RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography

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

¹ 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 Instruction Set Extensions (ISEs) for nine of the ten LWC final round submissions, namely Ascon, Elephant, GIFT-COFB, Grain-128AEAdv2, PHOTON-Beetle, Romulus, Sparkle, TinyJAMBU, and Xoodyak. We use RISC-V as the base instruction set architecture, but argue the analysis and designs offer more general insight. Our experimental results show that the more hardware-oriented candidates can achieve a higher speed-up through ISE than the more software-oriented ones, but nonetheless the latter still outperform the former in terms of throughput.

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 undeniably represent an 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 [MBTM] 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+1], and then again to 10 round 3 or final round submissions in 2021 [TMC+]. 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 footprint), 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 the design space of implementation techniques, 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 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, DGvK19]), 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.

**Organisation.** The paper is organised as follows. In Section 2 we present various background information, including work related to subsequent sections. In Section 3 we analyse 9 of the 10 LWC final round submissions, and produce associated ISE designs based on use of RISC-V as the base ISA. More specifically, we consider ISEs for RV32GC. 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. In doing
so, we introduce several implementation techniques of stand-alone value. For example, we show how to optimise bit-sliced implementation of GIFT-128 (for GIFT-COFB) using bit-manipulation instructions, rendering it more efficient than a fix-slicing alternative. Note that all material associated with the paper, e.g., source code relating to both hardware and software implementations, is available under an open source license.

**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. We only consider ISEs for those kernels, and, moreover, partial implementation of them 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 ISAP: the kernel used (namely the Ascon-\(p\) permutation) is already catered for by consideration of other algorithms (namely Ascon).

3. 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.

4. We only consider a 32-bit base ISA (and also ISEs for it therefore). Although consideration of a wider set of base 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 base ISA is deemed out of scope.

5. We do not consider support in the base ISA nor ISEs for countermeasures against implementation attack.

## 2 Background

**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 base 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 the 32-bit [SCA19, Chapter 2] RV32GC integer base ISA; the implied set of extensions therefore includes 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]. Given the context, we supplement this set by assuming 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] also form part of the base ISA we then extend with LWC-specific ISEs.

**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

---

1See https://github.com/scarv/lwise.
may be omitted. Let \( GPR[i] \), for \( 0 \leq i < 32 \), denote the \( i \)-th entry of the general-purpose register file. Note that \( GPR[0] \) is fixed to 0, in the sense reads from it always yields 0 and writes to it are ignored. Let \( x \ll y \) and \( x \ll\ll y \) (resp. \( x \gg y \) and \( x \gg\gg y \)) 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 \( x_{i..l} \) 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 \texttt{XLEN} to denote the word size. We adopt same approach, meaning \texttt{XLEN} = 32 because the context is RV32GC. The design process for a given algorithm potentially yields multiple ISE variants. To ensure clarity, let \( V_{XLEN} \) denote some \( i \)-th ISE variant which extends the base ISA associated with the stated \texttt{XLEN}; \( * \) can act as a wildcard for the variant index. For example, \( V_{0}^{32} \) and \( V_{1}^{32} \) would denote the 0-th and 1-st ISE variants for RV32GC, and \( V_{*}^{32} \) would denote all variants for RV32GC.

**Related work.** Steinegger and Primas [SP21] describe an ISE for RV32 to support Ascon-\( p \), implementing and evaluating it using the RISC-V 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-coded general-purpose registers to store the state), which, although delivering performance, arguably renders it more akin to a co-processor than traditional ISE.

Altunay 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 \( r_{s} \), it loads five 32-bit inputs \( x_{i} \leftarrow \text{MEM}[GPR[r_{s} + 4 \cdot i]^{3}] \), applies the S-box to produce outputs \( r_{i} \) from the inputs \( x_{i} \), then stores five 32-bit outputs \( \text{MEM}[GPR[r_{s} + 4 \cdot i]^{4}] \leftarrow r_{i} \), where \( 0 \leq i < 5 \) throughout.

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 GPR[\( r_{s} \)]. 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 GPR[\( r_{s} \)] and GPR[\( r_{s} \)] (with two instructions required to compute the most- and least-significant 32-bit half of the result). Note that this ISE cannot be used for either GIFT-128-128 or Skinny-128-384+, due to, e.g., the diffing substitution and permutation layers used (stemming from the different block size, per [BPP+17, Section 2] and [BJK+16, Section 2]).

### 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(s) 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 3.11 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):

**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. However, 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) specification, and 2) offer an easier route to standardisation and deployment as a result of limiting impact on other aspects of the base ISA. Beyond this, the constraints also facilitate extrapolation to other base 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 $\chi$ mapping of Keccak, and (iii) a permutation layer consisting of linear functions that are similar to the $\Sigma$ 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.

---

2https://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 \( x = x \oplus (x \gg n) \oplus (x \gg m) \) on each 64-bit word \( x \) of the state. On 32-bit ARM processors, the Ascon-\( p \) permutation is usually implemented in a bit-interleaved 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 exploit the “free” 32-bit rotations of ARM to speed up the permutations layer, but this comes at the expense of conversions between the bit-interleaved representation and normal representation whenever data is injected into or extracted from the state. Therefore, bit-interleaving makes no sense when targeting the RV32GC platform.

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 \( x = x \oplus (x \gg n) \oplus (x \gg m) \) map naturally to two custom \texttt{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.

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

3.3 Elephant

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

\[
\begin{align*}
\text{Dumbo} & = \text{Elephant-Spongent-}\pi[160] \\
\text{Jumbo} & = \text{Elephant-Spongent-}\pi[176] \\
\text{Delirium} & = \text{Elephant-Keccak-}f[200]
\end{align*}
\]

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

Kernel overview. Spongent-\( \pi[160] \) 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 \( \text{ICounter}_{160} \), i.e., \( 0^{153} \parallel \text{ICounter}_{160}(i) \), while the other one is \( \text{rev} \left( 0^{153} \parallel \text{ICounter}_{160}(i) \right) \), where \( i \) denotes the round index and \( \text{rev} \) is a function reversing the order of the bits of its input; 2) \( \text{sBoxLayer}_{160} \), a 4-bit S-box applied 40 times in parallel; 3) \( \text{pLayer}_{160} \), moving the bit \( j \) of state to bit position \( 40 \cdot j \mod 159 \) while the bit 159 keeps unmoved.

