) = ROTR_{28}(x) \oplus ROTR_{34}(x) \oplus ROTR_{39}(x)$$
$$\Sigma_1^{512}(x) = ROTR_{14}(x) \oplus ROTR_{18}(x) \oplus ROTR_{41}(x)$$
$$\sigma_0^{512}(x) = ROTR_1(x) \oplus ROTR_8(x) \oplus SHR_7(x)$$
$$\sigma_1^{512}(x) = ROTR_{19}(x) \oplus ROTR_{61}(x) \oplus SHR_6(x))$$

Since SHA-512 variables are defined to be 64 bits in size, such functions cannot be directly implemented as a single instruction yielding a 32-bit result, as is the case for SHA-256. Note that the temporary registers issue is aggravated due to the increased variable size. Six instructions are defined to accelerate SHA-512 sigma functions: `sha512sig0h`, `sha512sig0l`, `sha512sig1h`, `sha512sig1l`, `sha512sum0r` and `sha512sum1r`. With *Zknh* instructions σ_0^{512} and σ_1^{512} are computed with 2 different instructions, for the high and low parts of the result separately, while Σ_0^{512} and Σ_1^{512} are computed via 2 executions of the same instruction, but once with inverted operands, implying a rotation by 32 bits.

IV. Extending Ibex with Cryptography ISEs

A. Zkne extension for AES encryption

The *Zkne* instructions are implemented in the AES Unit module, shown in Fig. 1. Our implementation is derived from the functional unit from the original ISE proposal in [14], which features additional logic for AES decryption and the SM4 cipher. The AES unit is entirely combinational, executing both the `aes32esmi` and `aes32esi` instructions in a single clock cycle. Both instructions are of an extended R type, which expects as inputs 2 registers to be read from the register file, plus a 2-bit immediate Byte Select (*BS*). AND gates are used to gate inputs, minimizing switching activity in the AES unit.

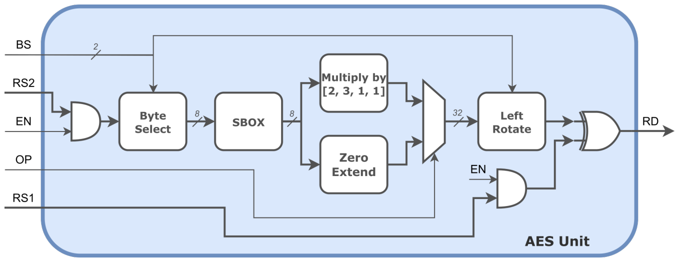

Fig. 1. AES Unit

Note how each sub-module within the AES Unit relates to the AES fundamental operations: `SubBytes` is performed one byte at a time in the SBOX sub-module, on the byte determined by the *BS* immediate value read from register *RS2*; `ShiftRows` is computed by selecting the appropriate byte from the *RS2* register, which contains a column of the current AES state, via the *BS* immediate; *MixColumns* is computed similarly to the T-Table method shown in Section III-A, where the result of `SubBytes` is multiplied by the [2, 3, 1, 1] vector and left-rotating it by *BS* bytes; `AddRoundKey` is computed by XOR-ing the result of *MixColumns* to the relevant word of the round key given in *RS1*. *RS1* should be equal to *RD*, such that this register accumulates the XOR between the 4 T-Table entries and the round key i.e. a column of the next round state.

B. Zknh extension for SHA-256 and SHA-512 hashing

Zknh instructions are also implemented in their own functional unit, shown in Fig. 2. The SHA-2 unit is again entirely combinational, with all *Zknh* instructions executing in a single cycle. The sigma functions discussed in Section III-B lend themselves to an efficient hardware implementation, seeing as shifts and rotations by static amounts have zero cost. The SHA-2 unit does not have a well-defined internal structure like the AES unit, and can be seen simply as a sea of XOR gates.

