

RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography

Hao Cheng¹, Johann Großschädl¹, Ben Marshall², Dan Page³ and
Thinh Pham³

¹ DCS and SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg.
 {hao.cheng, johann.groszschaedl}@uni.lu

² PQShield Ltd, Oxford, UK.
 ben.marshall@pqshield.com

³ Department of Computer Science, University of Bristol, Bristol, UK.
 {daniel.page, th.pham}@bristol.ac.uk

Abstract. The NIST LightWeight Cryptography (LWC) selection process aims to standardise cryptographic functionality which is suitable for resource-constrained devices. Since the outcome is likely to have significant, long-lived impact, careful evaluation of each submission with respect to metrics explicitly outlined in the call is imperative. Beyond the robustness of submissions against cryptanalytic attack, metrics related to their implementation (e.g., execution latency and memory footprint) form an important example. Aiming to provide evidence allowing richer evaluation with respect to such metrics, this paper presents the design, implementation, and evaluation of one separate Instruction Set Extension (ISE) for each of the 10 LWC final round submissions, namely ASCON, Elephant, GIFT-COFB, Grain-128AEADv2, ISAP, PHOTON-Beetle, Romulus, SPARKLE, TinyJAMBU, and XOODYAK; although we base the work on use of RISC-V, we argue that it provides more general insight.

Keywords: ISA, ISE, lightweight cryptography

1 Introduction

The LWC selection process. In a detailed survey of various examples, Bernstein [Ber20] notes that modern, open cryptographic selection processes (or contests) are not without their issues. Set within the broader context of standardised cryptographic functionality, however, they represent an undeniably important and influential mechanism: modulo imperfections stemming from the non-trivial technical *and* non-technical challenges involved, they act to motivate and organise collaborative effort, and, at best, produce more robust outcomes as a result.

After a series of exploratory workshops in 2015 and 2016 and a report [MBTM17] summarising the context and goals, NIST initiated a selection process for LightWeight Cryptography (LWC) via an associated call [SCA18c] released in 2018. The process scope involves two specific forms of cryptographic functionality, with each submission specifying a suite of algorithms with *required* support for an Authenticated Encryption with Associated Data (AEAD) API [SCA18c, Section 3.1], plus *optional* support for a hash function API [SCA18c, Section 3.2]. Although the term is open to interpretation more generally, the call defines lightweight to mean “*tailored for resource-constrained devices*” [SCA18c, Section 1]. This implies said algorithms should, e.g., be 1) efficient on constrained hardware and software platforms (versus existing standards), 2) efficient for short messages, and 3) amenable to countermeasures against implementation attacks.

The 56 round 1 submissions accepted were reduced to 32 round 2 submissions in 2019 [TMcc⁺19], and then again to 10 round 3 or final round submissions in 2021 [TMC⁺21]. The (ongoing) final round is expected to last approximately 12 months, implying a conclusion to the process in 2022. Beyond application of the minimum acceptability requirements [SCA18c, Section 3], a range of factors mean that objective comparison between and then selection of submissions in each round, the final round perhaps most importantly, is a significant challenge. First, even in the final round, there are a large number of submissions and variants thereof. Second, there are a large number of relevant implementation technologies: these include hardware-oriented (e.g., FPGA, ASIC) and software-oriented (e.g., micro-controller) instances. Third, there are a large number of relevant evaluation criteria [SCA18c, Section 4]: focusing on implementation-related examples, and so ignoring the complex, stand-alone challenge of cryptanalytic evaluation, these span at least cost [SCA18c, Section 4.3] (e.g., area and/or memory footprint¹), efficiency [SCA18c, Section 4.3] (e.g., latency, throughput), and resilience to implementation (e.g., side-channel and fault) attack [SCA18c, Section 4.2]. The product of these and other factors demands significant effort be invested, in part due to the design space of implementation techniques (spanning representation of data, and computation with it) and technologies which must be explored.

ISE-supported software implementation. Within said design space, Instruction Set Extensions (ISEs) attempt to add domain-specific support (e.g., state, instructions) to an otherwise general-purpose base Instruction Set Architecture (ISA). Although applicable to many domains, the study of cryptographic ISEs [BGM09, HV11, RI16] spans at least a 25 year period; work by Nahum et al. [NOOS95] is among the first identifiable instances.

As a fundamental and long-lived computer systems interface, the design and extension of an ISA demands careful consideration (cf. [Gue09, Section 4]) and must deliver quantified improvement for the workload of interest to be viable. ISEs often *are* viable, however, because, for example, they represent a hybrid between use of hardware or software alone. This is particularly true with respect to the constrained platforms and evaluation metrics of relevance to the LWC selection process: a well designed ISE *can* result in lower memory footprint and latency than a software-only implementation, *and* greater flexible and efficiency (with respect to improvement per additional logic gate) than a hardware-only implementation.

ISEs were not (explicitly) considered *during* the AES selection process, but, *after* it concluded in 2002, were added to almost every major ISA; at the time of writing, these include (at least) x86 [SCA18a, Section 12.13] (see also [Gue09, DGK19]), POWER [SCA18b, Section 6.11.1], ARMv8-A [SCA20, Section A2.3], SPARC [SCA16, Sections 7.3+7.4], and RISC-V [SCA22, Sections 2.4+2.5] (see also [MNP⁺21]). Using this fact as motivation, we argue that considering ISEs during the LWC selection process is important because doing so offers 1) improved understanding and concrete evidence which can inform the LWC process itself, and 2) preparatory analysis which can inform ISA designers seeking to support the LWC process outcome.

Contributions. As such, this paper makes two central contributions:

1. Based on careful analysis, we present the design, implementation, and evaluation of one separate ISE for each of the 10 LWC final round submissions; for most submissions, our work represents the first exploration of implementations supported by domain-specific ISEs.

¹Whereas the term memory footprint includes, e.g., static and dynamic data, we focus on instruction (or code) throughout: instruction footprint is the amount of memory required to house the instructions for a given implementation, which is a proxy for the number of instructions required.

2. We present a number of novel software-only (i.e., without requiring an ISE) techniques and implementations. For most submissions, our work represents the first exploration of implementations supported by special-purpose, cryptographic Zbkb and Zbkx bit manipulation extensions, an approach which is particularly effective for Elephant (Section 3.3) and GIFT-COFB (Section 3.4); in the latter case, for example, we demonstrate how to optimise bit-sliced implementations of GIFT-128 (as used in GIFT-COFB) using Zbkb, rendering it more efficient than either standard or fix-sliced alternatives for short plaintexts/ciphertexts.

Note that all material associated with the paper, e.g., documentation and source code relating to all hardware and software implementations, are openly available² under an open source license: we expect this material to evolve throughout the remainder of the LWC process and beyond.

Organisation. The paper is organised as follows. In Section 2 we present various background information, including a definition of the scope of and basis for our work. In Section 3 we analyse the LWC final round submissions, and produce associated ISE designs for RISC-V. Then, in Section 4 and Section 5 respectively, we discuss the implementation and evaluation of those designs based on instances of the RISC-V compliant Rocket [AAB⁺16] core.

2 Background

Scope. In part to cope with the large design space considered, and thus engineering effort required, we fix the scope of our work in the following ways:

1. For each submission, we only consider the primary algorithm; each such algorithm is based on a “building block” component or kernel which dominates computation. We only consider intra-kernel ISEs, i.e., ISEs for use *within* a given kernel: the definition of a kernel implies that any extra-kernel opportunities for ISEs have at best a marginal impact, so are not considered viable. Furthermore, we only consider partial implementation of a given kernel where appropriate. Romulus is based on the Skinny-128-384+ kernel, for example, but only uses it to encrypt data; we do not consider support for decryption, therefore, although it would clearly be possible to do so *if* it were more generally useful.
2. We do not consider the hash function API: focusing on the the AEAD API alone seems sufficient, because, for each submission, use of the same kernel is evident across the algorithms which support both APIs.
3. We only consider a 32-bit ISA (and also ISEs for it therefore). Although consideration of a wider set of ISAs is more generally useful, we rationalise this decision by noting it aligns with the (implied) scope of the LWC process: the NIST call outlines a requirement to consider “8-bit, 16-bit and 32-bit microcontroller architectures” [SCA18c, Section 3.4], for example, meaning a 64-bit ISA is deemed out of scope.
4. Although some discussion of the topic is included for completeness in Section 5, we do not consider support in the ISA nor ISEs for countermeasures against implementation attacks (other than their ability to deliver data-independent execution latency). We rationalise this decision by noting it aligns with the (implied) scope of the RISC-V scalar cryptographic extensions [SCA22]: for example, the Zkne and Zknd extensions [SCA22, Sections 2.4+2.5] for AES do not consider interaction with masking-based countermeasures against DPA-like attacks [KJJ99, MOP07].
5. For most submissions, we considered *multiple* ISE design variants. However, we only present results for the *single* ISE design variant we deem most effective, i.e., that which

²See <https://github.com/scarv/lwisc>.

offers the greatest improvement in execution latency per additional logic gate. We stress that the results are therefore a “snapshot”, rather than exhaustive exploration of the (large) design space.

RISC-V. RISC-V (see, e.g., [Wat16]) is an open ISA specification. It adopts *strongly* RISC-oriented design principles (so is similar to MIPS) and can be implemented, modified, or extended by anyone with neither licence nor royalty requirements (so is dissimilar to MIPS, ARM, and x86). A central tenet of the ISA is modularity: a general-purpose ISA can be augmented with a set of special-purpose, standard or non-standard (i.e., custom) extensions. As a result of these features, coupled with the surrounding community and availability of supporting infrastructure such as compilation tool-chains, a range of (typically open-source) RISC-V implementations exist.

In line with our scope, we focus on use of the **unextended ISA** RV32GC [SCA19, Chapter 2] as a starting point. We define the **base ISA**, i.e., a baseline for our work, as the unextended ISA *plus* Zbkb and Zbkx: it therefore includes the 32-bit integer ISA *plus* the general-purpose M (multiplication) [SCA19, Chapter 7], A (atomic) [SCA19, Chapter 8], F (single-precision floating-point) [SCA19, Chapter 11], D (double-precision floating-point) [SCA19, Chapter 12], and C (compressed) [SCA19, Chapter 16] extensions, *plus* the special-purpose, cryptographic Zbkb (a subset of K for bit manipulation instructions) [SCA22, Section 2.1] and Zbkx (a subset of K for crossbar permutation instructions) [SCA22, Section 2.2] extensions. We define an **extended ISA** as then capturing our work, i.e., the base ISA extended with support for an LWC-specific ISE.

Notation. Let $x_{(b)}$ denote an x expressed in radix- or base- b ; the base may be omitted, in which case it is safe to assume $b = 10$. Let $\text{MEM}[i]^b$ denote a b -byte access to some byte-addressable memory, using the address i ; note that where $b = 1$, the access granularity may be omitted. Let $\text{GPR}[i]$, for $0 \leq i < 32$, denote the i -th entry of the general-purpose register file. Note that $\text{GPR}[0]$ is fixed to 0, in the sense reads from it always yield 0 and writes to it are ignored. Let $x \ll y$ and $x \lll y$ (resp. $x \gg y$ and $x \ggg y$) denote left-shift and left-rotate (resp. right-shift and right-rotate) of x by y bits respectively. Let $x \parallel y$ denote concatenation of x and y , and $x_{h\dots l}$ denote extraction of bits h (the high, or more-significant index) through l (the low, or less-significant index) inclusive from some x . RISC-V uses XLEN to denote the word size. We adopt same approach, meaning $\text{XLEN} = 32$ because the context is RV32GC.

3 Design

NIST are careful to use “*algorithm(s)*” throughout [SCA18c, Section 5], presumably to at least allow selection of a suite of rather than a single algorithm. Although one could conclude that multi-algorithm ISEs, i.e., ISEs which support more than one algorithm, are attractive therefore, focusing on them is arguably premature until the outcome is clear. In this section, we therefore adopt a 2-step design process. First, we focus on independently developing an ISE design for each algorithm: each of the following subsections acts to summarise such a design at a high level, with any lower-level technical detail (e.g., instruction encoding, semantics, etc.) deferred to an associated appendix. We use a uniform structure in each such subsection by presenting 1) an overview of the submission, 2) an overview of the kernel within said submission that we focus on, 3) implementation options (including related work, e.g., implementation results), then, finally, 4) a description of the ISE design. Second, and based on the above, Section 3.12 concludes with a broader discussion of opportunities relating to design of ISAs, ISEs, and the algorithms themselves; by taking a broader perspective, this second step therefore highlights if and where multi-algorithm ISEs can be extracted from the single-algorithm ISE designs.

3.1 Constraints

In their study of support for AES in RISC-V, Marshall et al. [MNP⁺21, Section 3] codify a set of ISE requirements to guide their subsequent design process. We adopt the same requirements, which, for completeness, we reproduce here (numbered to match, noting we omit their AES-specific requirement 1):

Requirement 2. *The ISE must align with the wider RISC-V design principles. This means it should favour simple building-block operations, and use instruction encodings with at most 2 source register addresses and 1 destination register address.*

Requirement 3. *The ISE must use the RISC-V general-purpose scalar register file to store operands.*

Requirement 4. *The ISE must not introduce special-purpose architectural state, nor rely on special-purpose micro-architectural state.*

On one hand, we recognise that adopting these constraints means potential ISE designs might be ignored; this fact potentially renders our results sub-optimal, at least versus a more permissive alternative where the constraints are *not* adhered to. A pertinent example is the approach of Steinegger and Primas [SP21], which captures the 320-bit ASCON state within 10 general-purpose registers then used as input and output by a tightly-coupled accelerator for an entire round. This approach may be reasonable for a specific use-cases, and variants of it are in fact viable for *all* the LWC candidates. However, the approach violates Requirement 2: although a useful option in the overall design space, our approach (namely a focus on more traditional, RISC-like ISEs) is fundamentally different.

On the other hand, we argue that the same constraints maximise potential utility of our ISE designs. For example, within the context of RISC-V they 1) support multiple implementation options, including a more traditional integrated approach or via the in-development Custom Function Unit (CFU)³ specification, and 2) offer an easier route to standardisation and deployment as a result of limiting impact on other aspects of the ISA. Beyond this, the constraints also permit extrapolation to other ISAs, e.g., via the ARMv8-M custom instruction mechanism [CP20]; doing so would be more difficult otherwise.

3.2 ASCON

Submission overview. The ASCON [DEMS21] submission specifies the AEAD algorithms [DEMS21, Section 2.4] ASCON-128, ASCON-128A, and ASCON-80PQ, and the hash function algorithms [DEMS21, Section 2.5] ASCON-HASH and ASCON-HASHA. We focus on the primary algorithm ASCON-128, and, more specifically therefore, a kernel represented by the p^a and p^b permutations [DEMS21, Section 2.6] (a single permutation p , often referred to as ASCON- p , with a and b rounds respectively).

Kernel overview. The ASCON- p permutation manipulates a 320-bit state, which is organized in five 64-bit words, by iteratively applying a round function p . This round function is essentially a Substitution-Permutation Network (SPN) and comprises three parts: (i) the addition of an 8-bit round constant c_r to a 64-bit state-word, (ii) a substitution layer that operates across the five words of the state and implements an affine equivalent of the S-box in the χ mapping of KECCAK, and (iii) a permutation layer consisting of linear functions that are similar to the Σ functions in SHA2 and performed on each state-word individually. The S-box maps five input bits to five output bits and is applied to each column of the state, whereby the five state-words are arranged vertically.

³<https://cfu.readthedocs.io>

Implementation options. The substitution layer is normally implemented in a bit-sliced fashion using logical ANDs, XORs, and NOTs. On the other hand, the permutation layer performs an operation of the form $x = x \oplus (x \ggg n) \oplus (x \ggg m)$ on each 64-bit word x of the state. On 32-bit architectures, the ASCON- p permutation is usually implemented in a Bit-Interleaved (BI) fashion, which means each 64-bit word of the state is split up into two 32-bit words, one containing the bits at even positions and the other the bits at odd positions. This representation has the advantage that one can perform a 64-bit rotation through two 32-bit rotations, which is particularly beneficial on 32-bit ARM Cortex-M microcontrollers due to their “free” rotations. Even though bit-interleaving has the potential to speed up the linear functions of ASCON- p on any 32-bit platform (including RV32), one has to take into account that this performance gain for the permutation comes at the expense of conversions between the BI representation and normal representation whenever data is injected into or extracted from the state.

ISE description. The substitution layer consists of logical operations on 64-bit words, which can be split up into two operations on 32-bit chunks. An optimized implementation of the S-box requires 17 native RV32GC instructions [CJL⁺20], which can be reduced to 15 with the help of two Zbkb instructions. The permutation layer can achieve a more significant speed-up since its operations of the form $x = x \oplus (x \ggg n) \oplus (x \ggg m)$ map naturally to two custom `sigma` instructions that use the upper and lower part of a 64-bit state-word as input and produce either the upper or lower part of the result. The rotation amounts can be specified through immediate values. In this way, the instruction-count of the full permutation layer can be reduced from 80 (i.e., 16 per-word) to only 10. This reduction of the number of instructions to 10 is independent of whether bit-interleaving is applied or not, which means that using the BI representation has actually an adverse impact on the overall performance due to the conversions between BI and normal representation.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix A.

3.3 Elephant

Submission overview. The Elephant [BCDM21] submission specifies the AEAD algorithms

Dumbo	=	Elephant-Spongent- π [160]
Jumbo	=	Elephant-Spongent- π [176]
Delirium	=	Elephant-Keccak- f [200]

We focus on the primary algorithm Dumbo, and, more specifically therefore, a kernel represented by the Spongent- π [160] permutation (see also [BKL⁺13]).