Implementation options. We developed the pure-software implementation of Spongent-\( \pi[160] \) 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, Zbkx provides a dedicated instruction for the parallel 4-bit S-box, namely \texttt{xperm4}, which is very beneficial.
for sBoxLayer\textsubscript{160}. Concretely, the xperm-style look-up table for sBoxLayer\textsubscript{160} is constructed with three registers before Sponge-π\textsubscript{160} starts:

\begin{verbatim}
li rl, 0xF4120BDE ; the lower half of S-box look-up table
li rh, 0x63C958A7 ; the higher half of S-box look-up table
li rm, 0x88888888 ; the mask used in xperm-style S-box
\end{verbatim}

Each 32-bit word $S_i$ (stored in $rx$) can perform eight 4-bit S-boxes simultaneously with two xperm\textsubscript{4} and two XOR instructions via

\begin{verbatim}
xperm\textsubscript{4} ry, rl, rx
xor rx, rx, rm
xperm\textsubscript{4} rx, rh, rx
xor rx, rx, ry
\end{verbatim}

so in each round the whole sBoxLayer\textsubscript{160} needs 20 instructions in total. Last, we divide the pLayer\textsubscript{160} 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 3.11) to swap the bits between different words, i.e.,

\begin{verbatim}
SWAPMOVE(S0, S1, 0x00000000, 8);
SWAPMOVE(S0, S2, 0x00000000, 16);
SWAPMOVE(S0, S3, 0x00000000, 24);
SWAPMOVE(S1, S2, 0x00000000, 8);
SWAPMOVE(S1, S3, 0x00000000, 24);
SWAPMOVE(S2, S3, 0x00000000, 8);
SWAPMOVE(S2, S4, 0x00000000, 16);
SWAPMOVE(S3, S4, 0x00000000, 24);
\end{verbatim}

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 pLayer\textsubscript{160} (i.e., two “unzip” instructions) at the end. Moreover, we designed two instructions for the specific SWAPMOVE operations used in our second step of pLayer\textsubscript{160}. 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 (located in the supplementary material).

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. AddRoundKey includes three sub-steps: add round key (to $S_1$ and $S_2$), add round constant
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 $W_6 \parallel W_7$. With the help of pack instruction, we can quickly obtain $W_6 \gg 2 \parallel W_7 \gg 12$ through

```
pack ry, rx, rx ; ry = ( W7 ) \parallel ( W7 )
rori rx, rx, 16 ; rx = ( W7 ) \parallel ( W6 )
pack rx, rx, rx ; rx = ( W6 ) \parallel ( W6 )
rori ry, ry, 12 ; ry = ( W7 >>> 12 ) \parallel ( W7 >>> 12 )
or
rori rx, rx, 2 ; rx = ( W6 >>> 2 ) \parallel ( W6 >>> 2 )
pack rx, ry, rx ; rx = ( W6 >>> 2 ) \parallel ( 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 both. 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 3.11), 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 (located in the supplementary material).

### 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-128 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 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 (located in the supplementary material).

### 3.6 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 \cite{BCD+21, Section 3.3}. We focus on the primary algorithm PHOTON-Beetle-AEAD\cite{128}, and, more specifically therefore, a kernel represented by the PHOTON\textsubscript{256} permutation (see also \cite{GPP11}).

**Kernel overview.** The PHOTON\textsubscript{256} permutation operates on an internal state of 256 bits, organised into an \((8 \times 8)\)-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 \cite[Section 2.2]{GPP11}, 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 \(F_{2^4}\).

**Implementation options.** As reflected by the submission, 3 implementation techniques are applicable to PHOTON\textsubscript{256}; 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 \(F_{2^4}\), 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 \cite[Sections 2.4+2.5]{SCA22} for AES (as documented in \cite[MNP+21]{MNP+21}, stemming from work by Nadehara et al. \cite{NIK04} and Saarinen \cite{Saa20}).

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

### 3.7 Romulus

**Submission overview.** The Romulus \cite{GIK+21} submission specifies the AEAD algorithms Romulus-N \cite[Section 2.4.3]{GIK+21}, Romulus-M \cite[Section 2.4.4]{GIK+21}, and Romulus-T \cite[Section 2.4.5]{GIK+21}, and the hash function algorithm Romulus-H \cite[Section 2.4.6]{GIK+21}. 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 \cite{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 \((4 \times 4)\)-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 tweakey 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 implementation 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 F (located in the supplementary material).

3.8 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 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 instruction 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 $\ell$ 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 Zkbk.

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 $x = x \oplus (y \gg n)$ and $x = x + (y \gg n)$, 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 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 $\ell$ operation.

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

3.9 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 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 XoodooYak 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 rounds 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 $w = (S_i \gg n) \land (S_j \ll (32 - n))$.

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 H (located in the supplementary material).

### 3.10 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 $\theta$, a non-linear layer $\chi$, two plane-shifting layers ($\rho_{west}$ and $\rho_{east}$) between them, and a round-constant addition. Both $\rho$ 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 $\theta$ and in the $\chi$ layer, state-bits interact only vertically, i.e. within 3-bit columns. The $\theta$ layer mainly executes XORs and left-rotations by 5 and 14 bits. Finally, the non-linear layer $\chi$ 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.** Implementation for 32-bit ARM microcontrollers are normally optimized to take advantage of the “implicit” rotations of the second operand that most arithmetic/logical instruction offer. An optimization technique known as lane complementing allows one to reduce the number of bitwise complements that have to be carried out in the $\chi$ transformation from 12 per round to three. This optimization is not necessary on ARM due to the bic instruction, which combines a logical AND with a bitwise complement of the second operand, but reduces the execution time on RV32GC platforms as demonstrated in [CJL+20].

**ISE description.** When adhering to the requirements for custom instructions mentioned in Section 2, then the only opportunity to speed up Xoodoo[12] is the manipulation of the parity-plane (i.e., three 32-bit parity-lanes) through an operation of the form $e = (p \ll 5) \oplus (p \ll 14)$. 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 I (located in the supplementary material).

### 3.11 Discussion

**Observations regarding ISA design.**
- There are several algorithms (e.g., Sparkle) where operations of the form

\[
\text{GPR}[rd] \leftarrow \text{GPR}[r_{s1}] \odot (\text{GPR}[r_{s2}] \oplus \text{imm})
\]
for ⊙ ∈ {⊕, +, −, ...} and □ ∈ {≪, ≫, ≪, ≫} are useful. Consider, without loss of
generality, an example operation where ⊙ = ⊕ and □ = ≪ is realised using the base ISA
by the 2-instruction sequence

\[
\begin{align*}
\text{slli } & \text{rx, ry, imm} \\
\text{xor } & \text{rx, rx, rz}
\end{align*}
\]

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 “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

\[
\begin{align*}
\text{c.slli } & \text{ry, imm} \\
\text{c.xor } & \text{ry, rz}
\end{align*}
\]

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 on a
  given 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., ror in Zbkb [SCA22, Section 3.26] and similar) but not 64-bit
operations when XLEN = 32, for example. 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.

**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 \((8 \times 8)\)-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 awkward, 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 enable 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 first\(^3\) to define and make use of this primitive, which, with some cosmetic alterations, is captured by the following:

```plaintext
algorithm SWAPMOVE(x, y, m, n) begin
    t ← y ⊕ (x ≫ n)
    t ← t ∧ m
    x ← x ⊕ (t ≫ n)
    y ← y ⊕ t
    return (x, y)
end
```

The basic idea is that some bits in y are swapped with some bits in x, 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 PHOTON\(_{256}\) for example, which initialises an \((8 \times 8)\)-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 PHOTON\(_{256}\) 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.

\(^3\)Their goal is efficient software implementation of permutations, such as those used by DES; they cite some prior art, e.g., noting “his technique is utilised in versions of DES available from the Internet (for example Eric Young’s libdes)”.
Figure 2: A block diagram of the host core, highlighting our modifications (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.

(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 base 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 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.

Figure 2 illustrates the result, with our modifications highlighted in red. Note that the LWC FU realises a given ISE design so is different for each ISE design therefore; the Zbkb/x FU realises the Zbkb and Zbkx extensions so is fixed across all ISE designs.

\footnote{Per Section 2, recall that although Zbkb and Zbkx represent extensions to RV32GC, for example, they form part of the base ISA we consider; from the perspective of Rocket they are still (unsupported) extensions, however, so need an associated implementation.}
Table 1: A per-algorithm summary of the base and kernel implementations.

<table>
<thead>
<tr>
<th>Submission</th>
<th>Base implementation</th>
<th>Kernel implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascon</td>
<td>ascon128v1/ref</td>
<td>P[6</td>
</tr>
<tr>
<td>Elephant</td>
<td>elephant160v2/ref</td>
<td>permutation</td>
</tr>
<tr>
<td>Grain-128AEADV2</td>
<td>grain128aeadv2/v64</td>
<td>grain_keystream32</td>
</tr>
<tr>
<td>PHOTON-Beetle</td>
<td>photonbeetle128v1/ref</td>
<td>PHOTON_Permutation</td>
</tr>
<tr>
<td>Romulus</td>
<td>romulusn/ref</td>
<td>Skinny_{128,384_plus_enc</td>
</tr>
<tr>
<td>Sparkle</td>
<td>schwaemm256v2/opt</td>
<td>Sparkle_opt</td>
</tr>
<tr>
<td>TinyJAMBU</td>
<td>tinyjambu128v2/opt</td>
<td>state_update</td>
</tr>
<tr>
<td>Xoodyak</td>
<td>xoodyak_round3/ref</td>
<td>Xoodoo_Permute_12rounds</td>
</tr>
</tbody>
</table>

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 xc7k160tfbg676) target FPGA, and a Xilinx Spartan-6 (model xc6slx45) 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, as now realised by the host core, we developed an associated software implementation. For a given algorithm, we start with a base implementation. This is the source code submitted for a given algorithm. The base 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.

Low-level strategy. We use a RISC-V capable instance of the GNU tool-chain 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 caller-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.

---

For 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 2: Results of hardware-oriented evaluation, i.e., realisation of each ISE design: the per-algorithm results detail area measured in FPGA LUTs (plus overhead versus baseline in parentheses).

<table>
<thead>
<tr>
<th>Submission</th>
<th>Base core</th>
<th>Base core + Zbkb/x</th>
<th>Base core + Zbkb/x + V_{32}</th>
<th>Base core + Zbkb/x + V_{32}^2</th>
<th>Base core + Zbkb/x + V_{32}^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elephant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gift-COFB (BS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gift-COFB (FS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GIFT-COFB (BS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GIFT-COFB (FS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHOTON-Beetle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Romulus (TB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Romulus (FS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPARKLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TinyJAMBU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOOYUK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

- Although we do not consider implementation attack countermeasures per se, all replacement kernel implementations are constant-time: delivering this property is made easier by use of an ISE, versus some\(^7\) alternative implementation strategies.

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 existing\(^8\) ISE-supported implementation of AES-GCM [SCA07]. We attempt to align said implementation as closely as possible with the API used for LWC algorithms, by 1) “upgrading” it to support additional data, and 2) parameterising it using a 128-bit key.

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 base Rocket host 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 V_{32}^2 variant for SPARKLE demands the most area: implementation of the Zbkb/x and LWC FUs imply a 14% and 22% overhead respectively, meaning 36% cumulative versus the baseline.

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 baseline.

