Resource-Constrained Encryption: Extending Ibex with a QARMA Hardware Accelerator

Mathijs De Kremer*, Marco Brohet†, Subhadeep Banik§, Roberto Avanzi‡ and Francesco Regazzoni†§
* VU Amsterdam, The Netherlands, email: m.j.de.kremer@student.vu.nl
† University of Amsterdam, The Netherlands, email: m.j.a.brohet@uva.nl, f.regazzoni@uva.nl
‡ ARM Germany GmbH, Germany and CRI, University of Haifa, Israel, email: roberto.avanzi@gmail.com
§ Università della Svizzera italiana, Switzerland, email: subhadeep.banik@usi.ch, regazzoni@alari.ch

***Abstract*—The increasing prevalence of IoT devices calls for the need for strong, but efficient cryptography. In this paper we present two instruction set extensions for the lightweight encryption cipher QARMA-64 to the RISC-V instruction set, implemented for the Ibex core. The first extension performs the entire algorithm in hardware, divided over ten instructions. The second extension takes a more granular approach and instead implements the basic operations that the algorithm uses as custom instructions. The first extension achieves a speedup of ∼600x over the software implementation and a binary size reduction of over 2x. It achieves these results at the cost of an added field-programmable gate array (FPGA) utilization over the base Ibex design of 43.9% and 18.7% for, respectively, the number of lookup tables (LUTs) and flip-flops (FFs). The application-specific integrated circuit (ASIC) area for synthesis is increased by 92.4% over the base design. The second extension achieves a speedup of ∼19x over the software version while roughly maintaining the same binary size. This extension increases the number of utilized LUTs and FFs respectively by only 0.1% and 4.9%. The ASIC area for this design is increased by only 5.1%. The power consumption for the first extension is estimated at 543µW and for the second extension at 468µW.**

***Index Terms*—QARMA, Ibex, RISC-V, Hardware acceleration, FPGA, IoT, Cryptography, Instruction set extension**

I. Introduction

Internet-of-Things (IoT) devices are everywhere, from fridges to pacemakers to nuclear reactors [1]–[3]. In 2019, over seven billion IoT devices were deployed worldwide and this number is growing rapidly [4]. It is estimated that in 2030 over 25 billion IoT devices will be connected around the globe [4]. This prevalence of smart home, industrial, and medical appliances makes for the challenging task of guaranteeing security. IoT devices come with strict requirements on: 1) computing power, 2) energy, 3) physical chip size, 4) and storage. [5]. We can meet these constraints in two ways: Either we try to 1) optimize a proven block cipher like the Advanced Encryption Standard (AES), as is done in a study by Rouvroy, Standaert, Quisquater, *et al.* [6], or 2) we adopt a block cipher that is designed to be lightweight and to be used in devices with limited resources, like QARMA [7].

Our goal with this paper is to combine both approaches by designing a hardware accelerator for QARMA-64. We have designed this as instruction set extensions for the Ibex processor (formerly known as Zero-riscy [8]), which implements the RISC-V Instruction Set Architecture (ISA) [9]. QARMA is a lightweight cryptographic algorithm. It was used for the ARM pointer authentication[1], a process that verifies the authenticity of a pointer to prevent unauthorized memory access. We have chosen Ibex and QARMA-64 because Ibex was designed for IoT devices and QARMA-64 is a cryptographic algorithm that is designed to be lightweight.

We evaluate the performance of our extensions by the following metrics: 1) binary size, 2) number of clock cycles, 3) the chip area, and 4) power consumption. These metrics correspond with the aforementioned constraints on IoT devices. The size of the binary to execute the algorithm tells us how much additional memory and storage is needed to add encryption to an application. The number of clock cycles that the algorithm takes is important to test because many IoT devices need to meet regular deadlines [10] and adding the overhead from encryption could make IoT devices fail their deadlines. Lastly, the physical area that is added to the chip by extending it with QARMA-specific circuits directly influences the size of the chip and therefore influences the possible use cases of the device and the cost.

We have designed and implemented two approaches. The first approach relies on accelerating the entire algorithm in hardware. The second approach is more fine-grained and mixes accelerating certain parts of the algorithm while keeping some logic in software.

II. Background

A. QARMA

QARMA is a lightweight, tweakable block cipher [7]. Traditional block ciphers rely on modes of operations to provide variability under the same input and key. Tweakable block ciphers on the other hand, gain their variability from a third input, the tweak. On a high-level overview, QARMA can be divided into four parts: 1) key specialization, 2) the forward round functions, 3) the central construction, and 4) the backward round functions.

a) Key specialization: The key specialization begins with partitioning the 128-bit key K into the two 64-bit master keys k^0 and w^0. k^1 and w^1 are then derived from the master keys k^0 and w^0 with a few binary shifts that differ between encryption

[1]https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

DOI 10.1109/ASAP57973.2023.00034

and decryption. This is the only part of the algorithm where decryption and encryption differ.

b) Forward rounds: A forward round function consists of the following steps: 1) AddRoundTweakey, 2) ShuffleCells, 3) Mixcolumns, and 4) SubCells., as illustrated in Fig. 2. The tweak (T) is XOR-ed with the input block for the corresponding round. ShuffleCells and SubCells are permutations on the input block and MixColumns is a matrix multiplication. The forward round function is repeated r times, where $r = 7$ is chosen by the authors for QARMA-64, but $r = 6$ is considered safe against practical attacks [7]. $r = 6$ provides roughly 80 bits of security, which is similar to what PRESENT-80 offers [11]. This corresponds to a total number of rounds of 14 and is ideal for low-security applications in embedded systems where low gate count and low latency are critical.

c) Central construction: The central construction uses a forward round, a backward round, a key addition, and a matrix multiplication. It is designed to prevent reflection attacks [12].

d) Backward rounds: The backward round function is the inverse of the forward round function and is also repeated r times.