Kernel overview. Spongent- π [160] used in Dumbo is a 80-round Spongent permutation [BKL⁺13] (essentially a PRESENT-type permutation [BKL⁺07]). It operates on a 160-bit state and consists of three layers in each round: 1) XORing the state with two round constants, of which one is computed by a 7-bit LFSR ICounter_{160} , i.e., $0^{153} \parallel \text{ICounter}_{160}(i)$, while the other one is $\text{rev}(0^{153} \parallel \text{ICounter}_{160}(i))$, where i denotes the round index and rev is a function reversing the order of the bits of its input; 2) sBoxLayer_{160} , a 4-bit S-box applied 40 times in parallel; 3) pLayer_{160} , moving the bit j of state to bit position $40 \cdot j \bmod 159$ while the bit 159 keeps unmoved.

Implementation options. We developed the pure-software implementation of Spongent- π [160] from scratch by ourselves, in which we presented several optimisation techniques based on our base ISA. The 160-bit state is stored in five 32-bit words $S_0, S_1, S_2, S_3,$

and S_4 , where each S_i stores bits $32i$ to $32i + 31$ of the state. First, we precompute all the round constants so that the first layer is simplified to require only few instructions to load/prepare the constants plus then two XOR instructions. Second, Zbkb provides a dedicated instruction for the parallel 4-bit S-box, namely `xperm4`, which is very beneficial for `sBoxLayer160`. Concretely, the `xperm`-style look-up table for `sBoxLayer160` is construct with three registers before `Spongnt- π [160]` starts:

```
li r1, 0xF4120BDE ; the lower half of S-box look-up table
li r2, 0x63C958A7 ; the higher half of S-box look-up table
li r3, 0x88888888 ; the mask used in xperm-style S-box
```

Each 32-bit word S_i (stored in `rx`) can perform eight 4-bit S-boxes simultaneously with two `xperm4` and two XOR instructions via

```
xperm4 ry, r1, rx
xor    rx, rx, r2
xperm4 rx, r2, rx
xor    rx, rx, ry
```

so in each round the whole `sBoxLayer160` needs 20 instructions in total. Last, we divide the `pLayer160` into two steps: 1) for each word S_i , we firstly apply the `unzip` instruction (from Zbkb) twice and thus make S_i be a form shown in the 3rd row of Figure 1; 2) we then take advantage of eight `SWAPMOVE` operations (`SWAPMOVE` will be explained in detail in Section 3.12) to swap the bits between different words, i.e.,

```
SWAPMOVE(S0, S1, 0x000000FF, 8);    SWAPMOVE(S0, S2, 0x000000FF, 16);
SWAPMOVE(S0, S3, 0x000000FF, 24);    SWAPMOVE(S1, S2, 0x0000FF00, 8);
SWAPMOVE(S1, S4, 0x000000FF, 24);    SWAPMOVE(S2, S3, 0x0000FF00, 8);
SWAPMOVE(S2, S4, 0x0000FF00, 16);    SWAPMOVE(S3, S4, 0x00FF0000, 8);
```

and, afterwards, we use three `rori` instructions (for right-rotation, also from Zbkb) to make S_1 , S_2 , and S_3 correctly-aligned.

ISE description. At first, we designed a custom instruction for the parallel 4-bit S-box, where we integrated the first step of `pLayer160` (i.e., two “unzip” instructions) at the end. Moreover, we designed two instructions for the specific `SWAPMOVE` operations used in our second step of `pLayer160`. Because each of our custom instruction has 1 destination register and each `SWAPMOVE` swaps bits between two different words, so 2 custom instructions are therefore required to perform one complete `SWAPMOVE` here. We also integrated the final three right-rotations into the custom instruction to further reduce the latency.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix B.

3.4 GIFT-COFB

Submission overview. The GIFT-COFB [BCI⁺21] submission specifies an eponymous AEAD algorithm. We focus on this, the only and therefore primary algorithm, and, more specifically therefore, a kernel represented by the GIFT-128 block cipher (see also [BPP⁺17]).

Kernel overview. GIFT-128, belonging to GIFT block cipher family, is based on a SPN with a key length and a block size of both 128 bits. It is a 40-round block cipher with an identical round function that consists of three steps, namely `SubCells`, `PermBits`, and `AddRoundKey`. A typical technique to implement GIFT-128 is bit-slicing [BPP⁺17], where the 128-bit cipher state is expressed as four 32-bit slices S_0 , S_1 , S_2 , and S_3 . `SubCells` is essentially a 4-bit S-box, which needs 11 bitwise logical operations in bit-slicing. `PermBits` has a special property that bits in S_i remain in the same slice through the permutation.

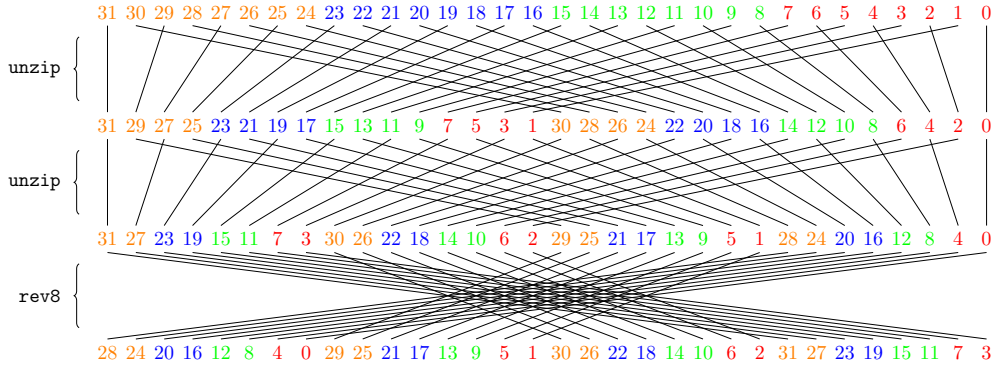


Figure 1: PermBits using instructions from Zbkb. The numbers denote the bit indices of the input 32-bit state slice.

AddRoundKey includes three sub-steps: add round key (to S_1 and S_2), add round constant (to S_3), and key state update (with a main operation of two 16-bit word-wise rotations). We refer readers to [BPP⁺17] or the GIFT-COFB specification [BCI⁺21] for more details.

Implementation options. In addition to naive bit-slicing, a new representation for GIFT-128, namely the fix-slicing, is proposed in [ANP20]. In this work, we considered both different types of state representation for GIFT-128. According to [ANP20], fix-slicing is faster on 32-bit ARM Cortex-M microcontrollers in relation to the naive bit-slicing. However, thanks to Zbkb instructions, we are able to execute the PermBits very efficiently, which makes naive bit-slicing outperform fix-slicing on our base ISA. In detail, only three or four instructions are required in order to permute a 32-bit state slice S_i in each PermBits operation (we save the last rori for S_3):

```
unzip rx, rx
unzip rx, rx
rev8  rx, rx
rori  rx, rx, imm
```

Figure 1 illustrates how unzip and rev8 permute bits (of a single S_i) during PermBits, from which we observe that the output of rev8 is already the output for S_3 [BCI⁺21, Table 2.2]. For S_0 , S_1 , and S_2 , we just further rotate the resulting state slice to the right (using rori) with the corresponding offset (i.e., 24, 16, and 8 respectively).

Furthermore, Zbkb can also speed up the key state update operation. Concretely, we assume a 32-bit key state word $W_6 \parallel W_7$ (stored in `rx`). With the help of pack instruction, we can quickly obtain $W_6 \ggg 2 \parallel W_7 \ggg 12$ through

```
pack ry, rx, rx ; ry = ( W7      ) || ( W7      )
rori rx, rx, 16 ; rx = ( W7      ) || ( W6      )
pack rx, rx, rx ; rx = ( W6      ) || ( W6      )
rori ry, ry, 12 ; ry = ( W7 >>> 12 ) || ( W7 >>> 12 )
rori rx, rx, 2  ; rx = ( W6 >>> 2  ) || ( W6 >>> 2  )
pack rx, ry, rx ; rx = ( W6 >>> 2  ) || ( W7 >>> 12 )
```

ISE description. We implemented both the fix-slicing and the naive bit-slicing implementation of GIFT-128 on the base ISA, and designed ISE for each of them. The fix-slicing implementation separates the computation of round key-update from the main GIFT-128 and uses an efficient round key pre-computation to align with the fix-slicing representation. On the other hand, the ISE for the bit-slicing implementation includes only two instructions to accelerate PermBits and the key state update, respectively. In essence, the ISE for fix-slicing include an instruction for the so-called SWAPMOVE operation (which

will be discussed in detail in Section 3.12), three instructions for the rotation of nibbles, bytes, and halfwords in a 32-bit register, whereby the rotation amount is encoded as an immediate value, and three further instructions for the key-update function. The latter three instructions perform a sequence of SWAPMOVES and operations that consist of rotations of 32-bit words, logical ANDs with a constant, and logical ORs. Each of the three key-update instructions operates on a single 32-bit word.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix C.

3.5 Grain-128AEADv2

Submission overview. The Grain-128AEADv2 [HJM⁺21] submission specifies an eponymous AEAD algorithm. We focus on this, the only and therefore primary algorithm, and, more specifically therefore, a kernel represented by the keystream-generation function of the underlying Grain-128a stream cipher (see also [HJM07, rHJM11]).

Kernel overview. Grain-128a is based on (a variant of) the “original” stream cipher Grain, which was a candidate of the eSTREAM competition and selected for the final eSTREAM portfolio. The kernel is a function that computes a 32-bit word of the keystream using an internal state of a size of 256 bits. This state consists of a 128-bit Linear Feedback Shift Register (LFSR) and a 128-bit Nonlinear Feedback Shift Register (NFSR). The kernel consists of three major sub-functions: one to update the LFSR (called f function), and one to update the NFSR (called g function) and one to compute the 32-bit output word (called h function).

Implementation options. A naive implementation of the sub-functions to update the LFSR and NFSR consists of a large number of bit-level operations. It is therefore more efficient to implement the sub-functions such that they operate on 32-bit words, in which case the kernel basically consists of shifts, ANDs, and XORs. The kernel of Grain-128AEADv2 is simpler (and, therefore, faster) than the kernel of the other NIST finalists, but this simplicity comes at the expense that the kernel is executed more often. Another specific property of this kernel is that the instructions provided by Zbkb/x (e.g. rotations) are not capable to reduce the execution time significantly.

ISE description. The kernel can be accelerated through a set of ten custom instructions, the most important of which is an instruction to extract a 32-bit word that lies at a certain position within a 64-bit word (held in two source registers). Furthermore, the set includes two instructions for the f function, three instructions for the g function, and four for the h function. Each of these instruction gets two state-words as input and computes the contribution of these two state-words to the result of f , g , and h , respectively. Finally, all the contributions have to be XORed together.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix D.

3.6 ISAP

Submission overview. The ISAP [DEM⁺21] submission specifies a family of permutation-based AEAD algorithms [DEMS21, Section 2.5] consisting of ISAP-A-128A, ISAP-K-128A, ISAP-A-128, and ISAP-K-128. We focus on the primary algorithm ISAP-A-128A, and, more

specifically therefore, a kernel represented by the ASCON- p permutation (see [DEMS21] and Section 3.2).

Kernel overview. The main distinguishing feature of the ISAP family is their built-in mode-level countermeasures against passive side-channel attacks. However, from a kernel perspective, the main instance ISAP-A-128A uses exactly the same ASCON- p permutation as the ASCON family of AEAD algorithms. ISAP-A-128A evaluates this permutation over either one, six, or 12 rounds, depending on the concrete (sub-)operation the permutation is part of. As already explained in Section 3.2, ASCON- p operates over a 320-bit state and consists of (i) a round-constant addition, (ii) a substitution layer based on a bit-sliced 5-bit S-box, and (iii) a linear layer performing XORs and rotations of 64-bit words.

Implementation options. Similar to ASCON, optimized implementation of ISAP-A-128A for 32-bit platforms can take advantage of bit-interleaving to speed up the linear layer of the permutation. However, as explained in Section 3.2, bit-interleaving has actually a negative effect on the overall performance when the linear layer is accelerated through a small set of custom instructions. This is because an ISE-supported implementation of the linear layer always consists of only 10 instructions, regardless of whether bit-interleaving is applied or not, which means the conversions between bit-interleaved and normal representation actually slow down the execution.

ISE description. The ISE described in Section 3.2 for ASCON- p can re-used for ISAP-A-128A.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix E.

3.7 PHOTON-Beetle

Submission overview. The PHOTON-Beetle [BCD⁺21] submission specifies the AEAD algorithm family PHOTON-Beetle-AEAD [BCD⁺21, Section 3.2] and the hash function algorithm family PHOTON-Beetle-Hash [BCD⁺21, Section 3.3]. We focus on the primary algorithm PHOTON-Beetle-AEAD[128], and, more specifically therefore, a kernel represented by the PHOTON₂₅₆ permutation (see also [GPP11]).

Kernel overview. The PHOTON₂₅₆ permutation operates on an internal state of 256 bits, organised into an (8×8) -element matrix of 4-bit nibbles. The permutation is SPN-like, consisting of 12 rounds that each apply 4 round functions: these are `AddConstant`, `SubCells`, `ShiftRows`, and `MixColumnsSerial`. Per [GPP11, Section 2.2], the 4-bit PRESENT S-box is used in `SubCells`; in contrast to the AES `MixColumns` round function, `MixColumnsSerial` is specifically optimised to facilitate a serial application of operations in \mathbb{F}_{2^4} .

Implementation options. As reflected by the submission, 3 implementation techniques are applicable to PHOTON₂₅₆; in line with the similar SPN-like structure, and, at least to some extent, round functions, said techniques to analogous to those for AES. First, one can focus on online computation. Doing so mirrors the algorithmic description, whereby each round function is computed; this potentially includes arithmetic in \mathbb{F}_{2^4} , bar small look-up tables, e.g., for the S-box. Second, one can focus on offline pre-computation. Doing so mirrors the AES T-tables technique: the action of `SubCells` and `MixColumnsSerial` is pre-computed using a look-up table, careful indexing into which can also cater for `ShiftRows`. Third, and finally, one can use bit-slicing.

ISE description. The ISE design assumes a column-packed representation, and consists of 1 instruction: the second implementation strategy above is followed, but the look-up table that would normally be computed offline is instead computed online (in hardware). Given an input column, the instruction computes 1 nibble of the output column by applying `SubCells` and `MixColumnsSerial`. This allows 8 such instructions to compute an entire output column (including `AddConstant` and `ShiftRows`, the latter realised simply through indexing of the columns); 64 such instructions can be used to compute an entire round. In a sense, this approach is similar to the design adopted by RISC-V [SCA22, Sections 2.4+2.5] for AES (as documented in [MNP⁺21], stemming from work by Nadehara et al. [NIK04] and Saarinen [Saa20]).

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix F.

3.8 Romulus

Submission overview. The Romulus [GIK⁺21] submission specifies the AEAD algorithms Romulus-N [GIK⁺21, Section 2.4.3], Romulus-M [GIK⁺21, Section 2.4.4], and Romulus-T [GIK⁺21, Section 2.4.5], and the hash function algorithm Romulus-H [GIK⁺21, Section 2.4.6]. We focus on the primary algorithm Romulus-N, and, more specifically therefore, a kernel represented by the Skinny-128-384+ tweakable block cipher (which is a reduced round variant of Skinny-128-384; see also [BJK⁺16]).

Kernel overview. Skinny-128-384 is an SPN-based tweakable block cipher that uses a compact S-box, a very sparse diffusion layer, and a very light key schedule. Due to the high security margin of Skinny, the Romulus designers decided to use a Skinny variant with a reduced number of rounds, namely 40 instead of 56. Skinny-128-384 operates on an internal state of a size of 128 bits that can be viewed as a (4×4) -element matrix of bytes, similar to the AES. The round function is composed of five operations in the following order: `SubCells`, `AddConstants`, `AddRoundTweakey`, `ShiftRows`, and `MixColumns`. `SubCells` applies an 8-bit S-box, which can be efficiently implemented in hardware, to every byte of the state. The `AddConstants` operation XORs some round-dependent constants to the first column of the state. `AddRoundTweakey` extracts eight bytes from the tweak key state and XORs them to the state, whereby the bytes are permuted and updated with simple LFSRs. `ShiftRows` rotates the bytes of the state row-wise to the right by 0, 1, 2, and 3 positions, similar to the `ShiftRows` transformation of the AES. Finally, `MixColumns` multiplies each byte-column of the state by a binary matrix.

Implementation options. The most efficient software implementations of Skinny-128-384 for 32-bit platforms are based on the fix-slicing technique, which can be seen as a special form of bit-slicing [AP20a]. In this work, we considered both the straightforward implementation that uses a look-up table for S-box as well as the fix-slicing implementation.

ISE description. For the table-based implementation, the ISE design assumes a row-packed representation of the state matrix, and can be described as supporting 1) update and use of the round constant (which involves application of an LFSR), 2) update of the tweak key (which involves application of an LFSR), and 3) application of the round functions. Using a row-packed representation, `MixColumns` can be realised via a short sequence of XORs; this allows the latter aspect of the ISE to focus on the remaining, row-oriented round functions, i.e. `SubCells`, `ShiftRows`, and `AddRoundTweakey`. Application of `SubCells` across an entire packed row of the state matrix is rationalised by the low cost S-box design: even if 4 parallel S-box instances are used, the cost in terms of area is still

low in relative terms. For the fix-slicing implementation, the ISE includes instructions for `MixColumns`, specific `SWAPMOVE` operations, and round key pre-computation (e.g., LFSR, key permutation, and key update).