**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

\(^7\)It 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; this is clearly unattractive from the perspective of implementation attacks.

\(^8\)https://github.com/rvkrypto/rvkrypto-fips
Table 3: Results of software-oriented evaluation, i.e., utilisation of each ISE design: the per-algorithm results detail latency measured in clock cycles (plus overhead versus baseline in parentheses) and footprint measured in bytes (plus overhead versus baseline in parentheses) associated with use of the original and replacement kernel implementations.

<table>
<thead>
<tr>
<th>Submission</th>
<th>Kernel</th>
<th>Metric</th>
<th>RV32GC</th>
<th>RV32GC + Zbkb/x</th>
<th>RV32GC + Zbkb/x + V_{32}</th>
<th>RV32GC + Zbkb/x + V_{32}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCON</td>
<td>P6</td>
<td>latency footprint</td>
<td>700.00 (1.00x)</td>
<td>280.00 (2.50x)</td>
<td>500.00 (1.00x)</td>
<td>280.00 (2.50x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>footprint</td>
<td>2718.00 (1.00x)</td>
<td>1050.00 (2.39x)</td>
<td>2718.00 (1.00x)</td>
<td>1050.00 (2.39x)</td>
</tr>
<tr>
<td>Elephant</td>
<td>permutation</td>
<td>latency</td>
<td>13864.00 (1.00x)</td>
<td>1944.00 (8.11x)</td>
<td>13864.00 (1.00x)</td>
<td>1944.00 (8.11x)</td>
</tr>
<tr>
<td>GIFT-COFB (BS)</td>
<td>gifti128</td>
<td>latency footprint</td>
<td>1451.00 (1.00x)</td>
<td>641.00 (2.31x)</td>
<td>1451.00 (1.00x)</td>
<td>641.00 (2.31x)</td>
</tr>
<tr>
<td>GIFT-COFB (FS)</td>
<td>gifti128</td>
<td>latency footprint</td>
<td>1386.00 (1.00x)</td>
<td>972.00 (1.43x)</td>
<td>1386.00 (1.00x)</td>
<td>972.00 (1.43x)</td>
</tr>
<tr>
<td>Grain-128AEDv2</td>
<td>grain_keystream32</td>
<td>latency footprint</td>
<td>235.00 (1.00x)</td>
<td>86.00 (2.73x)</td>
<td>235.00 (1.00x)</td>
<td>86.00 (2.73x)</td>
</tr>
<tr>
<td>PHOTON-Beetle</td>
<td>PHOTON_Permutation</td>
<td>latency footprint</td>
<td>67555.00 (1.00x)</td>
<td>1455.00 (4.51x)</td>
<td>67555.00 (1.00x)</td>
<td>1455.00 (4.51x)</td>
</tr>
<tr>
<td>Romulus (TB)</td>
<td>Skinny_128_384_plus_enc</td>
<td>latency footprint</td>
<td>14268.00 (1.00x)</td>
<td>1502.00 (9.50x)</td>
<td>14268.00 (1.00x)</td>
<td>1502.00 (9.50x)</td>
</tr>
<tr>
<td>Romulus (FS)</td>
<td>Skinny128_384_plus</td>
<td>latency footprint</td>
<td>17402.00 (1.00x)</td>
<td>7274.00 (2.39x)</td>
<td>17402.00 (1.00x)</td>
<td>7274.00 (2.39x)</td>
</tr>
<tr>
<td></td>
<td>precompute_rkeys</td>
<td>latency footprint</td>
<td>867.00 (1.00x)</td>
<td>200.00 (4.34x)</td>
<td>867.00 (1.00x)</td>
<td>200.00 (4.34x)</td>
</tr>
<tr>
<td></td>
<td>precompute_rtki</td>
<td>latency footprint</td>
<td>2814.00 (1.00x)</td>
<td>610.00 (4.61x)</td>
<td>2814.00 (1.00x)</td>
<td>610.00 (4.61x)</td>
</tr>
<tr>
<td>Sparkle</td>
<td>Sparkle_opt</td>
<td>latency footprint</td>
<td>5908.00 (1.00x)</td>
<td>4456.00 (1.33x)</td>
<td>5908.00 (1.00x)</td>
<td>4456.00 (1.33x)</td>
</tr>
<tr>
<td>TinyJAMBU</td>
<td>state_update (P1024)</td>
<td>latency footprint</td>
<td>515.00 (1.00x)</td>
<td>319.00 (1.60x)</td>
<td>515.00 (1.00x)</td>
<td>319.00 (1.60x)</td>
</tr>
<tr>
<td>Xoodyak</td>
<td>Xoodoo_Permute_12rounds</td>
<td>latency footprint</td>
<td>973.00 (1.00x)</td>
<td>777.00 (1.22x)</td>
<td>973.00 (1.00x)</td>
<td>777.00 (1.22x)</td>
</tr>
</tbody>
</table>

Table 4: Results of software-oriented evaluation, i.e., utilisation of each ISE design: the per-algorithm results detail latency measured in clock cycles (plus overhead versus baseline in parentheses) associated with use of the AEAD API (i.e., encryption and decryption via aead_encrypt and aead_decrypt, using 128 B plaintext, ciphertext, and associated data) as supported by the original and replacement kernel implementations.

<table>
<thead>
<tr>
<th>Submission</th>
<th>Functionality</th>
<th>Original kernel implementation</th>
<th>RV32GC</th>
<th>RV32GC + Zbkb/x</th>
<th>RV32GC + Zbkb/x + V_{32}</th>
<th>RV32GC + Zbkb/x + V_{32}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCON</td>
<td>aead_encrypt</td>
<td>43905.00 (1.00x)</td>
<td>52316 (1.23x)</td>
<td>16775 (2.96x)</td>
<td>52316 (1.23x)</td>
<td>16775 (2.96x)</td>
</tr>
<tr>
<td>Elephant</td>
<td>aead_decrypt</td>
<td>41414.00 (1.00x)</td>
<td>32694 (1.33x)</td>
<td>17159 (2.53x)</td>
<td>32694 (1.33x)</td>
<td>17159 (2.53x)</td>
</tr>
<tr>
<td>GIFT-COFB (BS)</td>
<td>aead_encrypt</td>
<td>16044010.00 (1.00x)</td>
<td>401534 (39.96x)</td>
<td>65118 (246.38x)</td>
<td>401534 (39.96x)</td>
<td>65118 (246.38x)</td>
</tr>
<tr>
<td>GIFT-COFB (FS)</td>
<td>aead_encrypt</td>
<td>16040475.00 (1.00x)</td>
<td>402787 (39.83x)</td>
<td>65079 (246.53x)</td>
<td>402787 (39.83x)</td>
<td>65079 (246.53x)</td>
</tr>
<tr>
<td>Grain-128AEDv2</td>
<td>aead_encrypt</td>
<td>687611.00 (1.00x)</td>
<td>125848 (16.35x)</td>
<td>29774 (24.76x)</td>
<td>125848 (16.35x)</td>
<td>29774 (24.76x)</td>
</tr>
<tr>
<td>Romulus (TB)</td>
<td>aead_encrypt</td>
<td>687543.00 (1.00x)</td>
<td>12093 (16.33x)</td>
<td>27819 (24.71x)</td>
<td>12093 (16.33x)</td>
<td>27819 (24.71x)</td>
</tr>
<tr>
<td>Romulus (FS)</td>
<td>aead_encrypt</td>
<td>687611.00 (1.00x)</td>