B. Ibex

RISC-V core we have chosen to implement an accelerator for QARMA on is Ibex. Ibex, formerly known as Zero-riscy, is a 32-bit core developed by the lowRISC organization. The pipeline of the Ibex core consists of two stages with an optional, experimental third stage. The first stage is the *instruction fetch* stage and the second stage is the *instruction decode and execute* stage.

III. QARMA Accelerators

In this section, we explain the design and implementation of our two approaches.

A. Accelerating the entire algorithm

The first approach we considered was implementing the entire 64-bit QARMA procedure in hardware (configuration A). In this approach, the tweak, input block, and key are loaded as inputs and hardware instructions return the encrypted block. One of the challenges of this approach is that our target architecture is 32-bit and supports at most two-operands-one-output instructions, while the 64-bit version of QARMA uses a 64-bit tweak, a 64-bit block size, and a 128-bit key. This makes for a total input size of 256 bits and an output size of 64 bits. Therefore, we need at least four instructions to load the required inputs. To distinguish between encryption and decryption, an additional instruction is added. Furthermore, two instructions are added to store respectively the lower and upper 32 bits of the output. This makes for a total of seven instructions. Additionally, executing the algorithm in one go would result in circuits that fail to meet the timing constraints of the target field-programmable gate array (FPGA) (10ns). This means that the algorithm can take too long to execute within one clock cycle. Consequently, not being able to meet the timing constraints would result in having to reduce the clock frequency and creating a slower CPU, which is undesirable. To keep the design simple and to be compatible with versions of the Ibex core that only use single-cycle instructions, we decided to limit our instructions to single cycle instructions. By doing so, we had to cut the algorithm into multiple steps. Results from one step are stored in hardware and used as input for the consecutive steps until the final instruction is executed and the results can be written back to the register file. A schematic of this implementation is shown in Fig. 1. The first four instructions load the input data. During the second instruction, we already prepare the tweaks for each round, to save computation room in the later stages. In the same way, we already specialize the keys during the fourth instruction. Instructions five to eight execute the algorithm and finally, the output data are written to the register file with instructions nine and ten.

```
always@(posedge clk_i) begin
  if (write_input_i) begin
    input_r[63:32] <= operand_l_i;
    input_r[31:0] <= operand_h_i;
  end
end
```

Listing 1: An example of how input data are stored for usage in the later stages of the algorithm. operand_l_i and operand_h_i are the two operands coming from the register file. input_r is the register that either contains the plaintext block on encryption or the ciphertext block on decryption.

Storing the input data is done by writing the input signals to internal registers at one of the rising edge of the clock. Which register to write the operands to is decided by the input signals *write_tweak_i*, *write_key1_i*, *write_key2_i*, and *write_input_i* (referring to the input block). These signals come from the decoder and are set according to the current instruction that is being decoded. This process has been illustrated for storing the block to decrypt or encrypt in Listing 1. The operands operand_l_i and operand_h_i, coming from the register file, contain the lower and upper part of the block that is to be encrypted or decrypted. The entire block is saved in an internal register with the name 'input_r'. Storing the other inputs and storing the intermediate results of the algorithm is done in a similar manner.

Since the key specialization differs between encryption and decryption, the decoder provides an output signal called 'decrypt_i'. The combinational circuit for key specialization is shown in Listing 2. 'round_alpha' is the constant *0xC0AC29B7C97C50DD* [7, p.11] (which is hardwired), '^' is the bitwise XOR operation, and $\{s_1, s_2\}$ concatenates s_1 with s_2.