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix G.

3.9 SPARKLE

Submission overview. The SPARKLE [BBdS⁺21] submission specifies the AEAD algorithm family SCHWAEMM [BBdS⁺21, Section 2.3] and the hash function algorithm family ESCH [BBdS⁺21, Section 2.2]. We focus on the primary algorithm SCHWAEMM256-128 and, more specifically therefore, a kernel represented by the SPARKLE permutation [BBdS⁺21, Section 2.1] (see also [BBdS⁺20b], noting underlying use of the Alzette [BBdS⁺20a] ARX-box).

Kernel overview. The SPARKLE permutation consists of three basic building blocks, namely (i) a non-linear layer that is composed of six parallel instances of the ARX-box Alzette, (ii) a simple linear diffusion layer, (iii) the addition of a step counter and round constant to the 384-bit state. Alzette can be seen as a small 64-bit block cipher that operates on two 32-bit words and performs three additions and four XORs whereby one of the operands is rotated by a fixed distance, as well as one ordinary addition and four ordinary XORs. On the other hand, the linear layer is, in essence, a Feistel round with a linear Feistel function, followed by a swap of the left and right half of the state.

Implementation options. An ARM Cortex-M implementation of Alzette consists of only 12 instructions when exploiting the “free” rotation of the second operand. On the other hand, when Alzette is implemented using the base RV32GC instruction set, a total of 33 arithmetic/logical instructions are necessary, which can be reduced to 19 instructions when the bit-manipulation extension Zbkb is available. The linear layer consists of two rotations of 32-bit words (which are part of the so-called ℓ operation) and a number of `xor` and register-move (i.e., `mv`) instructions. Using the base-ISA, the linear layer consists of 32 instructions, among which are six `mv` instructions. However, these `mv` instructions can be avoided when the permutation is fully unrolled, thereby reducing the instruction count of the linear layer to 24. A further reduction by four instructions is possible when using the rotation instructions from Zbkb.

ISE description. There are two basic options for speeding up Alzette with the help of custom instructions. The first is to define instructions for operations of the form $x = x \oplus (y \ggg n)$ and $x = x + (y \ggg n)$, where x and y are two 32-bit words and n is a fixed rotation amount, which can be encoded as an immediate value. In this case, a single instance of Alzette consists of 12 instructions and is very similar to an ARM Cortex-M implementation. A more speed-optimized ISE would consist of two custom instructions, of which one computes the x word of the output and the other the y word. Each of these instructions can be encoded with two source register addresses, one destination register address, and an immediate value specifying one of six 32-bit constants. In this case, Alzette consists of only two instructions. The instruction count of the linear layer can be reduced from 24 to 16 with the help of a custom instruction for the ℓ operation.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix H.

3.10 TinyJAMBU

Submission overview. The TinyJAMBU [WH21] submission specifies an eponymous AEAD algorithm family. We focus on the primary algorithm TinyJAMBU-128 [WH21, Section 3.3], and, more specifically, a kernel represented by the keyed permutation P_n , which is iterated either $n = 640$ times (P_{640}) or $n = 1024$ times (P_{1024}).

Kernel overview. The permutation P is based on a 128-bit non-linear feedback shift register whose feedback path consists of four bit-wise XORs and a bit-wise NAND, which is the only non-linear operation of TinyJAMBU. One can easily identify the state-update function as the most performance-critical operation; it gets besides the 128-bit state and the number of rounds also a key as input. However, TinyJAMBU does not involve a key-schedule. The permutation P_n distinguishes itself from the permutations of other finalists like ASCON, SPARKLE, and XOODYAK by an extremely small state size the fact that it is keyed (i.e., P_n is a non-public permutation). Furthermore, the number of rounds is much higher, which is compensated by an extremely simple round function (basically just a shift of the 128-bit state along with five bit-operations).

Implementation options. On a 32-bit processor, it is possible to compute 32 instances of the permutation simultaneously, which means the XOR and NAND operations are performed on 32-bit words. One of them is a word of the state, one a word from the key and the other four are extracted from the state at certain positions. The latter boils down to extracting a 32-bit word from two adjacent 32-bit state-words through an operation of the form $w = (S_i \gg n) \wedge (S_j \ll (32 - n))$.

ISE description. Extracting a 32-bit words from two state-words can be done with three native RV32GC instructions. However, this operation can be easily mapped to a custom instruction (which we call `fsri`) that reads two 32-bit words from registers and gets the position of the word to extract through an immediate value. Even though `fsri` saves only two instructions, it still improves the execution time of TinyJAMBU significantly since these word-extractions account for about 80% of the execution time of the state-update operation.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix I.

3.11 XOODYAK

Submission overview. The XOODYAK [DHM⁺21] submission specifies an eponymous algorithm, which supports both AEAD and hash function modes. We focus on this, the only and therefore primary algorithm, and, more specifically therefore, a kernel represented by the XOODOO[12] permutation (see also [DHAK18]).

Kernel overview. The state of the XOODOO[12] permutation has the form of a (3×4) -element matrix of 32-bit words, which can be visualized via three horizontal 128-bit planes (one above the other), each consisting of four 32-bit lanes. It is also possible to view the 384-bit state as 128 columns of three bits lying upon another (i.e., each bit belongs to a different plane). As its name indicates, XOODOO[12] executes 12 iterations of a round function consisting of five steps: a column-parity mixing layer θ , a non-linear layer χ , two plane-shifting layers (ρ_{west} and ρ_{east}) between them, and a round-constant addition. Both ρ layers move bits horizontally and perform lane-wise rotations of planes as well as rotations of lanes by 11, 1, and 8 bits to the left. On the other hand, in the parity-computation part

of θ and in the χ layer, state-bits interact only vertically, i.e. within 3-bit columns. The θ layer mainly executes XORs and left-rotations by 5 and 14 bits. Finally, the non-linear layer χ applies a 3-bit S-box to each column of the state, which can be computed using logical ANDs, XORs, and bitwise complements.

Implementation options. An optimised implementation of XODOO[12] permutation on RV32IMAC was proposed in [CJL⁺20]. This implementation takes advantage of a technique known as lane complementing, which allows one to reduce the number of bitwise complements that have to be carried out in the χ transformation from 12 per round to three. However, this optimisation is not necessary on our base ISA, due to the `andn` instruction provided by Zbkb. `andn` combines a logical AND with a bitwise complement of the second operand, which benefits the implementation of χ to be more straightforward and more efficient on our base ISA.

ISE description. When adhering to the requirements for custom instructions mentioned in Section 2, then the only opportunity to speed up XODOO[12] is the manipulation of the parity-plane (i.e., three 32-bit parity-lanes) through an operation of the form $e = (p \lll 5) \oplus (p \lll 14)$. We call the custom instruction implementing this operation `xorrol`.

ISE design. Note that additional, more detailed material relating to the ISE design for this candidate can be found in Appendix J.

3.12 Discussion

Observations regarding ISA design.

- There are several algorithms (e.g., SPARKLE) where operations of the form

$$\text{GPR}[rd] \leftarrow \text{GPR}[rs_1] \odot (\text{GPR}[rs_2] \boxdot imm)$$

for $\odot \in \{\oplus, +, -, \dots\}$ and $\boxdot \in \{\lll, \ggg, \lll, \ggg\}$ are useful. Consider, without loss of generality, an example operation where $\odot = \oplus$ and $\boxdot = \lll$ is realised using the base ISA by the 2-instruction sequence

```
slli rx, ry, imm
xor  rx, rx, rz
```

One could imagine two different approaches to improving this starting point. The arguably more CISC-like approach (see [CDPA16, Section V]) would be to add a dedicated “shift-then-XOR” instruction to the base ISA; more general-purpose instances of this same approach include the ARM Cortex-M “flexible second operand” mechanism. The arguably more RISC-like approach (see [CDPA16, Section VI]) would be to retain the original instructions (resp. micro-ops) only, but implement a mechanism by which they can be fused (or combined, into a macro-op). By using compressed instructions [SCA19, Chapter 16], for example, one can express a similar operation as

```
c.slli ry, imm
c.xor  ry, rz
```

Celio et al. [CDPA16] argue that by fusing these 2 instructions in the micro-architecture front-end, the same (effective) instruction throughput is achieved as use of the 1 non-compressed, dedicated instruction, but, crucially, without “bloating” the base ISA. However, a micro-architecture which supports fusion is more complex as a result; for resource-constrained devices, support for *dynamic*, run-time fusion is potentially unattractive therefore. A conceptual alternative would be *static*, compile-time fusion. If

there were a way to “merge” 2 compressed instructions into 1 non-compressed instruction, their fused semantics could be expressed at compile-time and executed by a less complex micro-architecture.

- There are several algorithms which use 32-bit (e.g., SPARKLE) or 64-bit (e.g., ASCON) rotation. This fact relates to a more general challenge of selecting an n -bit natural word size for an algorithm: one could say that a larger n can be a positive for base ISAs with a large word size (e.g., allowing more effective use of the data-path) but a negative for base ISAs with a small word size (e.g., because n -bit operations need to be synthesised by a sequence of m -bit alternatives, for $m < n$), and vice versa. Put another way, choice of an n somewhat biases how efficient an implementation of the algorithm can be when using a given base ISA.

The other dimension to this choice, however, is how well a particular ISA supports a particular n . There is precedent in RISC-V for supporting 32-bit operations when $XLEN = 64$ (e.g., `rorw` in `Zbkb` [SCA22, Section 3.26] and similar), but not 64-bit operations when $XLEN = 32$. Following a RISC-like design philosophy, the argument would likely be that the latter, e.g., 64-bit rotation, can and so therefore should be synthesised using a sequence of 32-bit instructions. That said, and although total orthogonality is clearly unrealistic, it seems there are some opportunities along similar lines. A pertinent example is a family of so-called funnel shift instructions, which appeared in drafts⁴ of the B extension but not the ratified B (i.e., `Zba`, `Zbb`, `Zbc`, and `Zbs`) nor `Zbkb` extensions. Although counterarguments (e.g., their ternary, 4-address format) exist, one *could* view their omission as a missed opportunity: a general-purpose funnel shift eliminates the need for bit-interleaving (where relevant) without needing a further, special-purpose ISE.

Observations regarding ISE design.

- For some algorithms, an ISE design for RV32GC is harder to scale (or generalise) into one for RV64GC than for other algorithms. `PHOTON-Beetle` uses `PHOTON256`, for example, which uses an (8×8) -element state matrix of 4-bit nibbles. Where $XLEN = 32$ it is possible to pack 1 column into each 32-bit word; where $XLEN = 64$, the natural generalisation is to pack 2 columns into each 64-bit word. However, this natural generalisation of the representation renders the associated implementation more difficult, e.g., with respect to the `ShiftRows` round function.

On one hand, this does not seem a significant problem; it is *already* true of support for AES in RISC-V (cf. `aes32esi` versus `aes64es` in `Zkne` [SCA22, Section 2.5]), for example. On the other hand, however, one *could* also argue that scalability is an attractive property and so favour designs which facilitate it.

- There are several algorithms (e.g., `Elephant` and `Romulus`) where “small” n -bit LFSRs, for $n < XLEN$, are used. Although the LFSR update is typically dominated by other components of a given algorithm, an associated ISE could plausibly offer incremental improvement over use of the base ISA alone; if it were parameterisable (e.g., with respect to the tap sequence), such an ISE could represent a somewhat general-purpose primitive.
- There are several algorithms (e.g., `GIFT` and `Romulus`) where the implementation technique of fix-slicing [ANP20, AP20b] is applicable; this fact is specifically highlighted and explored by Adomnicai and Peyrin [AP20a]. Where fix-slicing is applied, an implementation will often make use of a primitive termed `SWAPMOVE`. May et al. [MPC00, Section 3.1] are among the first⁵ to define and make use of this primitive: the basic idea is that some bits in an operand x are swapped with some bits in another operand

⁴See, e.g., Section 2.9.3 of version 0.93 via <https://github.com/riscv/riscv-bitmanip>.

⁵Their goal is efficient software implementation of permutations, such as those used by DES; they cite some prior art, e.g., noting “[t]his technique is utilised in versions of DES available from the Internet (for example Eric Young’s `libdes`)”.

y , with n and m controlling *which* bits. As such, `SWAPMOVE` has 3 inputs of XLEN bits (x , y , and m), 1 input of $\lceil \log_2 \text{XLEN} \rceil$ bits (n), and 2 outputs of XLEN bits (x and y). In various ISE designs, we cope with the number and type of inputs and outputs through specialisation, e.g., employing 1) a 1-operand variant that involves only x , and 2) a small, hard-coded set of n and m . Given a more general-purpose ISE for `SWAPMOVE` is more attractive, however, it seems useful to carefully explore the trade-off between general- and special-purpose. For example, through careful inter-algorithm analysis, it might be possible to identify a *somewhat* general-purpose set of n and m which afford a compact and so viable encoding.

Observations regarding algorithm design.

- For some algorithms, a change to the interface could plausibly yield more efficient implementations. `PHOTON-Beetle` uses `PHOTON256` for example, which initialises an (8×8) -element state matrix of 4-bit nibbles from a 16-element array of 8-bit bytes using a row-major ordering. Use of a column-oriented representation of the state matrix can imply a significant conversation overhead therefore, which could be reduced by changing the interface to allow a column-major ordering (although doing so clearly then penalises row-oriented representation in the same way).
- For some algorithms, a change to the parameterisation could plausibly yield more efficient implementations. `PHOTON-Beetle`, uses `PHOTON256` for example, which, per [GPP11, Section 2.2], implies use of the 4-bit PRESENT S-box. A different parameterisation is possible, however, which implies use of the 8-bit AES S-box: although reasonable counterarguments also exist, one could argue that opting for the latter will maximise overlap with existing ISEs and so minimise the additional hardware components required (e.g., by using an AES S-box shared with Zkne [SCA22, Section 2.5], if that extension were also supported).

4 Implementation

In the same way as the ISA, a given ISE design represents an interface between hardware and software. In this section we consider both sides of said interface, as defined in Section 3: Section 4.1 considers the hardware-oriented side, i.e., how the ISE is realised, then Section 4.2 considers the software-oriented side, i.e., how the ISE is utilised. Doing so shifts our focus from abstract design to concrete implementation, which then represents the basis for evaluation in Section 5.

4.1 Hardware

Host core. To realise each ISE design, we use the highly configurable, RISC-V compliant Rocket [AAB⁺16] host core. At a high level, the core executes instructions using a 5-stage, in-order pipeline; support is included within the core for a branch prediction mechanism, and in the wider system for a 16 kB instruction cache and a 16 kB data cache.

To support the execution of associated instructions, two modifications are made to the host core for each ISE design. First, an ISE-specific Functional Unit (FU) is integrated into the host core. At least two different approaches are possible, namely 1) an *internal* integration, where the FU is integrated directly into the pipeline, and 2) an *external* integration, which integrates the FU using the Rocket Custom Coprocessor (RoCC) [AAB⁺16, Section 4] interface. Although it requires less micro-architectural modification, using the RoCC interface locates the FU in the commit stage; this can degrade performance, due to inefficiency resulting from how forwarding is implemented. Our ISE designs are intended to permit single-cycle execution, which means the efficiency of forwarding is important. As such, we opt for the former approach, which allows location of

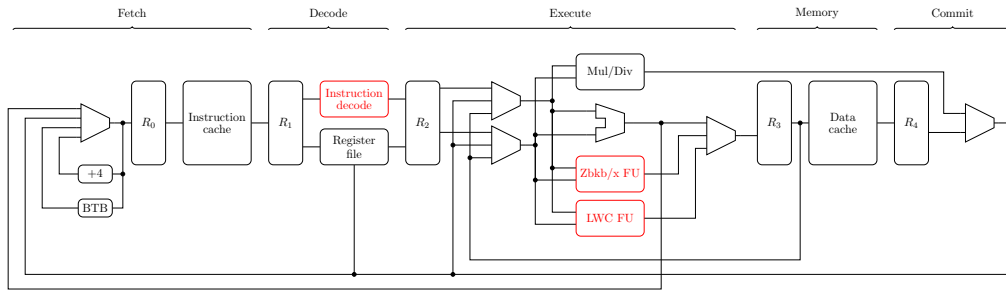


Figure 2: A block diagram highlighting features in our hardware implementation (e.g., integration of the Zbkb/x and LWC FUs) in red. Note that R_i denotes the i -th pipeline register, the component labelled Mul/Div supports multiplication and division, and a Branch Target Buffer (BTB) is shown toward the left-hand end of the pipeline.

the FU in the execute stage. Second, ISE-specific modifications are made to the instruction decoder, which, e.g., allow it to correctly provide input operands to the FU, control the FU so it performs the required computation, and accept output operands from the FU.

What we term the **unextended core**, i.e., Rocket as is, supports RV32GC only. In line with our definition of base ISA, we define the **base core**, i.e., a baseline for our work, as the unextended core *plus* additional⁶ support for Zbkb and Zbkx. We then further extend this base core with support for an LWC-specific ISE, yielding what we term an **extended core**. Figure 2 illustrates the outcome, with our modifications highlighted in red. Note that the Zbkb/x FU realises the Zbkb and Zbkx extensions, so is fixed across all ISE designs; the LWC FU realises a given ISE design, so is different for each ISE design therefore. Also note that neither the Zbkb/x FU nor any of the LWC FU extend the existing critical path, so have no impact on the clock frequency. As such, and by design, the associated instructions have a 1 cycle execution latency.