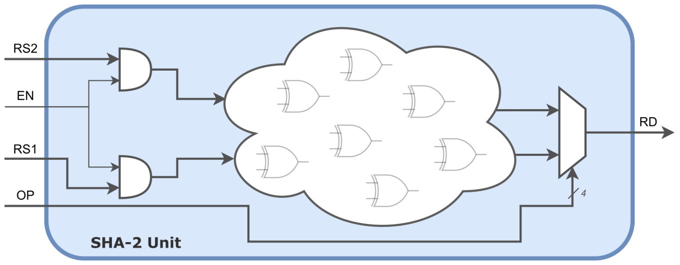

Fig. 2. SHA-2 Unit

C. Functional units integration

The integration of the AES and SHA-2 functional units in the Ibex processor's ID/EX pipeline stage can be seen in Fig. 3. New elements added to the processor are colored in blue, existing elements modified to accommodate the new functional units are colored in green, while unmodified elements are colored in yellow.

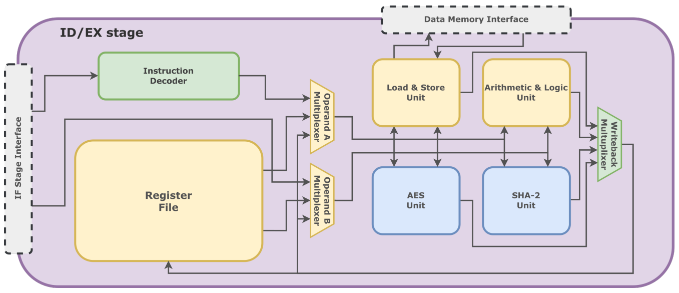

Fig. 3. Ibex ID/EX pipeline stage datapath

Fig. 3 also illustrates the benefits of implementing hardware acceleration as ISEs over loosely-coupled accelerators in embedded contexts. Note that only small changes in the instruction decoding and writeback logic are required to integrate functional units into the Ibex processor, while resource sharing with existing processor elements is extensively explored, namely in the register file and load/store unit.

D. Hardware-software interface

The compiler must be informed that a specific microarchitectural feature is available to compile software with instructions from standardized extensions. GCC achieves this using the `march` flag. For example, to use instructions from the *Zknh* extension, GCC should be called with the `march` flag set to *rv32imc_zknh*. The assembler can then generate binaries that match the encoding specified in the extension specification, from which the RTL implementation can decode and execute the specific operation to be accelerated.

The implementation of the algorithms themselves is accomplished through inline assembly inside TinyCrypt functions, conditionally compiled as either the original TinyCrypt C implementation or optimized assembly implementations. Function declarations remain the same, such that APIs are not modified when using hardware-accelerated assembly. This makes hardware acceleration transparent to the user, not leading to additional effort in software integration and not breaking existing code using the TinyCrypt library.

V. Experimental Evaluation

We compare the implementation and execution of the AES-128, AES-256, SHA-256 and SHA-512 algorithms in a baseline unmodified Ibex core and the extended Ibex core presented in Fig. 3. For both cases, we set the data independent timing flag in the Ibex control register. We consider the *SecureIbex* parameterization in both cases, excluding the too costly ECC RAM and lockstep features. We also consider the latch-based register file, 3-cycle multiplier and no writeback stage parameterizations.

It should be noted that the TinyCrypt library does not have ready-made implementations of AES-256 and SHA-512. We obtain a comparable AES-256 implementation by setting appropriate parameters at compile-time and small changes to the key schedule logic of the existing AES-128 implementation. For SHA-512, we consider an implementation from pull request #42. For AES only the encryption operation (using the forward cipher) is evaluated, seeing as the most common block cipher modes of operation such as CTR do not require the inverse cipher for the decryption operation, using the forward cipher for both encryption and decryption. We use GCC version 12.2.0 for generating binaries. For all cases, we consider as inputs the example test vectors from [1, 2].

For AES, both the Key Schedule and Encryption of a single block are evaluated. For SHA-256 and SHA-512, two scenarios are considered: Full, considering the incremental API calls with non-accelerated elements such as padding and IV initialization; Compression, in which only the hardware accelerated compression function $f(h_{i-1}, m_i)$ is evaluated.