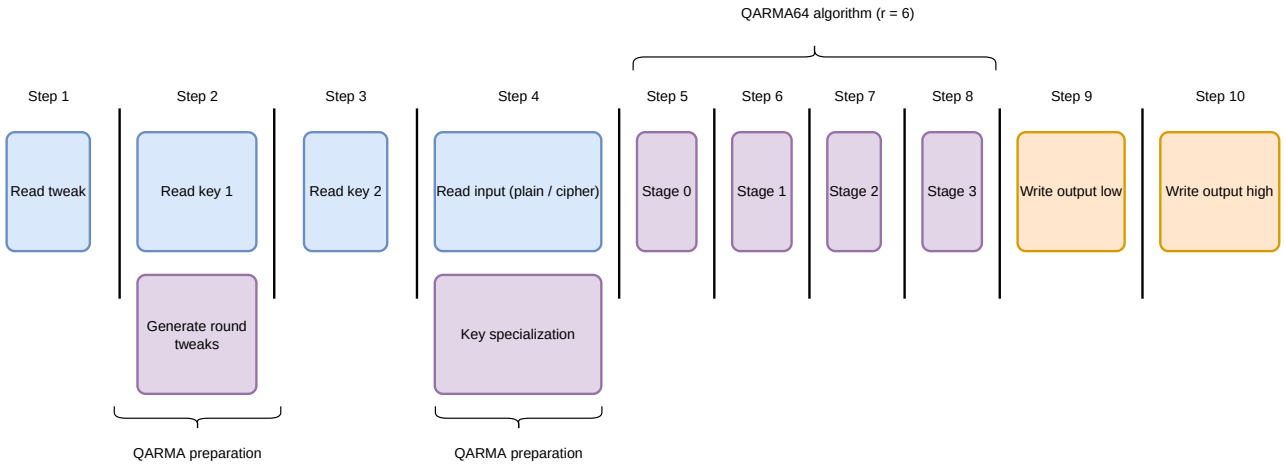

Fig. 1. Illustration of the QARMA 64 implementation as implemented in the Ibex core. The reading of the necessary data from the register file is highlighted in blue, the execution of the algorithm is highlighted in purple, and the writing of the output data is highlighted in orange.

```
always_comb begin
  if (decrypt_i) begin
    w1 = key1_r;
    k0 = key2_r ^ round_alpha;
    w0[63:1] = {w1[0], w1[63:2]};
    w0[0] = w1[1] ^ w1[63];
    k1 = mix_out;
  end else begin
    w0 = key1_r;
    k0 = key2_r;
    w1[63:1] = {w0[0], w0[63:2]};
    w1[0] = w0[1] ^ w0[63];
    k1 = k0;
  end
end
```

Listing 2: The combinational circuit for key specialization. Key specialization differs between encryption and decryption.

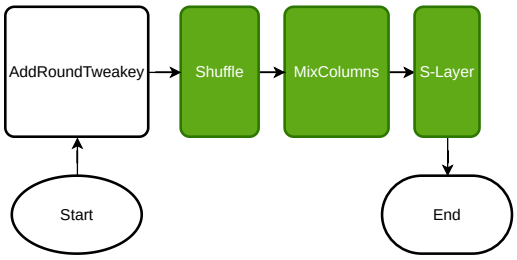

Fig. 2. Illustration showing how acceleration of basic parts of the QARMA algorithm can be used to construct a forward round algorithm. Five types of instructions are added: Shuffle and its inverse, MixColumns, S-Layer, and the Tweak update (not shown). AddRoundTweakey is not accelerated since it is a simple XOR operation.

B. Individually accelerating building blocks

Our second approach identifies more granular building blocks of the QARMA algorithm and accelerates those. The blocks are *S-Layer*, *Tweak update*, *MixColumns*, *Shuffle*, and *Inverse shuffle*. The S-Layer is a substitution used in the forward rounds. The backward rounds use its inverse, but the inverse is the same as the regular S-Layer because we are using S-Box one (which refers to the involutory S-Box σ_1 from [7]). The 64-bit S-box is hardwired into the Ibex core, since not having to load this S-Box over and over is an important part of the expected speedup. Since the second approach is partially defined in software, this approach gives the flexibility to increase the number of rounds for more bits of security. However, to provide a fair comparison with the first approach we will use $r = 6$, for a total of 14 rounds, in the remainder of this paper.

The Tweak update instruction updates the tweak by applying a permutation and using a Linear-Feedback Shift Register (LFSR). The MixColumns operation multiplies each column of the internal state with a matrix. The Shuffle operation is a permutation and is implemented in hardware together with its inverse. In Fig. 2, we illustrate how a forward round can be build from these building blocks. The backward round is the inverse of the forward round and the central construction is build in a similar fashion.

C. Selection of Functions for Hardware acceleration

To determine which parts of the QARMA algorithm are worth implementing as custom instructions, we profiled the reference C implementation[2]. The results are shown in Fig. 3 and show that all parts tested are worth implementing. Especially MixColumns is worth implementing because it occurs 14 times for one run of the entire algorithm. KeySpecialization and KeySpecialization_dec (the encryption and decryption variants of the key specialization) use MixColumns internally, but the number of cycles from the invocation of MixColumns in these functions is not taken into the profiling results because MixColumns can be accelerated separately. KeySpecialization and KeySpecialization_dec occur only once in the algorithm

[2]http://qameleon.site/

and require many inputs, which makes it cumbersome, slow, and expensive to implement as custom instructions. Therefore, KeySpecialization and KeySpecialization_dec are not implemented as custom instructions. After careful consideration, we further decided to not implement updateTweak_inv and SubCell4_inv because they are not needed. updateTweak_inv can be made redundant by saving the tweaks calculated in the forward rounds at an added memory footprint of $r \cdot 64$ bytes. For $r = 6$, this adds 384 bytes. We can substitute occurrences of SubCell4_inv by SubCell4 by observing that the SubCell4 function is an involutory function for S-Box one, the preferred S-Box, meaning that SubCell4 is equal to its inverse for S-Box one. Since all operations that are candidates for being implemented as a custom instruction operate on 64 bits of input, we do not need any additional operations for loading the data from the register file. However, the Ibex architecture only supports writing to one 32-bit register per instruction. To work around this limitation, we simply execute each instruction twice. The first time we write the lower half of the 64-bit output back to the register file and the second time we write the upper half back to the register file.