LWC FU. The implementation of each LWC FU stems fairly directly from the associated ISE definition; each such definition uses pseudo-code which is intentionally similar to the openly available⁷ Register Transfer Language (RTL) implementation used.

Experimental platform. To produce an experimental platform which permits evaluation of, e.g., area and cycle-accurate execution latency, we make use of the SASEBO-GIII [HKSS12]: this includes two FPGAs, namely a Xilinx Kintex-7 (model `xc7k160tfg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6s1x45`) support FPGA. We use the former exclusively, synthesising stand-alone designs for it using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimisation. The FPGA uses a 200 MHz external clock input, which is adjusted into a 50 MHz internal clock signal for use by the host core itself.

4.2 Software

High-level strategy. To utilise each ISE design, we developed a software implementation which can be executed by the associated extended core. For a given algorithm, we start with a base implementation. This is the source code⁸ submitted for a given algorithm. The base

⁶Per Section 2, recall that although Zbkb and Zbkx represent extensions to RV32GC, for example, they form part of the defined base ISA. Viewed from the perspective of the unextended core, however, they represent unsupported extensions and thus need an associated implementation.

⁷See <https://github.com/scarv/lwise>.

⁸For submission X, use of a base implementation Y typically means use of source code located in `X/Implementations/crypto_aead/Y` within the submission archive `X.zip`.

Table 1: A per-algorithm summary of the base and kernel implementations.

Submission	Base implementation	Kernel implementation
ASCON	ascon128v12/ref	P[6 12]
Elephant	elephant160v2/ref	permutation
GIFT-COFB	giftcofb128v1/ref	giftb128
Grain-128AEADv2	grain128aeadv2/x64	grain_keystream32
ISAP	isapa128av20/ref	Ascon_Permute_Nrounds
PHOTON-Beetle	photonbeetleaead128rate128v1/ref	PHOTON_Permutation
Romulus	romulusn/[ref fixslice_opt32]	Skinny[_128_384_plus_enc 128_384_plus]
SPARKLE	schwaemm256128v2/opt	Sparkle_opt
TinyJAMBU	tinyjambu128v2/opt	state_update
XOODYAK	xoodyakround3/ref	Xoodoo_Permute_12rounds

implementation is used as is, with one exception: the submission for **Grain-128AEADv2** was ported from C++ to C, then adapted to cope with, e.g., assumptions around unaligned access to memory. Using appropriate C pre-processor directives, we make minor alterations to the base implementation so the kernel implementation is selectable between the original and a compatible replacement developed by us; Table 1 summarises this information on a per-algorithm basis. We try to be consistent, using the most efficient parameterisation of and implementation strategy for the base implementation which is compatible with our replacement kernel.

We view this approach as effective, in the sense it 1) allows focus on the kernel in question (so limits the volume of work involved), but, equally, 2) allows evaluation of the ISE design within a algorithm-wide rather than kernel-only context (so maximises utility of the outcomes).

Low-level strategy. We use a RISC-V capable instance of the GNU tool-chain⁹ to compile each software implementation. Each replacement kernel implementation is written in assembly language. Rather than modify the tool-chain, instances of the `.insn` directive are used to generate ISE-based instructions.

- Each replacement kernel implementation is captured in a single, leaf function; there is no further opportunity for, e.g., function inlining. We respect the ABI, in the sense that a function prologue and epilogue are careful to preserve and restore any callee-save registers by using the stack.
- Use of an ISE almost always reduces the number of instructions required to implement a replacement kernel, meaning loop overhead which stems from iteration, e.g., over rounds within it, can become more prominent.

To address this while providing at least some consistency, we support either partial, 2-fold unrolling or full, n -fold unrolling (for an appropriate n) of rounds within a replacement kernel. The former is often useful, for example, to avoid unnecessary copying of state output by an i -th round for use as input by the subsequent, $(i + 1)$ -th round.

5 Evaluation

In this section, we present the result of evaluating our ISEs designs from both hardware and software perspectives. As a non-LWC comparison point, we consider an existing¹⁰ ISE-supported implementation of AES-GCM [SCA07]. We attempt to align said implementation as closely as possible with the API used, by 1) “upgrading” it to support additional data, and 2) parameterising it using a 128-bit key. A set of results, limited to the relevant, extended ISA case only, are included for reference alongside those for LWC candidates.

⁹Use of the Rocket core demands a specific tool-chain version; we used commit [b468107e701433e1caca3dbc8aef8d40e0c967ed](https://github.com/riscv/riscv-gnu-toolchain) of <https://github.com/riscv/riscv-gnu-toolchain>, yielding, e.g., a working GCC whose version was 9.2.0.

¹⁰<https://github.com/rvkrypto/rvkrypto-fips>

Note throughout that, within the context of GIFT-COFB, we use FS and BS to refer to implementations based on fix-slicing and bit-slicing respectively; within the context of Romulus, we use FS and TB to refer to implementations based on fix-slicing and look-up tables respectively.

Hardware. Table 2 presents a summary of synthesis results for each ISE design. Reflecting the constraints in Section 3.1, note that all ISE design require combinational logic only, i.e., no state, so we report the number of FPGA Look-up Tables (LUTs) only. We measure (cumulative) overhead relative to the unextended core alone, and so exclude the wider system: doing so seems more representative, in that, e.g., the caches, would dominate otherwise. For example, the ISE for SPARKLE (resp. TinyJAMBU) demands the most (resp. least) area: implementation of the Zbkb/x and LWC FUs produce a 14% and 10% (resp. 3%) overhead respectively, meaning 24% (resp. 17%) cumulative versus the unextended core.

For comparison, the ISE-supported implementation of AES-GCM makes use of Zbkc (for carryless multiplication) [SCA22, Section 2.2], Zbnd (for AES decryption) [SCA22, Section 2.4] and Zbne (for AES encryption) [SCA22, Section 2.5]. Our synthesis results show implementation of these extensions requires 567 additional LUTs, meaning an overhead of 31% cumulative versus the unextended core.

Software: kernel. Table 3 presents a summary of low-level results, focusing on the kernels in isolation. For each kernel, we report both absolute results i.e., execution latency (measured in clock cycles) and instruction footprint (measured in bytes), *and* relative results i.e., increase/decrease factor versus use of the base ISA alone. Note that for some kernels, e.g., GIFT and Romulus, we use auxiliary functions relating to pre-computation of round keys. For clarity, and because our ISEs can be used within them, we include these in addition to the kernel itself.

For comparison, single-block encryption via `aes128_enc_ecb_rvk32` (resp. decryption via `aes128_dec_ecb_rvk32`) using the ISE-supported implementation of AES-GCM requires 324 (resp. 321) cycles; the encryption key schedule via `aes128_enc_key_rvk32` (resp. decryption key schedule via `aes128_dec_key_rvk32`) requires 264 (resp. 719) cycles; the GHASH function (dominated by a multiplication in $\mathbb{F}_{2^{128}}$) via `ghash_mul_rv32` requires 135 cycles.

Software: API. Table 4, Table 5, and Table 6 present a summary of high-level results, focusing on the kernels in context, i.e., as invoked via the API using the `aead_encrypt` and `aead_decrypt` functions for a 16, 128, and 1024 byte plaintext (resp. ciphertext) respectively. This is important, because one kernel may represent a different proportion of the associated algorithm than another, and thus yield different overall improvements. We consider a range of cases, constrained such that the associated data and plaintext/ciphertext lengths are equal: counterarguments clearly exist (e.g., one might expect common use-cases to require a short(er), fixed length associated data, and a longer, variable length plaintext/cipher), but adopting this approach aligns with the NIST micro-controller benchmarking framework¹¹ and so allows easier comparison of results.

For comparison, multi-block encryption via `aes128_enc_gcm` (resp. decryption via `aes128_dec_vfy_gcm`) using the ISE-supported implementation of AES-GCM requires 2144, 7566, and 50742 (resp. 2309, 7716, and 50896) cycles for a 16, 128, and 1024 byte plaintext (resp. ciphertext).

¹¹See, e.g., <https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking>, and results in [TMC⁺21, Section 4 + Appendix A]: note that although the data format allows “*x bytes of associated data and y bytes of message*”, the data itself has $x = y$ in all cases.

Table 2: Hardware-oriented evaluation, i.e., realisation of each ISE design: area measured in FPGA LUTs (plus increase/decrease factor versus unextended core in parentheses).

Submission	Unextended core: RV32GC	Base core: RV32GC + Zbkb/x	Extended core: RV32GC + Zbkb/x + ISE
ASCON	3303 (1.000×)	3764 (1.140×)	4234 (1.282×)
Elephant			3938 (1.192×)
GIFT-COFB (BS)			3906 (1.183×)
GIFT-COFB (FS)			4370 (1.323×)
Grain-128AEADv2			4271 (1.293×)
ISAP			4234 (1.282×)
PHOTON-Beetle			3892 (1.178×)
Romulus (TB)			3998 (1.210×)
Romulus (FS)			4205 (1.273×)
SPARKLE			4097 (1.240×)
TinyJAMBU			3863 (1.170×)
XOODYAK			3814 (1.155×)
AES-GCM			4331 (1.311×)

Table 3: Software-oriented evaluation, i.e., utilisation of each ISE design: latency measured in clock cycles and instruction footprint measured in bytes (plus increase/decrease factor versus base ISA in parentheses) for direct kernel use.

Submission	Kernel	Metric	Replacement kernel implementation	
			Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
ASCON	P6	latency	700 (1.00×)	280 (2.50×)
		footprint	2718 (1.00×)	1050 (2.59×)
Elephant	permutation	latency	15804 (1.00×)	1944 (8.13×)
		footprint	25662 (1.00×)	7702 (3.33×)
GIFT-COFB (BS)	giftb128	latency	1481 (1.00×)	842 (1.76×)
		footprint	5770 (1.00×)	3210 (1.80×)
GIFT-COFB (FS)	giftb128	latency	1386 (1.00×)	972 (1.43×)
		footprint	4888 (1.00×)	3412 (1.43×)
	precompute_rkeys	latency	1306 (1.00×)	251 (5.20×)
		footprint	4830 (1.00×)	768 (6.29×)
Grain-128AEADv2	grain_keystream32	latency	235 (1.00×)	86 (2.73×)
		footprint	858 (1.00×)	262 (3.27×)
ISAP	Ascon_Permute_Nrounds	latency	736 (1.00×)	316 (2.33×)
		footprint	5980 (1.00×)	2364 (2.53×)
PHOTON-Beetle	PHOTON_Permutation	latency	67035 (1.00×)	1473 (45.51×)
		footprint	82486 (1.00×)	3466 (23.80×)
Romulus (TB)	Skinny_128_384_plus_enc	latency	14268 (1.00×)	1502 (9.50×)
		footprint	23508 (1.00×)	4612 (5.10×)
Romulus (FS)	Skinny128_384_plus	latency	6208 (1.00×)	2156 (2.88×)
		footprint	17402 (1.00×)	7274 (2.39×)
	precompute_rtk1	latency	867 (1.00×)	200 (4.34×)
		footprint	2814 (1.00×)	610 (4.61×)
		latency	3402 (1.00×)	1557 (2.18×)
		footprint	11290 (1.00×)	5186 (2.18×)
SPARKLE	Sparkle_opt	latency	1647 (1.00×)	525 (3.14×)
		footprint	5908 (1.00×)	1816 (3.25×)
TinyJAMBU	state_update (P1024)	latency	575 (1.00×)	319 (1.80×)
		footprint	2208 (1.00×)	1184 (1.86×)
XOODYAK	Xoodoo_Permute_12rounds	latency	873 (1.00×)	777 (1.12×)
		footprint	3394 (1.00×)	3010 (1.13×)
AES-GCM	aes128_enc_ecb_rvk32	latency		324
		footprint		556
	aes128_dec_ecb_rvk32	latency		321
		footprint		570
	aes128_enc_key_rvk32	latency		264
		footprint		266
	aes128_dec_key_rvk32	latency		719
		footprint		348
ghash_mul_rvk32	latency		135	
	footprint		252	

Table 4: Software-oriented evaluation, i.e., utilisation of each ISE design: latency measured in clock cycles (plus increase/decrease factor versus unextended ISA in parentheses) for indirect kernel use via AEAD API (with 16 B plaintext, ciphertext, and associated data).

Submission	Functionality	Original kernel implementation	Replacement kernel implementation	
		Unextended ISA: RV32GC	Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
ASCON	aead_encrypt	14801 (1.00×)	7839 (1.89×)	4059 (3.65×)
	aead_decrypt	14523 (1.00×)	7862 (1.85×)	4083 (3.56×)
Elephant	aead_encrypt	3487575 (1.00×)	87596 (39.81×)	14209 (245.45×)
	aead_decrypt	3487689 (1.00×)	87608 (39.81×)	14262 (244.54×)
GIFT-COFB (BS)	aead_encrypt	118062 (1.00×)	6957 (16.97×)	5082 (23.23×)
	aead_decrypt	118058 (1.00×)	6926 (17.05×)	5050 (23.38×)
GIFT-COFB (FS)	aead_encrypt	118062 (1.00×)	7957 (14.84×)	5664 (20.84×)
	aead_decrypt	118058 (1.00×)	7919 (14.91×)	5626 (20.98×)
Grain-128AEADv2	aead_encrypt	15471 (1.00×)	15025 (1.03×)	9962 (1.55×)
	aead_decrypt	15389 (1.00×)	14988 (1.03×)	9917 (1.55×)
ISAP	aead_encrypt	374476 (1.00×)	74480 (5.03×)	45521 (8.23×)
	aead_decrypt	198129 (1.00×)	42721 (4.64×)	25305 (7.83×)
PHOTON-Beetle	aead_encrypt	1407143 (1.00×)	203088 (6.93×)	5224 (269.36×)
	aead_decrypt	1407742 (1.00×)	203254 (6.93×)	5227 (269.32×)
Romulus (TB)	aead_encrypt	161068 (1.00×)	33293 (4.84×)	5287 (30.46×)
	aead_decrypt	161103 (1.00×)	33453 (4.82×)	5318 (30.29×)
Romulus (FS)	aead_encrypt	29686 (1.00×)	36613 (0.81×)	7104 (4.18×)
	aead_decrypt	30093 (1.00×)	36458 (0.83×)	7186 (4.19×)
SPARKLE	aead_encrypt	13141 (1.00×)	5829 (2.25×)	2424 (5.42×)
	aead_decrypt	13166 (1.00×)	5818 (2.26×)	2449 (5.38×)
TinyJAMBU	aead_encrypt	7908 (1.00×)	6690 (1.18×)	3891 (2.03×)
	aead_decrypt	7978 (1.00×)	6761 (1.18×)	3951 (2.02×)
XOODYAK	aead_encrypt	57766 (1.00×)	4191 (13.78×)	3921 (14.73×)
	aead_decrypt	57775 (1.00×)	4200 (13.76×)	3905 (14.80×)
AES-GCM	aes128_enc_gcm			2144
	aes128_dec_vfy_gcm			2309

Table 5: Software-oriented evaluation, i.e., utilisation of each ISE design: latency measured in clock cycles (plus increase/decrease factor versus unextended ISA in parentheses) for indirect kernel use via AEAD API (with 128 B plaintext, ciphertext, and associated data).

Submission	Functionality	Original kernel implementation	Replacement kernel implementation	
		Unextended ISA: RV32GC	Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
ASCON	aead_encrypt	43005 (1.00×)	32316 (1.33×)	16775 (2.56×)
	aead_decrypt	43414 (1.00×)	32694 (1.33×)	17159 (2.53×)
Elephant	aead_encrypt	16044010 (1.00×)	401543 (39.96×)	65118 (246.38×)
	aead_decrypt	16044075 (1.00×)	402787 (39.83×)	65079 (246.53×)
GIFT-COFB (BS)	aead_encrypt	687611 (1.00×)	42048 (16.35×)	31018 (22.17×)
	aead_decrypt	687543 (1.00×)	42093 (16.33×)	30887 (22.26×)
GIFT-COFB (FS)	aead_encrypt	687611 (1.00×)	41884 (16.42×)	33763 (20.36×)
	aead_decrypt	687543 (1.00×)	41749 (16.47×)	33642 (20.44×)
Grain-128AEADv2	aead_encrypt	87682 (1.00×)	85826 (1.02×)	64083 (1.37×)
	aead_decrypt	86656 (1.00×)	84897 (1.02×)	63148 (1.37×)
ISAP	aead_encrypt	489529 (1.00×)	135851 (3.60×)	77577 (6.31×)
	aead_decrypt	285242 (1.00×)	88894 (3.21×)	48138 (5.93×)
PHOTON-Beetle	aead_encrypt	8065027 (1.00×)	1149521 (7.02×)	29372 (274.58×)
	aead_decrypt	8063672 (1.00×)	1150013 (7.01×)	29407 (274.21×)
Romulus (TB)	aead_encrypt	1018364 (1.00×)	213180 (4.78×)	32880 (30.97×)
	aead_decrypt	1017990 (1.00×)	213444 (4.77×)	33049 (30.80×)
Romulus (FS)	aead_encrypt	177043 (1.00×)	203476 (0.87×)	40351 (4.39×)
	aead_decrypt	177326 (1.00×)	203444 (0.87×)	41257 (4.30×)
SPARKLE	aead_encrypt	30033 (1.00×)	12883 (2.33×)	5218 (5.76×)
	aead_decrypt	30053 (1.00×)	12910 (2.33×)	5268 (5.70×)
TinyJAMBU	aead_encrypt	39851 (1.00×)	33574 (1.19×)	19118 (2.08×)
	aead_decrypt	40432 (1.00×)	34033 (1.19×)	19562 (2.07×)
XOODYAK	aead_encrypt	192338 (1.00×)	14579 (13.19×)	13616 (14.13×)
	aead_decrypt	192149 (1.00×)	14397 (13.35×)	13429 (14.31×)
AES-GCM	aes128_enc_gcm			7566
	aes128_dec_vfy_gcm			7716

Table 6: Software-oriented evaluation, i.e., utilisation of each ISE design: latency measured in clock cycles (plus increase/decrease factor versus unextended ISA in parentheses) for indirect kernel use via AEAD API (with 1024 B plaintext, ciphertext, and associated data).