<td>41884 (16.42x)</td>
<td>33764 (20.36x)</td>
<td>41884 (16.42x)</td>
<td>33764 (20.36x)</td>
</tr>
<tr>
<td>Sparkle</td>
<td>aead_encrypt</td>
<td>687543.00 (1.00x)</td>
<td>41749 (16.47x)</td>
<td>33642 (20.44x)</td>
<td>41749 (16.47x)</td>
<td>33642 (20.44x)</td>
</tr>
<tr>
<td>TinyJAMBU</td>
<td>aead_encrypt</td>
<td>870582.00 (1.00x)</td>
<td>55826 (1.02x)</td>
<td>64083 (1.13x)</td>
<td>55826 (1.02x)</td>
<td>64083 (1.13x)</td>
</tr>
<tr>
<td>Xoodyak</td>
<td>aead_encrypt</td>
<td>860656.00 (1.00x)</td>
<td>84987 (1.02x)</td>
<td>63148 (1.37x)</td>
<td>84987 (1.02x)</td>
<td>63148 (1.37x)</td>
</tr>
</tbody>
</table>
(measured in clock cycles) and memory footprint (measured in bytes), and relative results i.e., improvement versus an associated baseline, captured by use of the base ISA alone. Note that for some kernels, e.g., GIFT and Romulus, we utilise 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. Also note that for some kernels, e.g., Sparkle and TinyJAMBU, Table 3 lists the same result for some $V^{32}$ and $V^{32}$. This is because the software implementation using those ISEs is similar (or even identical); their hardware implementation is different, however, since one is more general-purpose (resp. special-purpose) but has a larger (resp. smaller) area overhead.

For comparison, a 1-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; computation of the encryption key schedule via `aes128_enc_key_rvk32` (resp. decryption key schedule via `aes128_dec_key_rvk32`) requires 264 (resp. 719) cycles; computation of the GHASH function (dominated by a multiplication in $\mathbb{F}_{2^{128}}$) via `ghash_mul_rv32` requires 135 cycles.

Software: API. Table 4 presents 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. 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 framework\footnote{See, e.g., https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking, and results in [TMC+, Section 4 + Appendix A]: note that although the data format allows \textit{“x bytes of associated data and y bytes of message”}, the data itself has $x = y$ in all cases.} and so allows easier comparison of results. As such, Appendix J (located in the supplementary material) captures further cases beyond Table 4.

For comparison, 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).

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 9 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 3-address instructions), we first developed ISE designs for Ascon, Elephant, GIFT-COFB, Grain-128AEAdv2, PHOTON-Beetle, Romulus, Sparkle, TinyJAMBU, and Xoodyak, then implemented said designs using the RISC-V compliant Rocket host core. 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 memory 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 the base ISA 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 does 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- versus more software-oriented. Put simply, ISEs for the former typically offer a greater improvement than for the latter: 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 also 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.

References


[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


24 RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography


A Additional ISE design detail: Ascon

A.1 Additional notation

Define the look-up tables

\[ \text{ROT}_0 = \{ 19, 61, 1, 10, 7 \} \]
\[ \text{ROT}_1 = \{ 28, 39, 6, 17, 41 \} \]

A.2 \( \mathcal{V}_{32} \)

Instruction encoding.

| 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 |

Instruction semantics.

- \( \text{ascon.sigma.lo} \) rd, rs1, rs2, imm

1. \( x_{hi} \leftarrow \text{GPR}[rs2] \)
2. \( x_{lo} \leftarrow \text{GPR}[rs1] \)
3. \( x \leftarrow x_{hi} || x_{lo} \)
4. \( r \leftarrow x \oplus (x \gg \text{ROT}_0[\text{imm}]) \oplus (x \gg \text{ROT}_1[\text{imm}]) \)
5. \( \text{GPR}[rd] \leftarrow r_{63..32} \)

- \( \text{ascon.sigma.hi} \) rd, rs1, rs2, imm

1. \( x_{hi} \leftarrow \text{GPR}[rs2] \)
2. \( x_{lo} \leftarrow \text{GPR}[rs1] \)
3. \( x \leftarrow x_{hi} || x_{lo} \)
4. \( r \leftarrow x \oplus (x \gg \text{ROT}_0[\text{imm}]) \oplus (x \gg \text{ROT}_1[\text{imm}]) \)
5. \( \text{GPR}[rd] \leftarrow r_{63..32} \)
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

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

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

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

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

B.2 $\mathcal{V}_0^{32}$

Instruction encoding.

<table>
<thead>
<tr>
<th>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</th>
<th>0000</th>
<th>imm</th>
<th>rs2</th>
<th>rs1</th>
<th>111</th>
<th>rd</th>
<th>0001011</th>
<th>elephant.pstep.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>imm</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0001011</td>
<td>elephant.pstep.y</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>imm</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0001011</td>
<td>elephant.sstep</td>
<td></td>
</tr>
</tbody>
</table>

Instruction semantics.

• elephant.pstep.x rd, rs1, rs2, imm