D. QARMA operations

a) S-Layer: The S-Layer substitutes values of four bits with values defined in the Sbox. The 64 bit block is divided into 16 four-bit-sized cells. These cells are then used as keys in the Sbox. The Sbox that was used is Sbox one [7]. The inputs to the Sbox are the four-bit-sized cells and the output from the Sbox for every cell is written to the corresponding four-bit-sized region in the result signal. Configuration B1 corresponds to solely adding a custom instruction for this operation.

b) Tweak update: The Tweak update operation starts with a permutation, in the same way as the Shuffle operation. However, this time with another permutation h, as chosen by Avanzi [7] and originally defined in Beierle, Jean, Kölbl, *et al.* [13]. After the permutation, some of the 16 four-bit cells of the tweak are updated with one round of an LFSR ω, as defined in the work by Avanzi [7]. ω only updates indices 0, 1, 3, 4, 8, 11, and 13. The implementation of Tweak update as a custom instruction is also referred to as configuration B2 in the remainder of this paper.

c) Shuffle: The Shuffle operation permutes the 16 four-bit cells that make up one block according to the Midori permutation as chosen by Avanzi [7]. We refer the interested reader to the work of Alfarano, Beierle, Isobe, *et al.* [14] for a motivation on how to choose a good permutation for a lightweight cipher. Again, the 64-bit block is divided into 16 four-bit sized cells. These cells are stored in ixd[0:15]. Then, the cells are shuffled according to the permutation defined in signal p and finally connected to the correct four-bit-sized region in the result signal. Configuration B3 that corresponds to the implementation of this operation as a custom instruction. Its inverse (configuration B4) is implemented by exchanging p for its inverse.

d) Mixcolumns: The MixColumns operation is a matrix multiplication of the input block with a circulant matrix [15]. The circulant matrix that is used is the matrix $M_{4,2} = circ(0, \rho_1, \rho_2, \rho_1)$ [7], where ρ_i denotes a circular rotation of the nibble by i bits to the left. The input block (IB) is represented by a 4×4 matrix of 16 four-bit cells. The IB matrix is filled from left to right and top to bottom with the bits from the input block. The resulting matrix multiplication $M \cdot IB$ is shown in Equation 1. Configuration B5 corresponds implementing this operation as a custom instruction.

$$\begin{pmatrix} 0 & \rho_1 & \rho_2 & \rho_1 \\ \rho_1 & 0 & \rho_1 & \rho_2 \\ \rho_2 & \rho_1 & 0 & \rho_1 \\ \rho_1 & \rho_2 & \rho_1 & 0 \end{pmatrix} \cdot \begin{pmatrix} in_{0-3} & in_{4-7} & in_{8-11} & in_{12-15} \\ in_{16-19} & in_{20-23} & in_{24-27} & in_{28-31} \\ in_{32-35} & in_{36-39} & in_{40-43} & in_{44-47} \\ in_{48-51} & in_{52-55} & in_{56-59} & in_{60-63} \end{pmatrix} \quad (1)$$

For a more in depth explanation of the previous functions, we refer the interested reader to the work of Avanzi [7].

IV. Evaluation

In this section, we describe the results of running experiments on our implementations. The RISC-V GNU Compiler Toolchain is used to cross-compile the software for our architectures and the simulations are run on Verilator. The performance metrics that are tested are: *number of clock cycles*, *binary size*, *FPGA utilization*, *application-specific integrated circuit (ASIC) area*, and *power consumption*. For binary sizes and number of clock cycles the benchmark consists of encrypting and decrypting a single input block with the 64-bit QARMA cipher for $r = 6$. We report the number of clock cycles and the binary sizes for all configurations, but FPGA utilization, ASIC area, and power consumption are only reported for configurations 0, A, and B. Configuration A refers to accelerating the entire algorithm, configuration 0 refers to the base Ibex design, and configuration B refers to the individual acceleration of the selected components of the algorithm (further specified by configuration B1 to B5). The aforementioned metrics are not reported for configurations B1 to B5 because configurations B1 to B5 are part of configuration B and share some part of their designs. They are therefore integrated into one extension. Clock cycles and binary sizes are not influenced by this design decision, as we can call each instruction separately, but FPGA utilization, ASIC area, and power consumption are influenced by this decision. Therefore, we only report these metrics for configurations B1 to B5 combined (configuration B).

The configuration that was used for Ibex when running our experiments is the default, relaxed configuration with the multiplication / division unit.

All configurations were designed to meet the timing constraints of the 100MHz Arty A7-100T FPGA, which is the default target of the Ibex Super System [3].

a) Validation: The correctness of our implementations of the QARMA algorithm is verified against the test vectors provided by Avanzi [7], using S-Box σ_1, with $r = 6$, i.e. six forward rounds, two mid rounds and six backward rounds,