Submission	Functionality	Original kernel implementation	Replacement kernel implementation	
		Unextended ISA: RV32GC	Base ISA: RV32GC + Zbkb/x	Extended ISA: RV32GC + Zbkb/x + ISE
ASCON	aead_encrypt	270239 (1.00×)	228119 (1.18×)	118500 (2.28×)
	aead_decrypt	271095 (1.00×)	230828 (1.17×)	121209 (2.24×)
Elephant	aead_encrypt	109520728 (1.00×)	2749081 (39.84×)	444374 (246.46×)
	aead_decrypt	109520760 (1.00×)	2746736 (39.87×)	444425 (246.43×)
GIFT-COFB (BS)	aead_encrypt	5221431 (1.00×)	322059 (16.21×)	238677 (21.88×)
	aead_decrypt	5220757 (1.00×)	322841 (16.17×)	237408 (21.99×)
GIFT-COFB (FS)	aead_encrypt	5221431 (1.00×)	312881 (16.69×)	258136 (20.23×)
	aead_decrypt	5220757 (1.00×)	312008 (16.73×)	257072 (20.31×)
Grain-128AEADv2	aead_encrypt	663938 (1.00×)	650688 (1.02×)	495442 (1.34×)
	aead_decrypt	655304 (1.00×)	642831 (1.02×)	487569 (1.34×)
ISAP	aead_encrypt	1504643 (1.00×)	631668 (2.38×)	337357 (4.46×)
	aead_decrypt	1029262 (1.00×)	461594 (2.23×)	232652 (4.42×)
PHOTON-Beetle	aead_encrypt	61215512 (1.00×)	8718676 (7.02×)	221919 (275.85×)
	aead_decrypt	61215428 (1.00×)	8717466 (7.02×)	222088 (275.64×)
Romulus (TB)	aead_encrypt	7587976 (1.00×)	1600969 (4.74×)	246905 (30.73×)
	aead_decrypt	7579477 (1.00×)	1605325 (4.72×)	249031 (30.44×)
Romulus (FS)	aead_encrypt	1282828 (1.00×)	1442806 (0.89×)	286404 (4.48×)
	aead_decrypt	1287633 (1.00×)	1441150 (0.89×)	294374 (4.37×)
SPARKLE	aead_encrypt	185179 (1.00×)	78316 (2.36×)	30688 (6.03×)
	aead_decrypt	185202 (1.00×)	78346 (2.36×)	30720 (6.03×)
TinyJAMBU	aead_encrypt	295003 (1.00×)	248622 (1.19×)	140980 (2.09×)
	aead_decrypt	299601 (1.00×)	251993 (1.19×)	144338 (2.08×)
XOODYAK	aead_encrypt	1307532 (1.00×)	98574 (13.26×)	92139 (14.19×)
	aead_decrypt	1306193 (1.00×)	96744 (13.50×)	90319 (14.46×)
AES-GCM	aes128_enc_gcm			50742
	aes128_dec_vfy_gcm			50896

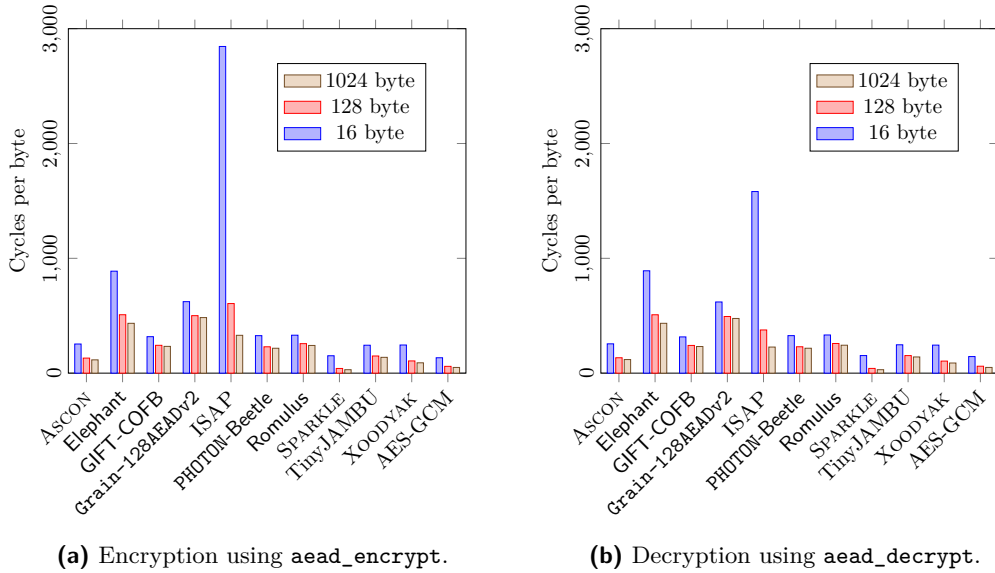


Figure 3: Two graphs summarising the data in Table 4, Table 5, and Table 6, with respect to encryption (left) and decryption (right): for each algorithm, we select the most efficient ISE variant (with respect to execution latency) and plot the normalised cycles per byte across all parameterisations considered (i.e., 16 B, 128 B, and 1024 B plaintext, ciphertext, and associated data).

Finally, Figure 3 presents a similar summary of the data to that used by NIST: for each algorithm, we select the most efficient ISE variant (with respect to execution latency) and plot the normalised cycles per byte across all parameterisations considered (i.e., 16 B, 128 B, and 1024 B plaintext, ciphertext, and associated data). As well as more clearly illustrating relative execution latency, including for the AES-GCM case, the graphs highlight cases where the overhead of initialisation is more (resp. less) effectively amortised for large (resp. small) inputs.

The results for the ISE-supported kernels show that the more hardware-oriented designs (e.g. Elephant, PHOTON-Beetle, Romulus (TB)) are generally accelerated by a larger extent than the more software-oriented designs, such as ASCON, SPARKLE, and XOODYAK, which were already relatively efficient with only the base-ISA. Among the latter three algorithms, SPARKLE achieves a much higher speed-up than XOODYAK, which is mainly because the ARX-box Alzette can be implemented with only two custom instructions since it operates on 64-bit parts of the state (i.e. two 32-bit words). On the other hand, XOODYAK is not particularly well-suited for ISE because it does not contain many operations that can be mapped to custom instructions with two source registers and one destination register.

An additional benefit of the ISE-supported implementations is their significantly smaller code size, which is mainly due to the reduced footprint of the kernels. Such size reductions are often downplayed and only seen as a minor side benefit of ISE, but such a view neglects the fact that a size reduction can yield a non-negligible reduction of execution time on processors with a small instruction cache. For example, according to Table 3, the base-ISA implementations of the kernels of Elephant, PHOTON-Beetle, and Romulus (TB) have a footprint of more than 16 kB and exceed the instruction-cache size of our Rocket core, thereby slowing down the execution due to cache misses. On the other hand, all ISE-supported kernels fit conveniently into the instruction cache.

Comparison with related work: hardware. Strictly limited to cases based on RISC-V, and presented in chronological order, various elements of related work yield useful comparison points.

Tehrani et al. [TGSMD20] describe an ISE for RV32 to support a range of lightweight, 64-bit block ciphers including GIFT-64-128 and Skinny-64-128, implementing and evaluating it using the VexRiscv core. First, they support computation of the substitution layer using a general-purpose instruction for nibble-wise table look-up; doing so is achieved by capturing the table (i.e., S-box) in 3 CSRs, and then applying it nibble-wise to a 32-bit input word supplied in $\text{GPR}[rs_1]$. Second, they support computation of the permutation layer. For GIFT-64-128 this takes the form of a special-purpose instruction, whereas for Skinny-64-128, a general-purpose instruction for nibble-wise matrix-vector multiplication is used; doing so is achieved by capturing a (constant) matrix in 8 CSRs, then applying it to a 64-bit input vector supplied in $\text{GPR}[rs_1]$ and $\text{GPR}[rs_2]$ (with two instructions required to compute the most- and least-significant 32-bit half of the result). We do not present a comparison with this work, because the ISE cannot be used¹² for either GIFT-128-128 or Skinny-128-384+ so is not applicable to GIFT-COFB or Romulus.

Altınay and Örs [AO21] describe an ISE for RV32 to support ASCON- p , implementing and evaluating it using the spike instruction set simulator. Their ISE includes two instructions. First, they support general-purpose rotation; similar instructions are now available via the standard B (bit manipulation) [SCA21, Section 1.3] and K (cryptography) [SCA22, Section 2.1] extensions. Second, they support special-purpose computation of the S-box. Their instruction for doing so is CISC-like, in the sense it operates on data resident in memory: using an input register address rs_1 , it loads five 32-bit inputs $x_i \leftarrow \text{MEM}[\text{GPR}[rs_1] + 4 \cdot i]^4$, applies the S-box to produce outputs r_i from the inputs x_i ,

¹²This is due to, e.g., the diffing substitution and permutation layers used, a fact which stems from the different block size (per [BPP⁺17, Section 2] and [BJK⁺16, Section 2]).

then stores five 32-bit outputs $\text{MEM}[\text{GPR}[rs_1] + 4 \cdot i]^4 \leftarrow r_i$, where $0 \leq i < 5$ throughout. We do not present a comparison with this work, because 1) the ISE falls outside our constraints as outlined in Section 3.1, and, moreover, 2) no non-simulated evaluation results (i.e., area overhead, and cycle accurate execution latency) are available for it.

Steinegger and Primas [SP21] describe an ISE for RV32 to support ASCON- p , implementing and evaluating it using the RI5CY core. Their ISE includes one instruction, which essentially supports computation of an entire ASCON- p round in hardware. Implementation therefore demands tight integration with the core (e.g., using 10 hard-wired general-purpose registers to store the state), which, although delivering performance, arguably renders it more akin to a tightly-coupled accelerator than traditional ISE. Although the ISE falls outside our constraints as outlined in Section 3.1, it *does* represent a competitive trade-off: modulo differences with respect to the core used, [SP21, Table 1 + Section 4] demonstrate that a factor of 1.1 area overhead permits a significant, factor of 50 improvement in execution latency for ASCON. For certain use-cases, this trade-off can be argued as more attractive than one based on a more hardware-oriented (e.g., purely using an IP core) or more software-oriented (i.e., using a more tightly constrained ISE, as in our work) alternative.

Comparison with related work: software. Strictly limited to cases based on RISC-V, and presented in chronological order, various elements of related work yield useful comparison points.

Jellema [Jel19] presents an optimised implementation of ASCON, based on use of an E31 (supporting RV32IMAC) core; [Jel19, Figure 10] suggests a measured $6 \cdot 118 = 708$ cycle execution latency for the 6-round ASCON- p permutation. Modulo the different core, this can be compared with the base ISA and extended ISA columns of Table 3, where we measure 700 and 280 cycles respectively. At face value one might expect use of Zbkb/x to offer greater improvement, but in fact this result is expected: although we can use `andn` and `orn` within the substitution layer, we cannot use `rol` or `ror` within the diffusion layer (because $\text{XLEN} = 32$, so 64-bit rotation is not supported). Alternatively, [Jel19, Figure 11] suggests a measured 552076 cycle execution latency for the encryption of a 4096 byte plaintext and (inferred) 0 byte associated data; for this parameterisation, our implementation takes 479764 cycles using the base ISA or 263043 cycles using the extended ISA, i.e., the LWC-specific ISE.

Lemmen [Lem20] presents an optimised implementation of Elephant, based on use of an E31 (supporting RV32IMAC) core. We do not present a comparison with this work, because it focuses on the non-primary parameterisation Elephant-Keccak- f [200] so falls outside our scope.

Campos et al. [CJL⁺20] present a limited study of LWC algorithms, with the goal of assessing the impact of selecting assembly language versus C for their implementation. Per [CJL⁺20, Section 2], their work is based on use of an E31 (supporting RV32IMAC) or VexRiscv (supporting RV32IM) core; we ignore use of the riscvOVPSim simulator, because, as they explain, it may not produce representative results. Modulo the different core, can be compared with the base ISA and extended ISA columns of Table 3. For ASCON, [CJL⁺20, Table 7] suggests a measured 750 cycle execution latency for the 6-round ASCON- p permutation; per Table 3, use of Zbkb/x means our implementation takes 700 cycles, or 280 with an LWC-specific ISE. For SPARKLE, [CJL⁺20, Section 3.2] suggests an approximated 1708 cycle execution latency for the SPARKLE-384 permutation; per Table 3, use of Zbkb/x means our implementation takes 1647 cycles, or 525 with an LWC-specific ISE. For XOODYAK, [CJL⁺20, Section 3.2] suggests an approximated 1596 cycle execution latency for the 12-round XOODOO permutation; per Table 3, use of Zbkb/x means our implementation takes 873 cycles, or 777 with an LWC-specific ISE.

Renner et al. [RPM20] present a hardware-in-the-loop benchmarking framework for

the LWC process; since their focus is the framework, they use the source code submitted for a given algorithm. Their work is based on use of a Kendryte K210 core. Modulo the different core, their results can be compared with the unextended ISA column of Table 4, Table 5, and Table 6.

Resilience against implementation attack. For constrained platforms of relevance to the LWC selection process, countermeasures against implementation attack are often classified as being either based on hiding [MOP07, Chapter 7] and/or masking [MOP07, Chapter 10]. Although we do not consider such countermeasures per se, some discussion of how our ISEs interact with them may still be useful:

- The principle of constant-time implementation (i.e., that which exhibits data-independent execution latency) is important; delivering it acts as a hiding-based countermeasure against certain forms of attack, and is generally easier for ISE-supported than software-only implementations. We note that *all* our replacement kernel implementations are constant-time, in certain cases¹³ representing an improvement to the base implementation considered.
- Other hiding countermeasures instrumented at the ISA level, e.g., temporal skewing or shuffling, typically apply to ISE-supported implementation much like software-only implementations. That said, however, one can debate whether they are as effective. For example, an ISE-supported implementation will typically comprise fewer instructions, meaning less Instruction Level Parallelism (ILP) to harness through shuffling, and lower diversification. In turn, this acts to limit the security improvement possible.
- The situation for masking-based countermeasures is more involved. For a linear operation, our ISEs can be used on a share-wise basis. For a non-linear operation, this is not possible: one would need to redefine the ISE to accept masked inputs and outputs, and augment the associated FU so it is mask-aware. We note that our adherence to 3-address instruction formats means [GGM⁺21] would be one way to accommodate this for $n = 2$ shares, whereas [MP21] would be another way to do so more generally, i.e., for $n > 2$ shares.
- It is important to note that ISAP is a somewhat special case, in the sense it delivers inherent mitigation for selected side-channel and fault attacks; since this is achieved at the mode level and our ISE applies at the kernel level (i.e., the ASCON- p permutation), we do not expect any negative interaction between said ISE and any security argument for ISAP.

That said, it is important to keep this functionality in mind when interpreting performance results. Although inefficient in relative terms, ISAP includes by-design mitigation that other candidates would have to deliver via post-design means: the resulting overhead is costed into ISAP already, complicating any direct comparison.

6 Conclusion

Summary. ISEs to support standard cryptographic algorithms, e.g., AES, have now been included in almost every major ISA. Anticipating the LWC process will yield an outcome that warrants similar support, this paper investigated ISEs for each of the 10 LWC final round submissions. Through careful analysis of the constituent algorithms, and following a set of principled constraints (e.g., alignment with the wider RISC-V design principles, such as use of 3-address instructions), we first developed ISE designs for ASCON, Elephant, GIFT-COFB, Grain-128AEADv2, ISAP, PHOTON-Beetle, Romulus, SPARKLE, TinyJAMBU, and XOODYAK, then implemented said designs using the RISC-V compliant Rocket core.

¹³It might be an unfair criticism given the overtly explanatory goal, but, for example, the reference implementation of PHOTON-Beetle involves multiplication in \mathbb{F}_{2^4} whose execution latency is data-dependent.

Broadly speaking, comparison with software-only alternatives shows that 1) the ISEs overhead in hardware is low, 2) the ISEs allow a reduction in execution latency, the degree of which is algorithm-dependent but significant in some cases, and, at the same time, 3) the ISEs allow constant-time execution, *and* a reduction in instruction footprint. Put together, these features highlight the value of ISEs within the context of resource-constrained devices and therefore the LWC process.