```python
x ← GPR[rs1]
y ← GPR[rs2]

if ( imm == 0 ) {
    r ← SWAPMOVE32_X( x, y, 0x000000FF, 8 )
}
else if ( imm == 1 ) {
    r ← SWAPMOVE32_X( x, y, 0x000000FF, 16 )
}
else if ( imm == 2 ) {
    r ← SWAPMOVE32_X( x, y, 0x000000FF, 24 )
}
else if ( imm == 3 ) {
    r ← SWAPMOVE32_X( x, y, 0x0000FF00, 8 )
}
else if ( imm == 4 ) {
    r ← SWAPMOVE32_X( x, y, 0x0000FF00, 24 ) >> 24
}
else if ( imm == 5 ) {
    r ← SWAPMOVE32_X( x, y, 0x0000FF0000, 16 ) >> 16
}
else if ( imm == 6 ) {
    r ← SWAPMOVE32_X( x, y, 0x00FF0000, 8 ) >> 8
}
```
GPR[rd] ← r

• elephant.pstep.x rd, rs1, rs2, imm

   x ← GPR[rs1]
y ← GPR[rs2]
if (imm == 0) {
r ← SWAPMOVE32_Y(x, y, 0x000000FF, 8)
}
else if (imm == 1) {
r ← SWAPMOVE32_Y(x, y, 0x000000FF, 16)
}
else if (imm == 2) {
r ← SWAPMOVE32_Y(x, y, 0x000000FF, 24)
}
else if (imm == 3) {
r ← SWAPMOVE32_Y(x, y, 0x0000FF00, 8)
}
else if (imm == 4) {
r ← SWAPMOVE32_Y(x, y, 0x0000FF00, 24)
}
else if (imm == 5) {
r ← SWAPMOVE32_Y(x, y, 0x00FF0000, 16)
}
else if (imm == 6) {
r ← SWAPMOVE32_Y(x, y, 0x00FF0000, 8)
}
GPR[rd] ← r

• elephant.sstep rd, rs1

   x ← GPR[rs1]
r ← SBOX[ x_{31..28} ] || SBOX[ x_{27..24} ] || SBOX[ x_{23..20} ] || SBOX[ x_{19..16} ] || SBOX[ x_{15..12} ] || SBOX[ x_{11..8} ] || SBOX[ x_{7..4} ] || SBOX[ x_{3..0} ]
r ← SWAPMOVE32(r, 0x0A0A0A0A, 3)
r ← SWAPMOVE32(r, 0x00CC00CC, 6)
r ← SWAPMOVE32(r, 0x0000FF00, 12)
r ← SWAPMOVE32(r, 0x0000FF00, 8)
GPR[rd] ← r
C  Additional ISE design detail: GIFT-COBF

C.1  Additional notation

Define the function

\[
\text{SWAPMOVE32}(x, m, n) \{
\begin{align*}
t &\leftarrow x \oplus (x \gg n) \\
t &\leftarrow t \& m \\
t &\leftarrow t \oplus (t \ll n) \\
x &\leftarrow t \oplus x
\end{align*}
\]

return x
\]

i.e., a variant of \text{SWAPMOVE} [MPC00, Section 3.1].

C.2  $\gamma^{32}_0$ (for fix-slicing implementation)

Instruction encoding.

<table>
<thead>
<tr>
<th>imm</th>
<th>rs2</th>
<th>rs1</th>
<th>111</th>
<th>rd</th>
<th>0001011</th>
<th>gift.swapmove</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>imm</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0001011</td>
</tr>
<tr>
<td>01</td>
<td>imm</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0001011</td>
</tr>
<tr>
<td>10</td>
<td>imm</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0001011</td>
</tr>
<tr>
<td>11</td>
<td>imm</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0101011</td>
</tr>
<tr>
<td>00</td>
<td>imm</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0101011</td>
</tr>
</tbody>
</table>

Instruction semantics.

- **gift.swapmove rd, rs1, rs2, imm**

1. \( x \leftarrow \text{GPR}[rs1] \)
2. \( n \leftarrow \text{GPR}[rs2] \)
3. \( r \leftarrow \text{SWAPMOVE32}(x, n, \text{imm}) \)
4. \( \text{GPR}[rd] \leftarrow r \)

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

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

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

1. \( x_3 \leftarrow \text{GPR}[rs1]_{31..24} \)
2. \( x_2 \leftarrow \text{GPR}[rs1]_{23..16} \)
3. \( x_1 \leftarrow \text{GPR}[rs1]_{15..8} \)
4. \( x_0 \leftarrow \text{GPR}[rs1]_{7..0} \)
5. \( r \leftarrow (x_3 \gg \text{imm}) \lor (x_2 \gg \text{imm}) \lor (x_1 \gg \text{imm}) \lor (x_0 \gg \text{imm}) \)
6. \( \text{GPR}[rd] \leftarrow r \)

- **gift.rori.h rd, rs1, imm**
x₁ ← GPR[rs1]_{31..16}

x₀ ← GPR[rs1]_{15..0}

r ← ( x₁ >>> imm ) || ( x₀ >>> imm )

GPR[rd] ← r

• gift.key.reorg rd, rs1, imm

x ← GPR[rs1]

if ( imm == 0 ) {
  r ← SWAPMOVE32( x, 0x00550055, 9 )
  r ← SWAPMOVE32( r, 0x00003333, 15 )
  r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
} else if ( imm == 1 ) {
  r ← SWAPMOVE32( x, 0x11111111, 3 )
  r ← SWAPMOVE32( r, 0x03030303, 6 )
  r ← SWAPMOVE32( r, 0x000F000F, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
} else if ( imm == 2 ) {
  r ← SWAPMOVE32( x, 0x0000AAAA, 15 )
  r ← SWAPMOVE32( r, 0x00003333, 18 )
  r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
} else if ( imm == 3 ) {
  r ← SWAPMOVE32( x, 0x0A0A0A0A, 3 )
  r ← SWAPMOVE32( r, 0x00CC00CC, 6 )
  r ← SWAPMOVE32( r, 0x0000F0F0, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
}

GPR[rd] ← r

• gift.key.upstd rd, rs1

x ← GPR[rs1]

r ← ( ( x >> 12 ) & 0x0000000F )

r ← r | ( ( x & 0x00000FFF ) << 4 )

r ← r | ( ( x & 0x003F003F ) << 14 )

GPR[rd] ← r

• gift.key.updfix rd, rs1, imm

x ← GPR[rs1]

if ( imm == 0 ) {
  r ← SWAPMOVE32( x, 0x00003333, 16 )
  r ← SWAPMOVE32( r, 0x55554444, 1 )
}

if ( imm == 1 ) {
  r ← SWAPMOVE32( x, 0x33333333, 24 )
  r ← SWAPMOVE32( r, 0x000F000F, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
}

if ( imm == 2 ) {
  r ← SWAPMOVE32( x, 0x0F0F0F0F, 24 )
  r ← SWAPMOVE32( r, 0x000F000F, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
}

if ( imm == 3 ) {
  r ← SWAPMOVE32( x, 0x000F000F, 12 )
  r ← SWAPMOVE32( r, 0x000000FF, 24 )
}

else if ( imm == 4 ) {
  r ← SWAPMOVE32( x, 0x003F003F, 24 )
  r ← SWAPMOVE32( r, 0x00030003, 16 )
}

else if ( imm == 5 ) {
  r ← SWAPMOVE32( x, 0x000000FF, 24 )
  r ← SWAPMOVE32( r, 0x00030003, 16 )
}

else if ( imm == 6 ) {
  r ← SWAPMOVE32( x, 0x000000FF, 24 )
  r ← SWAPMOVE32( r, 0x00030003, 16 )
  r ← SWAPMOVE32( r, 0x00030003, 16 )
}

GPR[rd] ← r
else if ( imm == 7 ) {
    r ← (( x >> 18 ) & 0x00003030 ) | (( x & 0x01010101 ) << 3 )
    r ← r | (( x >> 14 ) & 0x0000C0C0 ) | (( x & 0x0000E0E0 ) << 15 )
}
else if ( imm == 8 ) {
    r ← (( x >> 4 ) & 0xFFF00000 ) | (( x & 0x000F0000 ) << 12 )
    r ← r | (( x >> 8 ) & 0x000000FF ) | (( x & 0x000000FF ) << 8 )
}
else if ( imm == 9 ) {
    r ← (( x >> 6 ) & 0x03FF0000 ) | (( x & 0x003F0000 ) << 10 )
    r ← r | (( x >> 4 ) & 0x0000FFF ) | (( x & 0x0000000F ) << 12 )
}
GPR[rd] ← r

C.3 $\Psi_0^{32}$ (for bit-slicing implementation)

Instruction encoding.