[3]https://github.com/GregAC/ibex_super_system

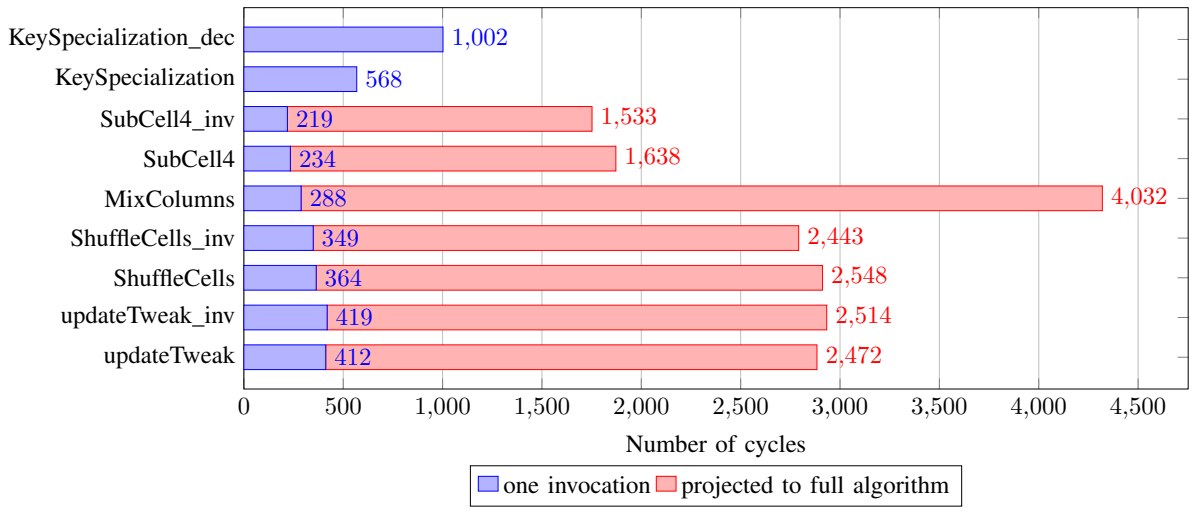

Fig. 3. Profiling the parts of the reference implementation of the QARMA 64 algorithm, ran on Ibex. The results in blue indicate the number of clock cycles the different parts need to execute once. The results in red are the total, projected number of cycles the parts need for $r = 6$. KeySpecialization and KeySpecialization_dec are only executed once. The key specializations internally also use MixColumns. The number of clock cycles needed to execute MixColumns inside the key specializations is not taken into account for the total number of clock cycles needed to execute the key specializations.

for a total of 14 rounds. With these parameters the test inputs and expected output become: • plaintext = fb623599da6e8127, • w^0 = 84be85ce9804e94b, • tweak = 477d469dec0b8762, • k^0 = ec2802d4e0a488e9, • and ciphertext = a512dd1e4e3ec582 [7]. The expected output was obtained in both the simulation environment and on an FPGA.

The benchmark that we used to test our configurations consists of the encryption and decryption of a single 64-bit block. The storing and loading of the data is included in the benchmark. For the software version, the overhead of converting the integers to arrays of bytes is also included in the benchmark.

b) Clock cycles: A speedup graph for the number of clock cycles is shown in Fig. 4. We compare the configurations with the software implementation.
We would expect that the decrease in the number of clock cycles for accelerating the individual parts of the algorithm adds up to the number of clock cycles for configuration B. However, we observe a discrepancy of around 900 clock cycles. We believe that this discrepancy can be explained by the removal of the conversion functions from the code. These conversion functions are needed if one or more software versions of the operations are used and can be discarded from the code by the compiler as an optimization when no software versions of the operations are used.

c) Binary sizes: Another metric that we tested is the size of the binaries that the compiler creates. The size improvements by comparing to the software version are shown in Fig. 5.

d) FPGA utilization: We also evaluated the physical size of the chip. This metric is estimated by two indicators for chip area: *FPGA utilization* and *ASIC area*. FPGA utilization is measured by the number of utilized LUTs and FFs, these are

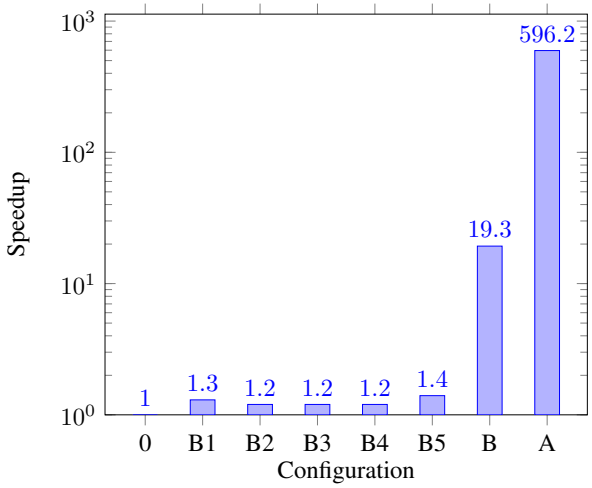

Fig. 4. The speedup of the different configurations over the software version (configuration 0), measured in clock cycles. A logarithmic scale was chosen to highlight the speedup of configuration B in contrast to B1 to B5.

TABLE I
THE NUMBER OF FPGA COMPONENTS AND THE ASIC AREA OF THE CONFIGURATIONS.

Configuration	Number of LUTs	Number of FFs	ASIC area (kGE)
0	4645	3138	23.72
A	2037	867	21.91
B	229	4	1.20

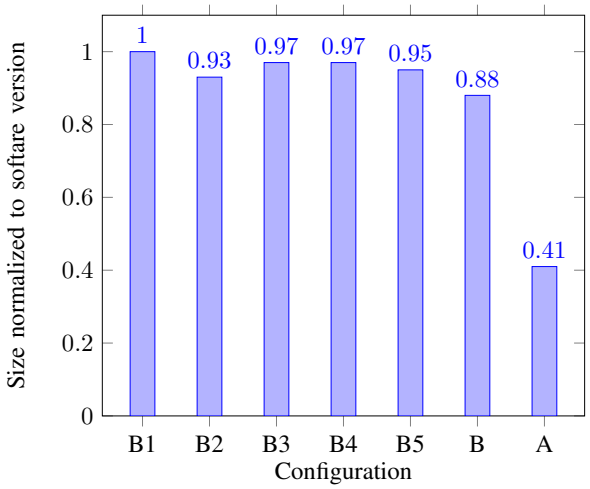

Fig. 5. The size of the binary resulting from compiling the QARMA-64 algorithm for the different configurations normalized to the size of the binary for the software version (configuration 0).