A. Performance Evaluation

Performance counters for each function call are presented in Table II. White columns show values for the baseline TinyCrypt implementations while grey columns show values for the optimized assembly with ISEs implementations. Performance gains in clock cycles are most notable in AES encryption, showing gains of 42.57x and 44.81x, while in the Key Schedule, observed gains were 3.82x and 4.55x in AES-128 and AES-256, respectively. Our results in terms of total instructions retired very closely match AES-128 results from [13], due to similar implementation techniques being used, originally proposed in [14]. AES-256 was not evaluated in [13, 14].

SHA-2 algorithms present different results than [15], which reports speed-ups in terms of clock cycles of 1.76x and 1.65x, whereas our work shows speed-ups of 1.45x and 1.74x for the compression functions of SHA-256 and SHA-512, respectively. We theorize this to be attributed to the use of loop unrolling and higher complexity hardware, but due to the omission of source code for both hardware and software in [15], we cannot elaborate any further on why our results do not match theirs. Furthermore, comparing clock cycle counts between the Full and Compression scenarios using *Zknh* instructions, non-trivial API overheads of 10% and 15% are seen. It is unclear if this is considered in [15].

B. Memory Usage Evaluation

We use the GCC `fstack-usage` flag to obtain static stack usage for each function and the GNU *nm* tool to obtain size information for functions and static data, presented in Fig. 4. Baseline TinyCrypt values are denoted with diagonal stripes while optimized assembly values are denoted with dots. The AES key schedule as well as plaintext data are not included. The AES SBOX LUT is only included in the Encryption cost, but is also used in the Key Schedule.

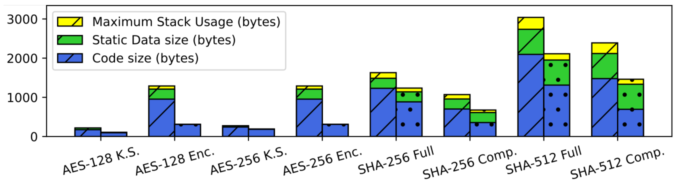

Fig. 4. Memory Usage for AES-128, AES-256, SHA-256 and SHA-512

Fig. 4 shows code size to be the most impactful factor in memory usage, with significant gains in code size most notably in AES (both) Encryption (3.08x), SHA-256 Compression (1.97x) and SHA-512 Compression (2.13x). Code size gains are most pronounced in AES Encryption, due to the very high density of the *Zkne* instructions, executing in a short block of instructions behavior that would otherwise be spread out across many function calls. The same applies to the reduction in static data and stack usage, seeing as there is no need for neither a SBOX LUT to be stored in memory nor a stack-allocated temporary state for computing `ShiftRows` and `MixColumns`. The total memory usage gains are 4.16x, 1.58x and 1.63x for AES (both), SHA-256 and SHA-512.

For the SHA-2 algorithms, gains in code size are attributed more significantly to optimizing assembly by hand rather than *Zknh* instructions, maximizing the use of compressed instructions. *Zknh* instructions make better manual register allocation possible by computing sigma functions with the use of fewer temporary registers, evidenced by the proportional increase of compressed instructions, shown in Table II.

C. Die Area Cost Evaluation

We synthesize both the baseline and extended cores using a high-density 8-track cell library for a 28 nm FDSOI process from ST Microelectronics. Results are shown in Table III. We use Cadence Genus version 1903 with the PLE flow, aiming to obtain the best possible correlation to physical implementation.