```
| 01 | 00000 | 00000 | rs1 | 110 | rd | 0101011 | gift.key.updstd |
| 11 | imm  | 00000 | rs1 | 110 | rd | 0101011 | gift.permbits.step |
```

Instruction semantics.

• gift.key.updstd rd, rs1

```
x ← GPR[rs1]  
(( x >> 12 ) & 0x0000000F )
r ← r | (( x & 0x000000FF ) << 4 )  
r ← r | (( x >> 2 ) & 0x3FF0000 )  
r ← r | (( x & 0x00300000 ) << 14 )  
GPR[rd] ← r
```

• gift.permbits.step rd, rs1, imm

```
x ← GPR[rs1]  
SWAPMOVE32( x, 0x0A0A0A0A, 3 )  
r ← SWAPMOVE32( r, 0x00000000 )  
r ← SWAPMOVE32( r, 0x00000000 )  
r ← r >>> imm  
GPR[rd] ← r
```
D Additional ISE design detail: Grain-128AEADv2

D.1 $V_0^{32}$

Instruction encoding.

<table>
<thead>
<tr>
<th>Address</th>
<th>imm</th>
<th>rs2</th>
<th>rs1</th>
<th>rd</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>00001</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>00100</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>00110</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>01000</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>01010</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>01100</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
<tr>
<td>10000</td>
<td>000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd 0101011</td>
</tr>
</tbody>
</table>

Instruction semantics.

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

  1. $x_{hi} \leftarrow \text{GPR}[rs1]$
  2. $x_{lo} \leftarrow \text{GPR}[rs2]$
  3. $x \leftarrow x_{hi} \| x_{lo}$
  4. $r \leftarrow x_{lo} \ll 26$ $	ext{imm}$
  5. $\text{GPR}[rd] \leftarrow r_{31..0}$

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

  1. $x_{hi} \leftarrow \text{GPR}[rs1]$
  2. $x_{lo} \leftarrow \text{GPR}[rs2]$
  3. $x \leftarrow x_{hi} \| x_{lo}$
  4. $r \leftarrow (x_{lo}) ^ (x_{lo} \gg 7)$
  5. $\text{GPR}[rd] \leftarrow r_{31..0}$

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

  1. $x_{hi} \leftarrow \text{GPR}[rs1]$
  2. $x_{lo} \leftarrow \text{GPR}[rs2]$
  3. $x \leftarrow x_{hi} \| x_{lo}$
  4. $r \leftarrow (x_{hi}) ^ (x_{lo} \gg 6) ^ (x_{lo} \gg 17)$
  5. $\text{GPR}[rd] \leftarrow r_{31..0}$

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

  1. $x_{hi} \leftarrow \text{GPR}[rs1]$
  2. $x_{lo} \leftarrow \text{GPR}[rs2]$
  3. $x \leftarrow x_{hi} \| x_{lo}$
  4. $r \leftarrow (x_{hi} \gg 26 ) ^ (x_{lo} \gg 26)$
  5. $\text{GPR}[rd] \leftarrow r_{31..0}$

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

  1. $x_{hi} \leftarrow \text{GPR}[rs1]$
  2. $x_{lo} \leftarrow \text{GPR}[rs2]$
  3. $x \leftarrow x_{hi} \| x_{lo}$
  4. $r \leftarrow (x_{hi} \gg 24 ) ^ (x_{lo} \gg 24)$
  5. $\text{GPR}[rd] \leftarrow r_{31..0}$

- **grain.gnn2 rd, rs1, rs2**
34  RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography

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

• grain.hnn0 rd, rs1, rs2
x_hi ← GPR[rs1]
x_lo ← GPR[rs2]
x ← x_hi || x_lo
r ← (x >> 2) ^ (x >> 15)
GPR[rd] ← r_{31..0}

• grain.hnn1 rd, rs1, rs2
x_hi ← GPR[rs1]
x_lo ← GPR[rs2]
x ← x_hi || x_lo
r ← ((x >> 4) ^ (x >> 13)
GPR[rd] ← r_{31..0}

• grain.hnn2 rd, rs1, rs2
x_hi ← GPR[rs1]
x_lo ← GPR[rs2]
x ← x_hi || x_lo
r ← ((x >> 9)^ (x >> 25)
GPR[rd] ← r_{31..0}

• grain.hln0 rd, rs1, rs2
x_hi ← GPR[rs1]
x_lo ← GPR[rs2]
x ← x_hi || x_lo
r ← ((x >> 13)^ (x >> 20)
GPR[rd] ← r_{31..0}
E Additional ISE design detail: PHOTON-Beetle

E.1 Additional notation

Let SBOX denote the 4-bit PHOTON S-box per [GP11], and GF2N_MUL denote multiplication in the PHOTON finite field. Define a look-up table

\[
M = \{ \{ 0x2, 0x4, 0x2, 0x8, 0x2, 0x8, 0x5, 0x6 \},
\{ 0xC, 0x9, 0x8, 0xD, 0x7, 0x5, 0x2 \},
\{ 0x4, 0x4, 0xD, 0x9, 0x4, 0x9, 0x9 \},
\{ 0x1, 0x6, 0x5, 0x1, 0xC, 0xD, 0xF, 0xE \},
\{ 0x7, 0x7, 0x5, 0x2 \},
\{ 0x4, 0x4, 0xD, 0x9, 0x4, 0x9, 0x9 \},
\{ 0x1, 0x6, 0x5, 0x1, 0xC, 0xD, 0xF, 0xE \},
\{ 0x7, 0x7, 0x5, 0x2 \} \}
\]

to configure the MixColumnsSerial round function.

E.2 V_0^{32}

Instruction encoding.

<table>
<thead>
<tr>
<th>0000</th>
<th>imm</th>
<th>rs2</th>
<th>rs1</th>
<th>i11</th>
<th>rd</th>
<th>1011011</th>
<th>photon.step</th>
</tr>
</thead>
</table>

Instruction semantics.

- photon.step rd, rs1, rs2, imm

```c
x ← GPR[rs1]
y ← GPR[rs2]
t ← SBOX( ( y >> (4 * imm) ) & 0xF )
r ← 0
for( int i = 0; i < 8; i++ ) {
r ← r | ( ( GF2N_MUL( M[ i ][ imm ], t ) ) << (4 * i) )
}
r ← r ^ x
GPR[rd] ← r
```
F  Additional ISE design detail: Romulus

F.1  Additional notation

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

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

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

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

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

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

```c
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

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

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

```c
SWAPMOVE32_Y(x,y,n,n) {
    t ← y ^ (x >> n)
    t ← t & m
    y ← y ^ (t)
    return y
}
```

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

F.2  $V^{32}_0$ (for table-based implementation)

Instruction encoding.
Instruction semantics.

- **romulus.rc.upd.enc rd, rs1**
  
  1. $x \leftarrow \text{GPR}[rs1]$
  2. $r \leftarrow \text{LFSR}_R(x)$
  3. $\text{GPR}[rd] \leftarrow r$

- **romulus.rc.use.enc.0 rd, rs1, rs2**
  
  1. $x \leftarrow \text{GPR}[rs1]$
  2. $y \leftarrow \text{GPR}[rs2]$
  3. $r \leftarrow y \land x_{3..0}$
  4. $\text{GPR}[rd] \leftarrow r$

- **romulus.rc.use.enc.1 rd, rs1, rs2**
  
  1. $x \leftarrow \text{GPR}[rs1]$
  2. $y \leftarrow \text{GPR}[rs2]$
  3. $r \leftarrow y \land x_{6..4}$
  4. $\text{GPR}[rd] \leftarrow r$

- **romulus.tk.upd.enc.0 rd, rs1, rs2, imm**
  
  1. $x \leftarrow \text{GPR}[rs1]$
  2. $y \leftarrow \text{GPR}[rs2]$
  3. 4. if ( imm == 1 ) {
    5. $r \leftarrow y_{(15..8)} || x_{(7..0)} || y_{(31..24)} || x_{(15..8)}$
    6. }
  8. else if( imm == 2 ) {
    9. $r \leftarrow \text{LFSR}_T(y_{(15..8)}) || \text{LFSR}_T(x_{(7..0)}) || \text{LFSR}_T(y_{(31..24)}) || \text{LFSR}_T(x_{(15..8)})$
   10. }
  12. else if( imm == 3 ) {
    13. $r \leftarrow \text{LFSR}_T(x_{(15..8)}) || \text{LFSR}_T(y_{(7..0)}) || \text{LFSR}_T(y_{(23..16)}) || \text{LFSR}_T(x_{(23..16)})$
    14. }
  16. 17. $\text{GPR}[rd] \leftarrow r$