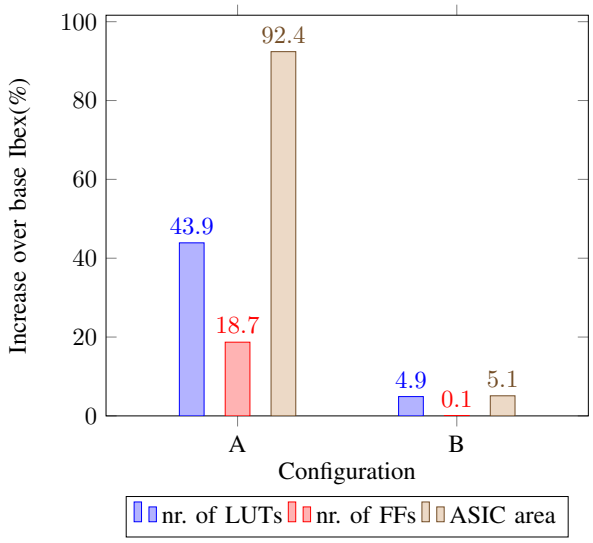

Fig. 6. The relative increase over configuration 0 for the ASIC area and the number of FPGA components when incorporating the extensions into configuration 0. The ASIC area size for configuration 0 has been extracted from the work of Gallmann, Vogel, Schiavone, *et al.* [16].

the most important building blocks of an FPGA. The results as reported by Vivado are shown in Fig. 6. A plot is shown of the relative increases in number of FPGA components used for implementing the custom instructions. The absolute numbers are shown in Table I. The base Ibex architecture uses 4645 LUTs and 3138 FFs. Only configurations 0, A, and B are shown because configurations B1 to B5 are tested on the same architecture as configuration B. This decision was made because configurations B1 to B5 share common logic. By implementing each instruction on the same architecture, we can make use of this shared common logic. This results in fewer LUTs and FFs that have to be used than the sum of implementing each instruction completely separate from the others.

e) ASIC area: The ASIC area is measured in gate equivalence (GE). The size of the base Ibex design has been extracted from the work of Gallmann, Vogel, Schiavone, *et al.* [16]. The increase in size over the base Ibex design (configuration 0) for configuration A and B is shown in Fig. 6. The absolute numbers are shown in The absolute numbers are shown in Table I. Our ASIC analysis used the 45nm FreePDK45 technology and a targeted clock speed of 100MHz.

f) Power consumption: The power consumption results are shown in Table II. The results were acquired using Synopsys Design Compiler with the default switching activity and a clock speed of 100MHz. The reported power estimates for configurations A and B comprise the extensions only (without the Ibex core). The power consumption for configuration 0 is, again, extracted from Gallmann, Vogel, Schiavone, *et al.* [16].

TABLE II
THE ESTIMATED POWER CONSUMPTION OF OUR DESIGNS AS REPORTED BY SYNOPSYS DESIGN COMPILER.

Configuration	Estimated power consumption (μW) at 100MHz
0	800
A	543
B	468

V. RELATED WORK

A. Custom instructions

Several studies have been conducted with the goal of speeding up (parts of) cryptographic algorithms with implementing custom instructions in hardware or making them more efficient in other resources. Most of these studies focus on AES, and no public studies have been conducted as of yet to accelerate (parts of) the QARMA algorithm in hardware, integrated into a CPU design. One such study noticed that most instruction sets lack instructions to perform arithmetic on polynomials [17]. By adding custom instructions for the multiplication of binary polynomials, Tillich and Großschädl [17] managed to achieve a 25% speedup for encryption and 20% speedup for decryption for the AES algorithm on a 32-bit SPARC V8 LEON-2 processor compared to the fastest software implementation of AES available for that CPU. Other custom instructions made by Tillich and Großschädl [18] can achieve a near 10x speedup over a software implementation of AES on a 32-bit processor. They realized this by implementing the S-Box and the MixColumns operations of the AES algorithm as custom instructions, again for the SPARC V8 Leon 2 processor. Grabher, Großschädl, and Page [19] have designed custom instructions for cryptographic primitives with an unconven-

tional approach: they used a technique called bit-slicing to effectively allow an n-bit processor to act as n 1-bit execution units operating in Single Instruction, Multiple Data (SIMD) mode. Their custom instructions are generic enough to be applicable to multiple block ciphers and they have been tested on DES [20], Serpent [21], AES and PRESENT[11].
A comparison of recent custom instruction implementations of lightweight ciphers incorporated into a RISC-V design has been made in [22]. The Instruction Set Extensions (ISEs) in that study are designed on a level of granularity that is comparable to configuration B in this paper. For example, the ISE for Romulus includes custom instructions for *SubCells*, *ShiftRows*, and *AddRoundTweakey*. Speedups in that study range from 1.12x for `Xoodyak` to 45.51x for PHOTON-Beetle. By comparison, configuration B realizes a speedup of 19.3x. The increase in CPU size, in comparison with the base CPU design, is only expressed in FPGA LUTs and ranges from 1.013x for `Xoodyak` to 1.161x for GIFT-COFB (FS). These numbers represent the increases in LUTs of adding the ISE to the RISC-V Rocket core with extensions *Zbkb/x*. By comparison, configuration B increases the number of LUTs by 1.049x. However, any absolute comparison will not be accurate, as we used the Ibex core instead of the Rocket core. These studies show us that the field of researching custom instructions for cryptographic applications is an active field.