Observations. Based on our work, several high-level observations seem important to stress. First, and particularly when carefully paired with implementation techniques such as fix-slicing, our results demonstrate software-only implementations using Zbkb/x can be significantly more efficient than using RV32GC alone. This fact paints Zbkb/x (and so also Zbb) in a positive light with respect to general-purpose support: implementations and benchmarking for RISC-V which do *not* consider Zbkb/x (or Zbb) disadvantage it versus, e.g., ARM. Second, our results highlight a difference in relative improvement between algorithms that are more hardware-oriented versus more software-oriented. Put simply, ISEs for the former (e.g., Elephant, PHOTON-Beetle, Romulus) typically offer a greater improvement than for the latter (e.g., ASCON, SPARKLE, XOODYAK): although the most efficient software-only implementations remain so when ISE support is considered, the difference between most and least efficient algorithms is significantly smaller. Stemming from the hybrid nature of ISE-supported software, this fact could be read as complicating the classification of hardware- versus software-oriented algorithms; either way, it highlights the need to consider use of ISEs as part of their evaluation. Third, our results act as evidence that ISEs which target an implementation technique (e.g., fix-slicing) are typically more general-purpose but less efficient, whereas ISEs which target an algorithm are typically less general-purpose but more efficient. Although a somewhat obvious statement, this suggests that once an outcome from the LWC process is known, the latter approach is more sensible in the longer term.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (<http://www.ukrise.org>) programme. The second author has been supported in part by the Fonds National de la Recherche (FNR) Luxembourg via grant C19/IS/13641232 (“Analysis and Protection of Lightweight Cryptographic Algorithms”).

References

- [AAB⁺16] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D.A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.
- [ANP20] A. Adomnicai, Z. Najm, and T. Peyrin. Fixslicing: A new GIFT representation: Fast constant-time implementations of GIFT and GIFT-COFB on ARM Cortex-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(3):402–427, 2020. <https://doi.org/10.13154/tches.v2020.i3.402-427>.

- [AO21] Ö. Altınay and B. Örs. Instruction extension of RV32I and GCC back end for Ascon lightweight cryptography algorithm. In *International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–6, 2021. <https://doi.org/10.1109/COINS51742.2021.9524190>.
- [AP20a] A. Adomnicai and T. Peyrin. Fixslicing - application to some NIST LWC round 2 candidates. In *4-th Lightweight Cryptography Workshop*, 2020. <https://csrc.nist.gov/Events/2020/lightweight-cryptography-workshop-2020>.
- [AP20b] A. Adomnicai and T. Peyrin. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(1):402–425, 2020. <https://doi.org/10.46586/tches.v2021.i1.402-425>.
- [BBdS⁺20a] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Alzette: a 64-bit ARX-box (feat. CRAX and TRAX). In *Advances in Cryptology (CRYPTO)*, LNCS 12172, pages 419–448. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-56877-1_15.
- [BBdS⁺20b] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Lightweight AEAD and hashing using the Sparkle permutation family. *IACR Transactions on Symmetric Cryptology*, 2020(S1):208–261, 2020. <https://doi.org/10.13154/tosc.v2020.iS1.208-261>.
- [BBdS⁺21] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, Amir Moradi, L. Perrin, A.R. Shahmirzadi, A. Udovenko, V. Velichkov, and Q. Wang. SCHWAEMM and ESCH: Lightweight authenticated encryption and hashing using the SPARKLE permutation family. Submission to NIST (version 1.2), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf>.
- [BCD⁺21] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. PHOTON-beetle. Submission to NIST, 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf>.
- [BCDM21] T. Beyne, Y.L. Chen, C. Dobraunig, and B. Mennink. Elephant. Submission to NIST (version 2.0), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf>.
- [BCI⁺21] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S.M. Sim, and Y. Todo. GIFT-COFB. Submission to NIST (version 1.1), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf>.
- [Ber20] D.J. Bernstein. Cryptographic competitions. Cryptology ePrint Archive, Report 2020/1608, 2020. <https://eprint.iacr.org/2020/1608>.

- [BGM09] S. Bartolini, R. Giorgi, and E. Martinelli. Instruction set extensions for cryptographic applications. In Ç.K. Koç, editor, *Cryptographic Engineering*, chapter 9, pages 191–233. Springer, 2009. https://doi.org/10.1007/978-0-387-71817-0_9.
- [BJK⁺16] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S.M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology (CRYPTO)*, LNCS 9815, pages 123–153. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-662-53008-5_5.
- [BKL⁺07] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 4727, pages 450–466. Springer-Verlag, 2007. https://doi.org/10.1007/978-3-540-74735-2_31.
- [BKL⁺13] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62(10):2041–2053, 2013. <https://doi.org/10.1109/TC.2012.196>.
- [BPP⁺17] S. Banik, S.K. Pandey, T. Peyrin, Y. Sasaki, S.M. Sim, and Y. Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 10529, pages 321–345. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-66787-4_16.
- [CDPA16] C. Celio, P. Dabbelt, D.A. Patterson, and K. Asanović. The renewed case for the reduced instruction set computer: Avoiding ISA bloat with macro-op fusion for RISC-V. *CoRR*, abs/1607.02318, 2016. <https://arxiv.org/abs/1607.02318>.
- [CJL⁺20] F. Campos, L. Jellema, M. Lemmen, L. Müller, D. Sprenkels, and B. Viguier. Assembly or optimized C for lightweight cryptography on RISC-V? In *Cryptology and Network Security (CANS)*, LNCS 12579, pages 526–545. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-65411-5_26.
- [CP20] L. Choquin and F. Piry. Arm custom instructions: Enabling innovation and greater flexibility on Arm. Technical report, Arm Ltd., 2020. <https://www.arm.com/why-arm/technologies/custom-instructions>.
- [DEM⁺21] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer. ISAP. Submission to NIST (version 2.0), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/isap-spec-final.pdf>.
- [DEMS21] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon. Submission to NIST (version 1.2), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf>.
- [DGK19] N. Drucker, S. Gueron, and V. Krasnov. Making AES great again: The forthcoming vectorized AES instruction. In *Information Technology New Generations (ITNG)*, AISC 800, pages 37–41. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-14070-0_6.

- [DHAK18] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38, 2018. <https://doi.org/10.13154/tosc.v2018.i4.1-38>.
- [DHM⁺21] J. Daemen, S. Hoffert, S. Mella, M. Peeters, G. van Assche, and R. van Keer. Xoodyak, a lightweight cryptographic scheme. Submission to NIST (version 2.0), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf>.
- [GGM⁺21] S. Gao, J. Großschädl, B. Marshall, D. Page, T. Pham, and F. Regazzoni. An instruction set extension to support software-based masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHESES)*, 2021(4):283–325, 2021. <https://doi.org/10.46586/tches.v2021.i4.283-325>.
- [GIK⁺21] C. Guo, T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus. Submission to NIST (version 1.3), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf>.
- [GPP11] J. Guo, T. Peyrin, and A. Poschmann. The PHOTON family of lightweight hash functions. In *Advances in Cryptology (CRYPTO)*, LNCS 6841, pages 222–239. Springer-Verlag, 2011. https://doi.org/10.1007/978-3-642-22792-9_13.
- [Gue09] S. Gueron. Intel’s new AES instructions for enhanced performance and security. In *Fast Software Encryption (FSE)*, LNCS 5665, pages 51–66. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-03317-9_4.
- [HJM07] M. Hell, T. Johansson, and W. Meier. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93, 2007. <https://doi.org/10.1504/IJWMC.2007.013798>.
- [HJM⁺21] M. Hell, T. Johansson, A. Maximov, W. Meier, J. Sönerup, and H. Yoshida. Grain-128AEADv2. Submission to NIST (version 2.0), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf>.
- [HKSS12] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics*, pages 657–660, 2012. <https://doi.org/10.1109/GCCE.2012.6379944>.
- [HV11] A. Hakkala and S. Virtanen. Accelerating cryptographic protocols: A review of theory and technologies. In *Communication Theory, Reliability, and Quality of Service (CTRQ)*, pages 103–109, 2011.
- [Jel19] L. Jellema. Optimizing Ascon on RISC-V. BSc thesis, Radboud University, 2019. https://www.cs.ru.nl/bachelors-theses/2019/Lars_Jellema___4388747___Optimizing_Ascon_on_RISC-V.pdf.
- [KJJ99] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, pages 388–397. Springer-Verlag, 1999. https://doi.org/10.1007/3-540-48405-1_25.

- [Lem20] M. Lemmen. Optimizing Elephant for RISC-V. BSc thesis, Radboud University, 2020. https://www.cs.ru.nl/bachelors-theses/2020/Mauk_Lemmen__4798937__Optimizing_Elephant_for_RISC-V.pdf.
- [MBTM17] K. McKay, L. Bassham, M.S. Turan, and N. Mouha. Report on lightweight cryptography. Technical report, 2017. <https://doi.org/10.6028/NIST.IR.8114>.
- [MNP⁺21] B. Marshall, G.R. Newell, D. Page, M.-J.O. Saarinen, and C. Wolf. The design of scalar AES instruction set extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(1):109–136, 2021. <https://doi.org/10.46586/tches.v2021.i1.109-136>.
- [MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. <https://doi.org/10.1007/978-0-387-38162-6>.
- [MP21] B. Marshall and D. Page. SME: Scalable Masking Extensions. Cryptology ePrint Archive, Paper 2021/1416, 2021. <https://eprint.iacr.org/2021/1416>.
- [MPC00] L. May, L. Penna, and A. Clark. An implementation of bitsliced DES on the Pentium MMXTM processor. In *Australasian Conference on Information Security and Privacy (ACISP)*, LNCS 1841, pages 112–122. Springer-Verlag, 2000. https://doi.org/10.1007/10718964_10.
- [NIK04] K. Nadehara, M. Ikekawa, and I. Kuroda. Extended instructions for the AES cryptography and their efficient implementation. In *Signal Processing Systems (SIPS)*, pages 152–157, 2004. <https://doi.org/10.1109/SIPS.2004.1363041>.
- [NOOS95] E. Nahum, S. O’Malley, H. Orman, and R. Schroepfel. Towards high performance cryptographic software. In *High Performance Communication Subsystems (HPCS)*, pages 69–72, 1995. <https://doi.org/10.1109/HPCS.1995.662009>.
- [rHJM11] M. Ågren, M. Hell, T. Johansson, and W. Meier. Grain128a: a new version of Grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing*, 5(1):48–59, 2011. <https://doi.org/10.1504/IJWMC.2011.044106>.
- [RI16] F. Regazzoni and P. Ienne. Instruction set extensions for secure applications. In *Design, Automation, and Test in Europe (DATE)*, pages 1529–1534, 2016.
- [RPM20] S. Renner, E. Pozzobon, and J. Mottok. A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In *International Conference on Information and Communications Security (ICICS)*, LNCS 12282, pages 495–509. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-61078-4_28.
- [Saa20] M.-J.O. Saarinen. A lightweight ISA extension for AES and SM4. 2020. <https://ascslab.org/conferences/secriscv/program.html>.
- [SCA07] Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. National Institute of Standards and Technology (NIST) Special Publication 800-38D, 2007.

- [SCA16] Oracle SPARC architecture 2011. Technical Report D1.0.0, Oracle Corp., 2016. <https://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf>.
- [SCA18a] Intel 64 and IA-32 architectures – software developer’s manual (volume 1: Basic architecture). Technical Report 325383-067US, Intel Corp., 2018. <http://software.intel.com/en-us/articles/intel-sdm>.
- [SCA18b] Power ISA. Technical Report 2.07 B, IBM, 2018. <https://ibm.ent.box.com/s/jd5w15gz301s5b5dt375mshpq9c3lh4u>.
- [SCA18c] Submission requirements and evaluation criteria for the lightweight cryptography standardization process, 2018. <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf>.
- [SCA19] The RISC-V instruction set manual. Technical Report Volume I: User-Level ISA (version 20190608-Base-Ratified), 2019. <http://riscv.org/specifications>.
- [SCA20] Arm architecture reference manual: Armv8, for Armv8-A architecture profile. Technical report, 2020. https://static.docs.arm.com/ddi0487/fa/DDI0487F_a_armv8_arm.pdf.
- [SCA21] RISC-V bit-manipulation ISA-extensions (version 1.0.0). Technical report, 2021. <https://github.com/riscv/riscv-bitmanip>.
- [SCA22] RISC-V cryptographic extension proposals. Technical Report Volume I: Scalar & Entropy Source Instructions (version 1.0.1), 2022. <https://github.com/riscv/riscv-crypto>.
- [SP21] S. Steinegger and R. Primas. A fast and compact RISC-V accelerator for Ascon and friends. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 12609, pages 53–67. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-68487-7_4.
- [TGSMD20] E. Tehrani, T. Graba, A. Si-Merabet, and J.-L. Danger. RISC-V extension for lightweight cryptography. In *Euromicro Conference on Digital System Design (DSD)*, pages 222–228, 2020. <https://doi.org/10.1109/DSD51259.2020.00045>.
- [TMC⁺21] M.S. Turan, K. McKay, D. Chang, Ç. Çalık, L. Bassham, J. Kang, and J. Kelsey. Status report on the second round of the NIST lightweight cryptography standardization process. Technical report, 2021. <https://doi.org/10.6028/NIST.IR.8369>.
- [TMcc⁺19] M.S. Turan, K. McKay, Ç. Çalık, D. Chang, and L. Bassham. Status report on the first round of the NIST lightweight cryptography standardization process. Technical report, 2019. <https://doi.org/10.6028/NIST.IR.8268>.
- [Wat16] A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California at Berkeley, 2016. <https://people.eecs.berkeley.edu/~krste/papers/ECS-2016-1.pdf>.
- [WH21] H. Wu and T. Huang. TinyJAMBU. Submission to NIST (version 2.0), 2021. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf>.

A Additional ISE design detail: ASCON

A.1 Additional notation

Define the look-up tables

```
ROT_0 = { 19, 61, 1, 10, 7 }
ROT_1 = { 28, 39, 6, 17, 41 }
```

A.2 ISE definition

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00	imm	rs2	rs1	111	rd	0101011											ascon.sigma.lo															
01	imm	rs2	rs1	111	rd	0101011											ascon.sigma.hi															

- `ascon.sigma.lo rd, rs1, rs2, imm`

```
1 | x_hi   ← GPR[rs2]
2 | x_lo   ← GPR[rs1]
3 | x      ← x_hi || x_lo
4 | r      ← x ^ ( x >>> ROT_0[ imm ] ) ^ ( x >>> ROT_1[ imm ] )
5 | GPR[rd] ← r_{31.. 0}
```

- `ascon.sigma.hi rd, rs1, rs2, imm`

```
1 | x_hi   ← GPR[rs2]
2 | x_lo   ← GPR[rs1]
3 | x      ← x_hi || x_lo
4 | r      ← x ^ ( x >>> ROT_0[ imm ] ) ^ ( x >>> ROT_1[ imm ] )
5 | GPR[rd] ← r_{63..32}
```

B Additional ISE design detail: Elephant

B.1 Additional notation

Let `SBOX` denote the 4-bit Spongent S-box per [BKL⁺13]. Define the functions

```
SWAPMOVE32 (x, m,n) {
  t ← x ^ ( x >> n )
  t ← t & m
  t ← t ^ ( t << n )
  x ← t ^ x

  return x
}

SWAPMOVE32_X(x,y,m,n) {
  t ← y ^ ( x >> n )
  t ← t & m
  x ← x ^ ( t << n )

  return x
}

SWAPMOVE32_Y(x,y,m,n) {
  t ← y ^ ( x >> n )
  t ← t & m
  y ← y ^ ( t )

  return y
}
```

i.e., variants of `SWAPMOVE` [MPC00, Section 3.1].

B.2 ISE definition

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000	imm	rs2	rs1	111	rd	0001011	elephant.pstep.x
0001	imm	rs2	rs1	111	rd	0001011	elephant.pstep.y
0010	imm	00000	rs1	110	rd	0001011	elephant.sstep

• elephant.pstep.x rd, rs1, rs2, imm

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 |
4 | if      ( imm == 0 ) {
5 |   r ← SWAPMOVE32_X( x, y, 0x000000FF, 8 )
6 | }
7 | else if ( imm == 1 ) {
8 |   r ← SWAPMOVE32_X( x, y, 0x000000FF, 16 )
9 | }
10 | else if ( imm == 2 ) {
11 |   r ← SWAPMOVE32_X( x, y, 0x000000FF, 24 )
12 | }
13 | else if ( imm == 3 ) {
14 |   r ← SWAPMOVE32_X( x, y, 0x0000FF00, 8 )
15 | }
16 | else if ( imm == 4 ) {
17 |   r ← SWAPMOVE32_X( x, y, 0x000000FF, 24 ) >>> 24
18 | }
19 | else if ( imm == 5 ) {
20 |   r ← SWAPMOVE32_X( x, y, 0x0000FF00, 16 ) >>> 16
21 | }
22 | else if ( imm == 6 ) {
23 |   r ← SWAPMOVE32_X( x, y, 0x00FF0000, 8 ) >>> 8
24 | }
25 |
26 | GPR[rd] ← r

```

• elephant.pstep.y rd, rs1, rs2, imm

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 |
4 | if      ( imm == 0 ) {
5 |   r ← SWAPMOVE32_Y( x, y, 0x000000FF, 8 )
6 | }
7 | else if ( imm == 1 ) {
8 |   r ← SWAPMOVE32_Y( x, y, 0x000000FF, 16 )
9 | }
10 | else if ( imm == 2 ) {
11 |   r ← SWAPMOVE32_Y( x, y, 0x000000FF, 24 )
12 | }
13 | else if ( imm == 3 ) {
14 |   r ← SWAPMOVE32_Y( x, y, 0x0000FF00, 8 )
15 | }
16 | else if ( imm == 4 ) {
17 |   r ← SWAPMOVE32_Y( x, y, 0x000000FF, 24 )
18 | }
19 | else if ( imm == 5 ) {
20 |   r ← SWAPMOVE32_Y( x, y, 0x0000FF00, 16 )
21 | }
22 | else if ( imm == 6 ) {
23 |   r ← SWAPMOVE32_Y( x, y, 0x00FF0000, 8 )
24 | }
25 |
26 | GPR[rd] ← r

```

• elephant.sstep rd, rs1

```

1 | x      ← GPR[rs1]
2 |
3 | r      ← SBOX[ x_{31..28} ] || SBOX[ x_{27..24} ] ||
4 |         SBOX[ x_{23..20} ] || SBOX[ x_{19..16} ] ||
5 |         SBOX[ x_{15..12} ] || SBOX[ x_{11.. 8} ] ||
6 |         SBOX[ x_{ 7.. 4} ] || SBOX[ x_{ 3.. 0} ]
7 |

```

```

8 | r      ← SWAPMOVE32( r, 0x0A0A0A0A, 3 )
9 | r      ← SWAPMOVE32( r, 0x00CC00CC, 6 )
10 | r     ← SWAPMOVE32( r, 0x0000F0F0, 12 )
11 | r     ← SWAPMOVE32( r, 0x0000FF00, 8 )
12 |
13 | GPR[rd] ← r

```

C Additional ISE design detail: GIFT-COFB

C.1 Additional notation

Define the function

```

SWAPMOVE32(x,m,n) {
  t ← x ^ ( x >> n )
  t ← t & m
  t ← t ^ ( t << n )
  x ← t ^ x
  return x
}

```

i.e., a variant of SWAPMOVE [MPC00, Section 3.1].