TABLE II
IBEX PERFORMANCE COUNTERS FOR EXECUTIONS OF THE AES-128, AES-256, SHA-256 AND SHA-512 ALGORITHMS

Performance Counters	AES-128 K.S.		AES-128 Enc.		AES-256 K.S.		AES-256 Enc.		SHA-256 Full		SHA-256 Comp.		SHA-512 Full		SHA-512 Comp.	
Clock Cycles	**1111**	**291**	**13366**	**314**	**1596**	**351**	**18910**	**422**	**7007**	**5049**	**6248**	**4295**	**18650**	**11197**	**17550**	**10091**
Instructions Retired	**697**	**213**	**8110**	**238**	**959**	**244**	**11486**	**324**	**6060**	**4088**	**5619**	**3647**	**16059**	**9286**	**15434**	**8660**
LSU Busy	190	58	2836	52	255	75	3996	69	602	554	424	376	2170	1242	1863	936
Fetch Wait	138	10	640	15	217	19	892	19	145	189	79	128	173	403	95	319
Loads	**146**	**14**	**1872**	**48**	**195**	**15**	**2644**	**64**	**377**	**333**	**340**	**296**	**1368**	**809**	**1306**	**748**
Stores	**44**	**44**	**964**	**4**	**60**	**60**	**1352**	**4**	**225**	**221**	**84**	**80**	**802**	**433**	**557**	**188**
Jumps	0	0	797	1	6	0	1141	1	82	36	62	16	98	36	78	16
Conditional Branches	86	10	407	8	159	13	563	10	118	182	64	128	150	230	80	160
Taken Cond. Branches	72	9	221	4	142	7	309	6	90	139	62	111	122	187	78	143
Compressed Instructions	**473**	**111**	**3381**	**23**	**628**	**71**	**4805**	**29**	**1921**	**2362**	**1793**	**2234**	**5626**	**4485**	**5472**	**4330**
Multiplier Busy	0	0	576	0	0	0	832	0	0	0	0	0	0	0	0	0
Divider Busy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We set a target clock frequency of 500 MHz, which leads to a fair comparison considering both synthesized cores are reported to have 0 ns timing slack. We consider a timing worst-case PVT corner of slow process, 0.75 V and 125 C for synthesis. We perform synthesis instantiating only the standardized performance counters i.e. clock cycles and instructions retired.

TABLE III
SYNTHESIS RESULTS FOR THE BASELINE AND EXTENDED IBEX CORES

Synthesis results	Baseline	Extended
Cell Area (μm^2)	11,238	12,446
Net Area (μm^2)	6,992	7,279
Total Area (μm^2)	18,230	19,726
Cell Instance Count	10,289	11,687
Equivalent NAND2 gates	34,433	38,132

From Table III, we most notably see that implementing the *Zkne* and *Zknh* extensions leads to a 10% increase in cell area. The benefits obtained through the ISEs in performance, energy and memory efficiency always proportionally outweigh the total area cost.

Gallmann et al. [18] report a 31.47 kGE value for a similar Ibex parameterization as our baseline case and tight timing constraints using a commercial synthesis tool. Nişancı et al. [15] report their higher complexity 5-stage core to be of 19.71 kGE in cell area. No information concerning timing constraints and PDK used were provided, only that open-source synthesis tools were used. Our baseline case is expected to present slightly higher area usage than [18], seeing as we use the *SecureIbex* configuration, due mainly to the ECC protected register file. Due to the use of open-source synthesis tools, area results from [13, 14, 15, 16] should not be seen as accurate estimates for area overheads due to cryptography ISEs.

D. Energy Consumption Evaluation

We evaluate energy consumption via gate-level simulations of the synthesized cores as discussed in Section V-C. Switching activity during each function call is collected and fed into Genus for power analysis. We consider a nominal PVT corner of typical process, 0.9 V and 25 C for power analysis. Additionally, we consider memory access energy costs obtained from CACTI [19]. For a 16KB dual-port SRAM with low-power bit-cells in 28 nm technology, values of 63.362 pJ/bit for reads and 41.436 pJ/bit for writes are reported by the tool. Energy results are shown in Fig. 5.

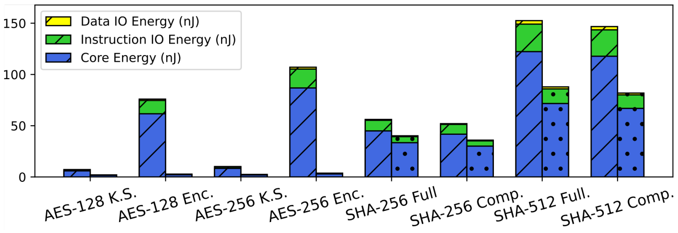

Fig. 5. Energy Consumption Evaluation for the AES-128, AES-256, SHA-256 and SHA-512 algorithms

We can see that energy consumption is dominated by the RISC-V core rather than memory IO. Total energy efficiency gains of 27.81x, 28.91x, 1.45x, and 1.79x are observed for AES-128 Encryption, AES-256 Encryption, SHA-256 Compression, and SHA-512 Compression, respectively.