B. Lightweight Algorithms

A recent study conducted by Thakor, Razzaque, and Khandaker [23] gives us a comparison of state-of-the-art lightweight cryptographic algorithms. It extensively compares for several metrics, including: latency, software efficiency, hardware efficiency, and power consumption. Furthermore, it gives a comparison of the security of these algorithms by analyzing the vulnerability of the ciphers to several cryptanalysis techniques. Some of the ciphers that stand out are: Simon & Speck [24] and Midori [25]. Simon and Speck both exhibit impressive latency, throughput, and memory efficiency, while Midori performs well with regards to power consumption and hardware efficiency. We also want to point the attention to PRESENT and CLEFIA [26]. These two lightweight ciphers may exhibit less impressive performance characteristics than the aforementioned ciphers [23], but their security has been extensively tested as shown by their standardization into ISO standards *Information security - Lightweight cryptographic - Part 2: Block ciphers* [27] and *Information technology — Automatic identification and data capture techniques — Part 11: Crypto suite PRESENT-80 security services for air interface communications* [28]. Therefore, they are a relatively safe option while still offering good performance [23].

C. Accelerators

Another strategy to employ cryptography in IoT devices is the usage of dedicated hardware accelerators on the device. Accelerators differ from custom instructions in the sense that they are implemented outside of the CPU. They can operate directly on memory that is shared with the CPU. This could make it possible to perform operations in parallel to the CPU. Again, most research in this area is focused on AES. The AES co-processor designed by Hocquet, Kamel, Regazzoni, *et al.* [29] shows us that it is possible to deploy cryptography in μW power environments. Their co-processor can be deployed in passive radio frequency identification (RFID) tags. In other words, their design can be used to run the AES algorithm in devices that have no internal power source, and, instead, are powered by radio waves, for example. Mathew, Satpathy, Suresh, *et al.* [30] have created a co-processor for AES with a throughput of 432 Mbps for encryption and 671 Mbps for decryption, operating at a total power consumption of 13 mW. The co-processor designed by Zhang, Yang, Saligane, *et al.* [31] further improves performance on AES by reducing the design size by 41% and increasing energy efficiency at 432Mbps by 3.1x compared to the co-processor designed by Mathew, Satpathy, Suresh, *et al.* [30].

VI. Conclusions

The rapid market growth of IoT devices has increased the need for encryption on devices with limited resources. The development of methods to provide resource-constrained encryption is essential to protect IoT devices. A cryptographic algorithm that is specifically designed for this use-case is `QARMA`. We have shown two approaches for implementing custom instructions that execute `QARMA` in hardware for the RISC-V architecture Ibex. Our first approach consisted of implementing the full algorithm in hardware. This approach yielded the biggest speedup and binary size reductions, but it came at the cost of significantly larger die area. Our second approach yielded a more modest speedup and very little reduction in binary size. However, this approach increased the design complexity only a little bit. The two designs are on the extreme sides of the granularity spectrum. The first design explores the most coarse-grained approach, while the second design explores the most fine-grained approach. We can think of other designs that lay between these two designs. However, we suspect that those kind of designs will yield lesser speedups than configuration A, while suffering from similar size increases as configuration A, as they will require similar synchronization between operations. Future research will be needed to draw definitive conclusions on design points that lay between configurations A and B.
Since we did not modify the synthesis process for the Ibex Super System, optimizing the implementation of the registers might yield better results. However, these optimizations should be implemented in both the base core and the extended core for a fair comparison.
With these two distinctly differently performing implementations, we provide a choice for IoT hardware designers to decide on the trade-off between performance on the one hand and chip size on the other. We think that the usability of configuration A is limited in practice because of the size of the ASIC almost doubling. However, it does provide upper bounds in terms of chip size and speedup for hardware accelerators for `QARMA`-64. On the other hand, we believe configuration B

can be used in practice because of its more modest chip size increase.

References

[1] J. Rouillard, "The Pervasive Fridge. A smart computer system against uneaten food loss," in *Seventh International Conference on Systems (ICONS2012)*, Saint-Gilles, Réunion, Feb. 2012, pp. 135–140.

[2] S.-Y. Lee, M. Y. Su, M.-C. Liang, *et al.*, "A programmable implantable microstimulator SoC with wireless telemetry: Application in closed-loop endocardial stimulation for cardiac pacemaker," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 5, no. 6, pp. 511–522, 2011. DOI: 10.1109/TBCAS.2011.2177661.

[3] M. Giot, L. Vermeeren, A. Lyoussi, *et al.*, "Nuclear instrumentation and measurement: a review based on the ANIMMA conferences," *EPJ N - Nuclear Sciences & Technologies*, vol. 3, p. 33, Dec. 2017. DOI: 10.1051/epjn/2017023.

[4] L. S. Vailshery. "Number of IoT connected devices worldwide 2019-2030." (Mar. 17, 2022), [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (visited on 05/27/2022).

[5] W. Trappe, R. Howard, and R. S. Moore, "Low-energy security: Limits and opportunities in the Internet of Things," *IEEE Security & Privacy*, vol. 13, no. 1, pp. 14–21, 2015. DOI: 10.1109/MSP.2015.7.

[6] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat, "Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications," in *International Conference on Information Technology: Coding and Computing*, vol. 2, 2004, 583–587 Vol.2. DOI: 10.1109/ITCC.2004.1286716.

[7] R. Avanzi, "The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric Even-Mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-Boxes," *IACR Trans. Symmetric Cryptol.*, no. 1, pp. 4–44, 2017. DOI: 10.13154/tosc.v2017.i1.4-44.

[8] P. Davide Schiavone, F. Conti, D. Rossi, *et al.*, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[9] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, *The RISC-V instruction set manual, volume I: User-level ISA, version 2.0*, May 6, 2014. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf (visited on 05/09/2022).