C.2 ISE definition (for fix-sliced implementation)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

00	imm	rs2	rs1	111	rd	0001011	gift.swapmove
01	imm	00000	rs1	110	rd	0001011	gift.ori.n
10	imm	00000	rs1	110	rd	0001011	gift.ori.b
11	imm	00000	rs1	110	rd	0001011	gift.ori.h
00	imm	00000	rs1	110	rd	0101011	gift.key.reorg
01	00000	00000	rs1	110	rd	0101011	gift.key.upstd
10	imm	00000	rs1	110	rd	0101011	gift.key.updfix

- gift.swapmove rd, rs1, rs2, imm

```

1 | x      ← GPR[rs1]
2 | m      ← GPR[rs2]
3 | r      ← SWAPMOVE32( x, m, imm )
4 | GPR[rd] ← r

```

- gift.ori.n rd, rs1, imm

```

1 | x_7    ← GPR[rs1]_{31..28}
2 | x_6    ← GPR[rs1]_{27..24}
3 | x_5    ← GPR[rs1]_{23..20}
4 | x_4    ← GPR[rs1]_{19..16}
5 | x_3    ← GPR[rs1]_{15..12}
6 | x_2    ← GPR[rs1]_{11.. 8}
7 | x_1    ← GPR[rs1]_{ 7.. 4}
8 | x_0    ← GPR[rs1]_{ 3.. 0}
9 | r      ← ( x_7 >>> imm ) || ( x_6 >>> imm ) ||
10 |        ( x_5 >>> imm ) || ( x_4 >>> imm ) ||
11 |        ( x_3 >>> imm ) || ( x_2 >>> imm ) ||
12 |        ( x_1 >>> imm ) || ( x_0 >>> imm )
13 | GPR[rd] ← r

```

- gift.ori.b rd, rs1, imm

```

1 | x_3    ← GPR[rs1]_{31..24}
2 | x_2    ← GPR[rs1]_{23..16}
3 | x_1    ← GPR[rs1]_{15.. 8}
4 | x_0    ← GPR[rs1]_{ 7.. 0}
5 | r      ← ( x_3 >>> imm ) || ( x_2 >>> imm ) ||
6 |        ( x_1 >>> imm ) || ( x_0 >>> imm )
7 | GPR[rd] ← r

```

- **gift.rori.h rd, rs1, imm**

```

1 | x_1    ← GPR[rs1]_{31..16}
2 | x_0    ← GPR[rs1]_{15.. 0}
3 | r      ← ( x_1 >>> imm ) || ( x_0 >>> imm )
4 | GPR[rd] ← r

```

- **gift.key.reorg rd, rs1, imm**

```

1 | x      ← GPR[rs1]
2 |
3 | if     ( imm == 0 ) {
4 |   r ← SWAPMOVE32( x, 0x00550055, 9 )
5 |   r ← SWAPMOVE32( r, 0x00003333, 18 )
6 |   r ← SWAPMOVE32( r, 0x000F000F, 12 )
7 |   r ← SWAPMOVE32( r, 0x000000FF, 24 )
8 | }
9 | else if ( imm == 1 ) {
10 |  r ← SWAPMOVE32( x, 0x11111111, 3 )
11 |  r ← SWAPMOVE32( r, 0x03030303, 6 )
12 |  r ← SWAPMOVE32( r, 0x000F000F, 12 )
13 |  r ← SWAPMOVE32( r, 0x000000FF, 24 )
14 |  else if ( imm == 2 ) {
15 |    r ← SWAPMOVE32( x, 0x0000AAAA, 15 )
16 |    r ← SWAPMOVE32( r, 0x00003333, 18 )
17 |    r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
18 |    r ← SWAPMOVE32( r, 0x000000FF, 24 )
19 |  else if ( imm == 3 ) {
20 |    r ← SWAPMOVE32( x, 0x0A0A0A0A, 3 )
21 |    r ← SWAPMOVE32( r, 0x00CC00CC, 6 )
22 |    r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
23 |    r ← SWAPMOVE32( r, 0x000000FF, 24 )
24 |  }
25 | }
26 | GPR[rd] ← r

```

- **gift.key.updstd rd, rs1**

```

1 | x      ← GPR[rs1]
2 |
3 | r      ← ( ( x >> 12 ) & 0x0000000F )
4 | r ← r | ( ( x & 0x00000FFF ) << 4 )
5 | r      ← r | ( ( x >> 2 ) & 0x3FFF0000 )
6 | r      ← r | ( ( x & 0x00030000 ) << 14 )
7 |
8 | GPR[rd] ← r

```

- **gift.key.updfix rd, rs1, imm**

```

1 | x      ← GPR[rs1]
2 |
3 | if     ( imm == 0 ) {
4 |   r ← SWAPMOVE32( x, 0x00003333, 16 )
5 |   r ← SWAPMOVE32( r, 0x55554444, 1 )
6 | }
7 | else if ( imm == 1 ) {
8 |   r ← ( ( x & 0x33333333 ) >>> 24 )
9 |   r ← r | ( ( x & 0xCCCCCCCC ) >>> 16 )
10 |  r ← SWAPMOVE32( r, 0x55551100, 1 )
11 | }
12 | else if ( imm == 2 ) {
13 |   r ← ( ( x >> 4 ) & 0x0F000F00 ) | ( ( x & 0x0F000F00 ) << 4 )
14 |   r ← r | ( ( x >> 6 ) & 0x00030003 ) | ( ( x & 0x003F003F ) << 2 )
15 | }
16 | else if ( imm == 3 ) {
17 |   r ← ( ( x >> 6 ) & 0x03000300 ) | ( ( x & 0x3F003F00 ) << 2 )
18 |   r ← r | ( ( x >> 5 ) & 0x00070007 ) | ( ( x & 0x001F001F ) << 3 )
19 | }
20 | else if ( imm == 4 ) {
21 |   r ← ( ( x & 0xAAAAAAAA ) >>> 24 )
22 |   r ← r | ( ( x & 0x55555555 ) >>> 16 )
23 | }
24 | else if ( imm == 5 ) {
25 |   r ← ( ( x & 0x55555555 ) >>> 24 )
26 |   r ← r | ( ( x & 0xAAAAAAAA ) >>> 20 )
27 | }
28 | else if ( imm == 6 ) {
29 |   r ← ( ( x >> 2 ) & 0x03030303 ) | ( ( x & 0x03030303 ) << 2 )

```

```

30 | r ← r | ( ( x >> 1 ) & 0x70707070 ) | ( ( x & 0x10101010 ) << 3 )
31 | }
32 | else if ( imm == 7 ) {
33 |   r ← ( ( x >> 18 ) & 0x00003030 ) | ( ( x & 0x01010101 ) << 3 )
34 |   r ← r | ( ( x >> 14 ) & 0x0000C0C0 ) | ( ( x & 0x0000E0E0 ) << 15 )
35 |   r ← r | ( ( x >> 1 ) & 0x07070707 ) | ( ( x & 0x00001010 ) << 19 )
36 | }
37 | else if ( imm == 8 ) {
38 |   r ← ( ( x >> 4 ) & 0x0FFF0000 ) | ( ( x & 0x000F0000 ) << 12 )
39 |   r ← r | ( ( x >> 8 ) & 0x000000FF ) | ( ( x & 0x000000FF ) << 8 )
40 | }
41 | else if ( imm == 9 ) {
42 |   r ← ( ( x >> 6 ) & 0x03FF0000 ) | ( ( x & 0x003F0000 ) << 10 )
43 |   r ← r | ( ( x >> 4 ) & 0x00000FFF ) | ( ( x & 0x0000000F ) << 12 )
44 | }
45 |
46 | GPR[rd] ← r

```

C.3 ISE definition (for bit-sliced implementation)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01	00000	00000	rs1	110	rd	0101011	gift.key.updstd																								
11	imm	00000	rs1	110	rd	0101011	gift.permbits.step																								

- gift.key.updstd rd, rs1

```

1 | x ← GPR[rs1]
2 |
3 | r ← ( ( x >> 12 ) & 0x0000000F )
4 | r ← r | ( ( x & 0x00000FFF ) << 4 )
5 | r ← r | ( ( x >> 2 ) & 0x3FFF0000 )
6 | r ← r | ( ( x & 0x00030000 ) << 14 )
7 |
8 | GPR[rd] ← r

```

- gift.permbits.step rd, rs1, imm

```

1 | x ← GPR[rs1]
2 |
3 | r ← SWAPMOVE32( x, 0x0A0A0A0A, 3 )
4 | r ← SWAPMOVE32( r, 0x00CC00CC, 6 )
5 | r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
6 | r ← SWAPMOVE32( r, 0x000000FF, 24 )
7 | r ← r >>> imm
8 |
9 | GPR[rd] ← r

```

D Additional ISE design detail: Grain-128AEADv2

D.1 ISE definition

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00	imm	rs2	rs1	111	rd	0001011	grain.extr																								
0000	000	rs2	rs1	111	rd	0101011	grain.fln0																								
0001	000	rs2	rs1	111	rd	0101011	grain.fln2																								
0010	000	rs2	rs1	111	rd	0101011	grain.gnn0																								
0011	000	rs2	rs1	111	rd	0101011	grain.gnn1																								
0100	000	rs2	rs1	111	rd	0101011	grain.gnn2																								
0101	000	rs2	rs1	111	rd	0101011	grain.hnn0																								
0110	000	rs2	rs1	111	rd	0101011	grain.hnn1																								
0111	000	rs2	rs1	111	rd	0101011	grain.hnn2																								
1000	000	rs2	rs1	111	rd	0101011	grain.hln0																								

- **grain.extr rd, rs1, rs2, imm**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← x >> imm
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.fln0 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x_lo ) ^ ( x >> 7 )
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.fln2 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x_hi ) ^ ( x >> 6 ) ^ ( x >> 17 )
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.gnn0 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x_lo ) ^ ( x >> 26 ) ^ ( ( x >> 11 ) & ( x >> 13 ) ) ^
5 |       ( ( x >> 17 ) & ( x >> 18 ) ) ^
6 |       ( ( x >> 22 ) & ( x >> 24 ) & ( x >> 25 ) )
7 | GPR[rd] ← r_{31.. 0}

```

- **grain.gnn1 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x >> 24 ) ^ ( ( x >> 8 ) & ( x >> 16 ) )
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.gnn2 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x_hi ) ^ ( x >> 27 ) ^ ( ( x >> 4 ) & ( x >> 20 ) ) ^
5 |       ( ( x >> 24 ) & ( x >> 28 ) & ( x >> 29 ) & ( x >> 31 ) ) ^
6 |       ( ( x >> 6 ) & ( x >> 14 ) & ( x >> 18 ) )
7 | GPR[rd] ← r_{31.. 0}

```

- **grain.hnn0 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x >> 2 ) ^ ( x >> 15 )
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.hnn1 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x >> 4 ) ^ ( x >> 13 )
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.hnn2 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x     ← x_hi || x_lo
4 | r     ← ( x_lo ) ^ ( x >> 9 ) ^ ( x >> 25 )
5 | GPR[rd] ← r_{31.. 0}

```

- **grain.hln0 rd, rs1, rs2**

```

1 | x_hi   ← GPR[rs1]
2 | x_lo   ← GPR[rs2]
3 | x      ← x_hi || x_lo
4 | r      ← ( x >> 13 ) & ( x >> 20 )
5 | GPR[rd] ← r_{31.. 0}

```

E Additional ISE design detail: ISAP

The ISE for ISAP is the same as for ASCON, which can be found in Appendix A.

F Additional ISE design detail: PHOTON-Beetle

F.1 Additional notation

Let SBOX denote the 4-bit PHOTON S-box per [GPP11], and GF2N_MUL denote multiplication in the PHOTON finite field. Define a look-up table

```

M = { { 0x2, 0x4, 0x2, 0xB, 0x2, 0x8, 0x5, 0x6 },
      { 0xC, 0x9, 0x8, 0xD, 0x7, 0x7, 0x5, 0x2 },
      { 0x4, 0x4, 0xD, 0xD, 0x9, 0x4, 0xD, 0x9 },
      { 0x1, 0x6, 0x5, 0x1, 0xC, 0xD, 0xF, 0xE },
      { 0xF, 0xC, 0x9, 0xD, 0xE, 0x5, 0xE, 0xD },
      { 0x9, 0xE, 0x5, 0xF, 0x4, 0xC, 0x9, 0x6 },
      { 0xC, 0x2, 0x2, 0xA, 0x3, 0x1, 0x1, 0xE },
      { 0xF, 0x1, 0xD, 0xA, 0x5, 0xA, 0x2, 0x3 } }

```

to configure the MixColumnsSerial round function.

F.2 ISE definition

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000	imm	rs2	rs1	111	rd	1011011	photon.step
------	-----	-----	-----	-----	----	---------	-------------

- `photon.step rd, rs1, rs2, imm`

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 |
4 | t      ← SBOX[ ( y >> ( 4 * imm ) ) & 0xF ]
5 | r      ← 0
6 |
7 | for( int i = 0; i < 8; i++ ) {
8 |   r ← r | ( ( GF2N_MUL( M[ i ][ imm ], t ) ) << ( 4 * i ) )
9 | }
10 |
11 | r      ← r ^ x
12 |
13 | GPR[rd] ← r

```

G Additional ISE design detail: Romulus

G.1 Additional notation

Let SBOX denote the 8-bit Skinny S-box per [BJK⁺16]. Define the functions

```

RC_LFSR_FWD( x ) {
  return x_4 || x_3 || x_2 || x_1 || x_0 || ( x_5 ^ x_4 ^ 1 )
}

RC_LFSR_REV( x ) {
  return ( x_5 ^ x_0 ^ 1 ) || x_4 || x_3 || x_2 || x_1 || x_0
}

```



```

TK2_LFSR_FWD( x ) {
  return x_6 || x_5 || x_4 || x_3 || x_2 || x_1 || x_0 || ( x_5 ^ x_7 )
}

TK2_LFSR_REV( x ) {
  return ( x_6 ^ x_0 ) || x_6 || x_5 || x_4 || x_3 || x_2 || x_1 || x_0
}

TK3_LFSR_FWD( x ) {
  return ( x_6 ^ x_0 ) || x_7 || x_6 || x_5 || x_4 || x_3 || x_2 || x_1
}

TK3_LFSR_REV( x ) {
  return x_6 || x_5 || x_4 || x_3 || x_2 || x_1 || x_0 || ( x_5 ^ x_7 )
}

```

Define the functions

```

SWAPMOVE32 (x, m,n) {
  t ← x ^ ( x >> n )
  t ← t & m
  t ← t ^ ( t << n )
  x ← t ^ x
}

return x
}

SWAPMOVE32_X(x,y,m,n) {
  t ← y ^ ( x >> n )
  t ← t & m
  x ← x ^ ( t << n )
}

return x
}

SWAPMOVE32_Y(x,y,m,n) {
  t ← y ^ ( x >> n )
  t ← t & m
  y ← y ^ ( t << n )
}

return y
}

```

i.e., variants of SWAPMOVE [MPC00, Section 3.1].