VI. CONCLUSION

In this work RISC-V scalar cryptography extensions have been demonstrated to greatly accelerate AES-128, AES-256, SHA-256, and SHA-512 in the context of the low-complexity 2-stage Ibex core. Providing a performance gain of 42.57x, 44.81x, 1.45x and 1.74x in terms of clock cycles for AES encryption and the SHA-2 compression functions, the combined *Zkne* and *Zknh* extensions come at a low area overhead of 10%. Furthermore, improvements in memory efficiency (considering code size, static data size and stack usage) of 4.16x, 4.16x, 1.58x and 1.63x have been measured, as well as gains in energy efficiency of 27.81x, 28.91x, 1.45x, and 1.79x.

REFERENCES

[1] M. Dworkin *et al.*, "Advanced Encryption Standard (AES)," 2001-11-26 2001. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.197

[2] Q. Dang, "Secure Hash Standard (SHS)," 2012-03-06 2012. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4

[3] RISC-V Foundation, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213," December 2019. [Online]. Available: https://riscv.org/technical/specifications

[4] ——, "RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions, Document Version v1.0.1," February 2022. [Online]. Available: https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar

[5] lowRISC, "Ibex RISC-V Core," 2018. [Online]. Available: https://github.com/lowRISC/ibex

[6] University of Bristol, "SCARV: a side-channel hardened RISC-V platform," 2018. [Online]. Available: https://github.com/scarv/scarv

[7] ETH Zurich, "PULP platform," 2016. [Online]. Available: https://github.com/pulp-platform/pulpino

[8] A. Zgheib *et al.*, "Extending a RISC-V core with an AES hardware accelerator to meet IOT constraints," in *SMACD / PRIME*, 2021, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/9547962

[9] S. S. Philip *et al.*, "Intermittent Computing Emulation of Ultralow-Power Processors: Evaluation of Backup Strategies for RISC-V," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 1, pp. 82–94, 2023. [Online]. Available: https://doi.org/10.1109/TCAD.2022.3169108

[10] N. Dao *et al.*, "FlexBex: A RISC-V with a Reconfigurable Instruction Extension," in *ICFPT*, 2020, pp. 190–195. [Online]. Available: https://doi.org/10.1109/ICFPT51103.2020.00034

[11] lowRISC, "OpenTitan project," 2017. [Online]. Available: https://github.com/lowRISC/opentitan

[12] Intel, "TinyCrypt Cryptographic Library," 2017. [Online]. Available: https://github.com/intel/tinycrypt

[13] B. Marshall *et al.*, "The design of scalar AES Instruction Set Extensions for RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 109–136, Dec. 2020. [Online]. Available: https://doi.org/10.46586/tches.v2021.i1.109-136

[14] M.-J. O. Saarinen, "A Lightweight ISA Extension for AES and SM4," 2020. [Online]. Available: https://arxiv.org/abs/2002.07041

[15] G. Nişancı *et al.*, "Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms," *Cryptography*, vol. 6, no. 3, p. 41, Aug. 2022. [Online]. Available: https://doi.org/10.3390/cryptography6030041

[16] B. Marshall *et al.*, "Implementing the Draft RISC-V Scalar Cryptography Extensions," in *Hardware and Architectural Support for Security and Privacy*, ser. HASP '20. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458903.3458904

[17] C. Paar and J. Pelzl, *Understanding Cryptography*. Springer Berlin Heidelberg, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-04101-3

[18] N. Gallmann *et al.*, "From Swift to Mighty: A Cost-Benefit Analysis of Ibex and CV32E40P Regarding Application Performance, Power and Area," 2021. [Online]. Available: https://carrv.github.io/2021/papers/CARRV2021_paper_8_Gallmann.pdf

[19] R. Balasubramonian *et al.*, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017. [Online]. Available: https://doi.org/10.1145/3085572