[10] S. Ahmad, S. Malik, I. Ullah, *et al.*, "An adaptive approach based on resource-awareness towards power-efficient real-time periodic task modeling on embedded IoT devices," *Processes*, vol. 6, no. 7, p. 90, 2018. DOI: 10.3390/pr6070090.

[11] A. Bogdanov, L. R. Knudsen, G. Leander, *et al.*, "PRESENT: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES*, Springer Berlin Heidelberg, pp. 450–466. DOI: 10.1007/978-3-540-74735-2_31.

[12] H. Soleimany, C. Blondeau, X. Yu, *et al.*, "Reflection cryptanalysis of PRINCE-like ciphers," *Journal of Cryptology*, vol. 28, no. 3, pp. 718–744, 2015. DOI: 10.1007/s00145-013-9175-4.

[13] C. Beierle, J. Jean, S. Kölbl, *et al.*, "The SKINNY family of block ciphers and its low-latency variant MANTIS," in *Annual International Cryptology Conference*, Springer, 2016, pp. 123–153. DOI: 10.1007/978-3-662-53008-5_5.

[14] G. N. Alfarano, C. Beierle, T. Isobe, S. Kölbl, and G. Leander, "ShiftRows alternatives for AES-like ciphers and optimal cell permutations for Midori and Skinny," *IACR Transactions on Symmetric Cryptology*, no. 2, pp. 20–47, Jun. 2018. DOI: 10.13154/tosc.v2018.i2.20-47.

[15] R. M. Gray, *Toeplitz and Circulant Matrices: A Review*. now Publishers Inc, 2005. DOI: 10.1561/9781933019680.

[16] N. Gallmann, P. Vogel, P. D. Schiavone, and L. Benini, "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.

[17] S. Tillich and J. Großschädl, "Accelerating AES using instruction set extensions for elliptic curve cryptography," in *International Conference on Computational Science and Its Applications*, Springer, 2005, pp. 665–675. DOI: 10.1007/11424826_70.

[18] S. Tillich and J. Großschädl, "Instruction set extensions for efficient AES implementation on 32-bit processors," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, Springer, 2006, pp. 270–284. DOI: 10.1007/11894063_22.

[19] P. Grabher, J. Großschädl, and D. Page, "Light-weight instruction set extensions for bit-sliced cryptography," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, E. Oswald and P. Rohatgi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 331–345, ISBN: 978-3-540-85053-3. DOI: 10.1007/978-3-540-85053-3_21.

[20] National Institute of Standards and Technology, "Data encryption standard (DES)," *FIPS PUB 46-3*, Oct. 1999. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25#pubs-documentation.

[21] E. Biham, R. Anderson, and L. Knudsen, "Serpent: A new block cipher proposal," in *International workshop on fast software encryption*, Springer, 1998, pp. 222–238. DOI: 10.1007/3-540-69710-1_15.

[22] H. Cheng, J. Großschädl, B. Marshall, D. Page, and T. Pham, "RISC-V instruction set extensions for lightweight symmetric cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, no. 1, pp. 193–237, Nov. 2022. DOI: 10.46586/tches.v2023.i1.193-237.

[23] V. A. Thakor, M. A. Razzaque, and M. R. A. Khandaker, "Lightweight cryptography algorithms for resource-constrained IoT devices: A review, comparison and research opportunities," *IEEE Access*, vol. 9, pp. 28 177–28 193, 2021. DOI: 10.1109/ACCESS.2021.3052867.

[24] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference*, New York, NY, USA: Association for Computing Machinery, 2015. DOI: 10.1145/2744769.2747946.

[25] S. Banik, A. Bogdanov, T. Isobe, *et al.*, "Midori: A block cipher for low energy," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2015, pp. 411–436. DOI: 10.1007/978-3-662-48800-3_17.

[26] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit blockcipher CLEFIA (extended abstract)," in *Fast Software Encryption*, A. Biryukov, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 181–195, ISBN: 978-3-540-74619-5. DOI: 10.1007/978-3-540-74619-5_12.

[27] "Information security - lightweight cryptographic - part 2: Block ciphers," International Organization for Standardization, Geneva, CH, Standard, Nov. 2019. [Online]. Available: https://www.iso.org/standard/78477.html.

[28] "Information technology — automatic identification and data capture techniques — part 11: Crypto suite present-80 security services for air interface communications," International Organization for Standardization, Geneva, CH, Standard, Aug. 2014. [Online]. Available: https://www.iso.org/standard/60441.html.

[29] C. Hocquet, D. Kamel, F. Regazzoni, *et al.*, "Harvesting the potential of nano-CMOS for lightweight cryptography: An ultra-low-voltage 65 nm AES coprocessor for passive RFID tags," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 79–86, 2011. DOI: 10.1007/s13389-011-0005-z.

[30] S. Mathew, S. Satpathy, V. Suresh, *et al.*, "340 mV—1.1 V, 289 Gbps/W, 2090-gate NanoAES hardware accelerator with area-optimized encrypt/decrypt GF$(2^4)^2$ polynomials in 22 nm tri-gate CMOS," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 4, pp. 1048–1058, 2015. DOI: 10.1109/JSSC.2014.2384039.

[31] Y. Zhang, K. Yang, M. Saligane, D. Blaauw, and D. Sylvester, "A compact 446 Gbps/W AES accelerator for mobile SoC and IoT in 40 nm," in *Symposium on VLSI circuits (VLSI-circuits)*, IEEE, 2016, pp. 1–2. DOI: 10.1109/VLSIC.2016.7573553.