G.2 ISE definition (for table-based implementation)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000	000	00000	rs1	110	rd	0001011	romulus.rc.upd.enc																								
0010	000	rs2	rs1	111	rd	0001011	romulus.rc.use.enc.0																								
0011	000	rs2	rs1	111	rd	0001011	romulus.rc.use.enc.1																								
0001	imm	rs2	rs1	111	rd	0101011	romulus.tk.upd.enc.0																								
0010	imm	rs2	rs1	111	rd	0101011	romulus.tk.upd.enc.1																								
0000	imm	rs2	rs1	111	rd	1011011	romulus.rstep.enc																								

- romulus.rc.upd.enc rd, rs1

```

1 | x      ← GPR[rs1]
2 | r      ← LFSR_RC( x )
3 | GPR[rd] ← r

```

- romulus.rc.use.enc.0 rd, rs1, rs2

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 | r      ← y ^ x_{3..0}
4 | GPR[rd] ← r

```

- romulus.rc.use.enc.1 rd, rs1, rs2

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 | r      ← y ^ x_{6..4}
4 | GPR[rd] ← r

```

• **romulus.tk.upd.enc.0 rd, rs1, rs2, imm**

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 |
4 | if     ( imm == 1 ) {
5 |   r ←   y_{15.. 8}  ||          x_{ 7.. 0}  ||
6 |         y_{31..24}  ||          x_{15.. 8}
7 | }
8 | else if( imm == 2 ) {
9 |   r ← LFSR_TK2( y_{15.. 8} ) || LFSR_TK2( x_{ 7.. 0} ) ||
10 |      LFSR_TK2( y_{31..24} ) || LFSR_TK2( x_{15.. 8} )
11 | }
12 | else if( imm == 3 ) {
13 |   r ← LFSR_TK3( y_{15.. 8} ) || LFSR_TK3( x_{ 7.. 0} ) ||
14 |      LFSR_TK3( y_{31..24} ) || LFSR_TK3( x_{15.. 8} )
15 | }
16 |
17 | GPR[rd] ← r

```

• **romulus.tk.upd.enc.1 rd, rs1, rs2, imm**

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 |
4 | if     ( imm == 1 ) {
5 |   r ←   x_{31..24}  ||          y_{ 7.. 0}  ||
6 |         y_{23..16}  ||          x_{23..16}
7 | }
8 | else if( imm == 2 ) {
9 |   r ← LFSR_TK2( x_{31..24} ) || LFSR_TK2( y_{ 7.. 0} ) ||
10 |      LFSR_TK2( y_{23..16} ) || LFSR_TK2( x_{23..16} )
11 | }
12 | else if( imm == 3 ) {
13 |   r ← LFSR_TK3( x_{31..24} ) || LFSR_TK3( y_{ 7.. 0} ) ||
14 |      LFSR_TK3( y_{23..16} ) || LFSR_TK3( x_{23..16} )
15 | }
16 |
17 | GPR[rd] ← r

```

• **romulus.rstep.enc rd, rs1, rs2, imm**

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 |
4 | if     ( imm == 2 ) {
5 |   y ← 2
6 | }
7 | else if( imm == 3 ) {
8 |   y ← 0
9 | }
10 |
11 | t      ← SBOX[ x_{31..24} ] || SBOX[ x_{23..16} ] ||
12 |         SBOX[ x_{15.. 8} ] || SBOX[ x_{ 7.. 0} ]
13 |
14 | t      ← t ^ y
15 |
16 | if     ( imm == 0 ) {
17 |   r ← t <<< 0
18 | }
19 | else if( imm == 1 ) {
20 |   r ← t <<< 8
21 | }
22 | else if( imm == 2 ) {
23 |   r ← t <<< 16
24 | }
25 | else if( imm == 3 ) {
26 |   r ← t <<< 24
27 | }
28 |
29 | GPR[rd] ← r

```

G.3 ISE definition (for fix-sliced implementation)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000	imm	00000	rs1	110	rd	1111011	romulus.mixcolumns
0001	imm	rs2	rs1	111	rd	1111011	romulus.swapmove.x
0010	imm	rs2	rs1	111	rd	1111011	romulus.swapmove.y
0100	imm	00000	rs1	110	rd	0101011	romulus.permtk
0101	imm	00000	rs1	110	rd	0101011	romulus.tkupd.0
0110	imm	00000	rs1	110	rd	0101011	romulus.tkupd.1
0100	000	rs2	rs1	111	rd	0001011	romulus.lfsr2
0101	000	rs2	rs1	111	rd	0001011	romulus.lfsr3

- romulus.mixcolumns rd, rs1, imm

```

1  x      ← GPR[rs1]
2
3  if      ( imm == 0 ) {
4    r ← x ^ ( ( ( x >>> 24 ) & 0x0C0C0C0C ) >>> 30 )
5    r ← r ^ ( ( ( r >>> 16 ) & 0xC0C0C0C0 ) >>> 4 )
6    r ← r ^ ( ( ( r >>> 8 ) & 0x0C0C0C0C ) >>> 2 )
7  }
8  else if ( imm == 1 ) {
9    r ← x ^ ( ( ( x >>> 16 ) & 0x30303030 ) >>> 30 )
10   r ← r ^ ( ( ( r ) & 0x03030303 ) >>> 28 )
11   r ← r ^ ( ( ( r >>> 16 ) & 0x30303030 ) >>> 2 )
12 }
13 else if ( imm == 2 ) {
14   r ← x ^ ( ( ( x >>> 8 ) & 0xC0C0C0C0 ) >>> 6 )
15   r ← r ^ ( ( ( r >>> 16 ) & 0x0C0C0C0C ) >>> 28 )
16   r ← r ^ ( ( ( r >>> 24 ) & 0xC0C0C0C0 ) >>> 2 )
17 }
18 else if ( imm == 3 ) {
19   r ← x ^ ( ( ( x ) & 0x03030303 ) >>> 30 )
20   r ← r ^ ( ( ( r ) & 0x30303030 ) >>> 4 )
21   r ← r ^ ( ( ( r ) & 0x03030303 ) >>> 26 )
22 }
23
24 GPR[rd] ← r

```

- romulus.swapmove.x rd, rs1, rs2, imm

```

1  x      ← GPR[rs1]
2  y      ← GPR[rs2]
3
4  if      ( imm == 0 ) {
5    r ← SWAPMOVE32_X( x, y, 0x55555555, 1 )
6  }
7  else if ( imm == 1 ) {
8    r ← SWAPMOVE32_X( x, y, 0x30303030, 2 )
9  }
10 else if ( imm == 2 ) {
11   r ← SWAPMOVE32_X( x, y, 0x0C0C0C0C, 4 )
12 }
13 else if ( imm == 3 ) {
14   r ← SWAPMOVE32_X( x, y, 0x03030303, 6 )
15 }
16 else if ( imm == 4 ) {
17   r ← SWAPMOVE32_X( x, y, 0x0C0C0C0C, 2 )
18 }
19 else if ( imm == 5 ) {
20   r ← SWAPMOVE32_X( x, y, 0x03030303, 4 )
21 }
22 else if ( imm == 6 ) {
23   r ← SWAPMOVE32_X( x, y, 0x03030303, 2 )
24 }
25 else if ( imm == 7 ) {
26   r ← SWAPMOVE32 ( x, 0x0A0A0A0A, 3 )
27 }
28
29 GPR[rd] ← r

```

• **romulus.swapmove.y rd, rs1, rs2, imm**

```

1  x      ← GPR[rs1]
2  y      ← GPR[rs2]
3
4  if      ( imm == 0 ) {
5    r ← SWAPMOVE32_Y( x, y, 0x55555555, 1 )
6  }
7  else if ( imm == 1 ) {
8    r ← SWAPMOVE32_Y( x, y, 0x30303030, 2 )
9  }
10 else if ( imm == 2 ) {
11   r ← SWAPMOVE32_Y( x, y, 0x0C0C0C0C, 4 )
12 }
13 else if ( imm == 3 ) {
14   r ← SWAPMOVE32_Y( x, y, 0x03030303, 6 )
15 }
16 else if ( imm == 4 ) {
17   r ← SWAPMOVE32_Y( x, y, 0x0C0C0C0C, 2 )
18 }
19 else if ( imm == 5 ) {
20   r ← SWAPMOVE32_Y( x, y, 0x03030303, 4 )
21 }
22 else if ( imm == 6 ) {
23   r ← SWAPMOVE32_Y( x, y, 0x03030303, 2 )
24 }
25
26 GPR[rd] ← r

```

• **romulus.permtk rd, rs1, imm**

```

1  x      ← GPR[rs1]
2
3  if      ( imm == 0 ) {
4    r ← ( ( ( x >>> 14 ) & 0xCC00CC00 )      )
5    r ← r | ( ( ( x      ) & 0x000000FF ) << 16 )
6    r ← r | ( ( ( x      ) & 0xCC000000 ) >> 2 )
7    r ← r | ( ( ( x      ) & 0x0033CC00 ) >> 8 )
8    r ← r | ( ( ( x      ) & 0x00CC0000 ) >> 18 )
9  }
10 else if ( imm == 1 ) {
11   r ← ( ( ( x >>> 22 ) & 0xCC0000CC )      )
12   r ← r | ( ( ( x >>> 16 ) & 0x3300CC00 )      )
13   r ← r | ( ( ( x >>> 24 ) & 0x00CC3300 )      )
14   r ← r | ( ( ( x      ) & 0x00CC00CC ) >> 2 )
15 }
16 else if ( imm == 2 ) {
17   r ← ( ( ( x >>> 6 ) & 0xCCCC0000 )      )
18   r ← r | ( ( ( x >>> 24 ) & 0x330000CC )      )
19   r ← r | ( ( ( x >>> 10 ) & 0x00003333 )      )
20   r ← r | ( ( ( x      ) & 0x000000CC ) << 14 )
21   r ← r | ( ( ( x      ) & 0x00003300 ) << 2 )
22 }
23 else if ( imm == 3 ) {
24   r ← ( ( ( x >>> 24 ) & 0xCC000033 )      )
25   r ← r | ( ( ( x >>> 8 ) & 0x33CC0000 )      )
26   r ← r | ( ( ( x >>> 26 ) & 0x00333300 )      )
27   r ← r | ( ( ( x      ) & 0x00333300 ) >> 6 )
28 }
29 else if ( imm == 4 ) {
30   r ← ( ( ( x >>> 8 ) & 0xCC330000 )      )
31   r ← r | ( ( ( x >>> 26 ) & 0x33000033 )      )
32   r ← r | ( ( ( x >>> 22 ) & 0x00CCCC00 )      )
33   r ← r | ( ( ( x      ) & 0x00330000 ) >> 14 )
34   r ← r | ( ( ( x      ) & 0x0000CC00 ) >> 2 )
35 }
36 else if ( imm == 5 ) {
37   r ← ( ( ( x >>> 8 ) & 0x0000CC33 )      )
38   r ← r | ( ( ( x >>> 30 ) & 0x00CC00CC )      )
39   r ← r | ( ( ( x >>> 10 ) & 0x33330000 )      )
40   r ← r | ( ( ( x >>> 16 ) & 0xCC003300 )      )
41 }
42 else if ( imm == 6 ) {
43   r ← ( ( ( x >>> 24 ) & 0x0033CC00 )      )
44   r ← r | ( ( ( x >>> 14 ) & 0x00CC0000 )      )
45   r ← r | ( ( ( x >>> 30 ) & 0xCC000000 )      )
46   r ← r | ( ( ( x >>> 16 ) & 0x000000FF )      )
47   r ← r | ( ( ( x >>> 18 ) & 0x33003300 )      )
48 }

```

```
49 |
50 | GPR[rd] ← r
```

• **romulus.tkupd.0 rd, rs1, imm**

```
1 | x      ← GPR[rs1]
2 |
3 | if    ( imm == 0 ) {
4 |   r ←  ( ( x >>> 26 ) & 0xC3C3C3C3 )
5 | }
6 | else if ( imm == 1 ) {
7 |   r ←  ( ( x >>> 16 ) & 0xF0F0F0F0 )
8 | }
9 | else if ( imm == 2 ) {
10 |  r ←  ( ( x >>> 10 ) & 0xC3C3C3C3 )
11 | }
12 |
13 | GPR[rd] ← r
```

• **romulus.tkupd.1 rd, rs1, imm**

```
1 | x      ← GPR[rs1]
2 |
3 | if    ( imm == 0 ) {
4 |   r ←  ( ( x >>> 28 ) & 0x03030303 )
5 |   r ← r | ( ( x >>> 12 ) & 0x0C0C0C0C )
6 | }
7 | else if ( imm == 1 ) {
8 |   r ←  ( ( x >>> 14 ) & 0x30303030 )
9 |   r ← r | ( ( x >>> 6 ) & 0x0C0C0C0C )
10 | }
11 | else if ( imm == 2 ) {
12 |   r ←  ( ( x >>> 12 ) & 0x03030303 )
13 |   r ← r | ( ( x >>> 28 ) & 0x0C0C0C0C )
14 | }
15 | else if ( imm == 3 ) {
16 |   r ←  ( ( x >>> 30 ) & 0x30303030 )
17 |   r ← r | ( ( x >>> 22 ) & 0x0C0C0C0C )
18 | }
19 |
20 | GPR[rd] ← r
```

• **romulus.lfsr2 rd, rs1, rs2**

```
1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 | r      ← x ^ ( ( y & 0xAAAAAAAA ) )
4 | r      ← ( ( ( r ) & 0xAAAAAAAA ) >> 1 ) |
5 |        ( ( ( r << 1 ) & 0xAAAAAAAA ) )
6 | GPR[rd] ← r
```

• **romulus.lfsr3 rd, rs1, rs2**

```
1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 | r      ← x ^ ( ( y & 0xAAAAAAAA ) >> 1 )
4 | r      ← ( ( ( r ) & 0xAAAAAAAA ) >> 1 ) |
5 |        ( ( ( r << 1 ) & 0xAAAAAAAA ) )
6 | GPR[rd] ← r
```

H Additional ISE design detail: SPARKLE

H.1 Additional notation

Define the look-up tables

```
ROT_0 = { 31, 17, 0, 24 }
ROT_1 = { 24, 17, 31, 16 }
```

```
RCON = { 0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738,
         0xBB1185EB, 0x4F7C7B57, 0xCFBFA1C8, 0xC2B3293D }
```

Define the function

```

ELL( x ) {
    return ( x ^ ( x << 16 ) ) >>> 16
}

```

H.2 ISE definition

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0							
0000010		rs2	rs1	111	rd	1111011	sparkle.ell
0000	imm	rs2	rs1	110	rd	1011011	sparkle.rcon
1000	imm	rs2	rs1	111	rd	1011011	sparkle.whole.enci.x
1001	imm	rs2	rs1	111	rd	1011011	sparkle.whole.enci.y

- **sparkle.ell rd, rs1, rs2**

```

1 | x      ← GPR[rs1]
2 | y      ← GPR[rs2]
3 | r      ← ELL( x ^ y )
4 | GPR[rd] ← r

```

- **sparkle.rcon rd, rs1, imm**

```

1 | x      ← GPR[rs1]
2 | r      ← x ^ RCON[imm]
3 | GPR[rd] ← r

```

- **sparkle.whole.enci.x rd, rs1, rs2, imm**

```

1 | xi     ← GPR[rs1]
2 | yi     ← GPR[rs2]
3 | ci     ← RCON[imm]
4 | xi     ← xi + ( yi >>> 31 )
5 | yi     ← yi ^ ( xi >>> 24 )
6 | xi     ← xi ^ ci
7 | xi     ← xi + ( yi >>> 17 )
8 | yi     ← yi ^ ( xi >>> 17 )
9 | xi     ← xi ^ ci
10 | xi    ← xi + ( yi >>> 0 )
11 | yi    ← yi ^ ( xi >>> 31 )
12 | xi    ← xi ^ ci
13 | xi    ← xi + ( yi >>> 24 )
14 | yi    ← yi ^ ( xi >>> 16 )
15 | xi    ← xi ^ ci
16 | GPR[rd] ← xi

```

- **sparkle.whole.enci.y rd, rs1, rs2, imm**

```

1 | xi     ← GPR[rs1]
2 | yi     ← GPR[rs2]
3 | ci     ← RCON[imm]
4 | xi     ← xi + ( yi >>> 31 )
5 | yi     ← yi ^ ( xi >>> 24 )
6 | xi     ← xi ^ ci
7 | xi     ← xi + ( yi >>> 17 )
8 | yi     ← yi ^ ( xi >>> 17 )
9 | xi     ← xi ^ ci
10 | xi    ← xi + ( yi >>> 0 )
11 | yi    ← yi ^ ( xi >>> 31 )
12 | xi    ← xi ^ ci
13 | xi    ← xi + ( yi >>> 24 )
14 | yi    ← yi ^ ( xi >>> 16 )
15 | xi    ← xi ^ ci
16 | GPR[rd] ← yi

```


I Additional ISE design detail: TinyJAMBU

I.1 ISE definition

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

00	00000	rs2	rs1	111	rd	1111011	jambu.fsr.15
00	00001	rs2	rs1	111	rd	1111011	jambu.fsr.6
00	00010	rs2	rs1	111	rd	1111011	jambu.fsr.21
00	00011	rs2	rs1	111	rd	1111011	jambu.fsr.27

- **jambu.fsr.15 rd, rs1, rs2**

```

1 | x_hi ← GPR[rs2]
2 | x_lo ← GPR[rs1]
3 | r ← ( x_hi || x_lo ) >>> 15
4 | GPR[rd] ← r_{31.. 0}

```

- **jambu.fsr.6 rd, rs1, rs2**

```

1 | x_hi ← GPR[rs2]
2 | x_lo ← GPR[rs1]
3 | r ← ( x_hi || x_lo ) >>> 6
4 | GPR[rd] ← r_{31.. 0}

```

- **jambu.fsr.21 rd, rs1, rs2**

```

1 | x_hi ← GPR[rs2]
2 | x_lo ← GPR[rs1]
3 | r ← ( x_hi || x_lo ) >>> 21
4 | GPR[rd] ← r_{31.. 0}

```

- **jambu.fsr.27 rd, rs1, rs2**

```

1 | x_hi ← GPR[rs2]
2 | x_lo ← GPR[rs1]
3 | r ← ( x_hi || x_lo ) >>> 27
4 | GPR[rd] ← r_{31.. 0}

```

J Additional ISE design detail: XOODYAK

J.1 ISE definition

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

01	00000	rs2	rs1	111	rd	0101011	xoodyak.xorrol
----	-------	-----	-----	-----	----	---------	----------------

- **xoodyak.xorrol rd, rs1, rs2**

```

1 | x ← GPR[rs1]
2 | y ← GPR[rs2]
3 | r ← ( x <<< 5 ) ^ ( y <<< 14 )
4 | GPR[rd] ← r

```