- **romulus.tk.upd.enc.1 rd, rs1, rs2, imm**
  
  1. $x \leftarrow \text{GPR}[rs1]$
  2. $y \leftarrow \text{GPR}[rs2]$
  3. 4. if ( imm == 1 ) {
    5. $r \leftarrow x_{(31..24)} || y_{(7..0)} || y_{(23..16)} || x_{(23..16)}$
    6. }
  8. else if( imm == 2 ) {
    9. $r \leftarrow \text{LFSR}_T(x_{(31..24)}) || \text{LFSR}_T(y_{(7..0)}) || \text{LFSR}_T(x_{(23..16)}) || \text{LFSR}_T(y_{(23..16)})$
   10. }
  12. else if( imm == 3 ) {
    13. $r \leftarrow \text{LFSR}_T(x_{(31..24)}) || \text{LFSR}_T(y_{(7..0)}) || \text{LFSR}_T(y_{(23..16)}) || \text{LFSR}_T(x_{(23..16)})$
    14. }
  16. 17. $\text{GPR}[rd] \leftarrow r$
• **romulus.rstep.enc rd, rs1, rs2, imm**

```plaintext
x ← GPR[rs1]
y ← GPR[rs2]
if ( imm == 2 ) {
    y ← 2
} else if( imm == 3 ) {
    y ← 0
}
t ← \text{SBOX}[ x_{31..24} ] || \text{SBOX}[ x_{23..16} ] || \text{SBOX}[ x_{15..8} ] || \text{SBOX}[ x_{7..0} ]
t ← t ^ y
if ( imm == 0 ) {
    r ← t <<< 0
} else if( imm == 1 ) {
    r ← t <<< 8
} else if( imm == 2 ) {
    r ← t <<< 16
} else if( imm == 3 ) {
    r ← t <<< 24
}
GPR[rd] ← r
```

**F.3 \( \Psi^3_0 \) (for fix-slicing implementation)**

**Instruction encoding.**

<table>
<thead>
<tr>
<th></th>
<th>imm</th>
<th>rs2</th>
<th>rs1</th>
<th>rd</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>1111011</td>
</tr>
<tr>
<td>0001</td>
<td>imm</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
</tr>
<tr>
<td>0010</td>
<td>imm</td>
<td>0000</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
</tr>
<tr>
<td>0100</td>
<td>imm</td>
<td>0000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
</tr>
<tr>
<td>0101</td>
<td>imm</td>
<td>0000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
</tr>
<tr>
<td>0110</td>
<td>imm</td>
<td>0000</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
</tr>
<tr>
<td>0100</td>
<td>0000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
</tr>
<tr>
<td>0101</td>
<td>0000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
</tr>
</tbody>
</table>

**Instruction semantics.**

• **romulus.mixcolumns rd, rs1, imm**

```plaintext
x ← GPR[rs1]
r ← x ^ ((x >>> 24) & 0x0C0C0C0C) >>> 30
r ← r ^ (( ( r >> 16 ) & 0x0C0C0C0C ) >>> 4 )
r ← r ^ (( r >>> 8 ) & 0x0C0C0C0C ) >>> 2
if ( imm == 1 ) {
    r ← x ^ ((x >>> 16) & 0x30303030) >>> 30
    r ← r ^ (( ( r ) & 0x03030303 ) >>> 28 )
    r ← r ^ (( ( r >> 16 ) & 0x30303030 ) >>> 2 )
} else if ( imm == 2 ) {
    r ← x ^ ((x >>> 8) & 0x0C0C0C0C) >>> 6
    r ← r ^ (( ( r >>> 24 ) & 0x0C0C0C0C ) >>> 2 )
```

RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography
else if ( imm == 3 ) {
  r ← x ^ ((x & 0x03030303) >>> 30)
  r ← r ^ ((r & 0x30303030) >>> 4)
  r ← r ^ ((r & 0x03030303) >>> 26)
}
GPR[rd] ← r

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

GPR[rd] ← romulus.swapmove.x rd, rs1, rs2, imm

else if ( imm == 0 ) {
  r ← SWAPMVGE32_X( x, y, 0x55555555, 1 )
} else if ( imm == 1 ) {
  r ← SWAPMVGE32_X( x, y, 0x30303030, 2 )
} else if ( imm == 2 ) {
  r ← SWAPMVGE32_X( x, y, 0x0C0C0C0C, 4 )
} else if ( imm == 3 ) {
  r ← SWAPMVGE32_X( x, y, 0x03030303, 6 )
} else if ( imm == 4 ) {
  r ← SWAPMVGE32_X( x, y, 0x0C0C0C0C, 2 )
} else if ( imm == 5 ) {
  r ← SWAPMVGE32_X( x, y, 0x03030303, 4 )
} else if ( imm == 6 ) {
  r ← SWAPMVGE32_X( x, y, 0x03030303, 2 )
} else if ( imm == 7 ) {
  r ← SWAPMVGE32( x, 0x0A0A0A0A, 3 )
}
GPR[rd] ← r

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

GPR[rd] ← romulus.swapmove.y rd, rs1, rs2, imm

else if ( imm == 0 ) {
  r ← SWAPMVGE32_Y( x, y, 0x55555555, 1 )
} else if ( imm == 1 ) {
  r ← SWAPMVGE32_Y( x, y, 0x30303030, 2 )
} else if ( imm == 2 ) {
  r ← SWAPMVGE32_Y( x, y, 0x0C0C0C0C, 4 )
} else if ( imm == 3 ) {
  r ← SWAPMVGE32_Y( x, y, 0x03030303, 6 )
} else if ( imm == 4 ) {
  r ← SWAPMVGE32_Y( x, y, 0x0C0C0C0C, 2 )
} else if ( imm == 5 ) {
  r ← SWAPMVGE32_Y( x, y, 0x03030303, 4 )
} else if ( imm == 6 ) {
  r ← SWAPMVGE32_Y( x, y, 0x03030303, 2 )
}
GPR[rd] ← r

• romulus.permtk rd, rs1, imm

GPR[rd] ← romulus.permtk rd, rs1, imm

else if ( imm == 0 ) {
  r ← ( ( ( x >>> 14 ) & 0x000000FF ) << 16 )
  r ← r | ( ( ( x ) & 0x000000FF ) << 2 )
40 RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography

```plaintext
\begin{align*}
7 & \text{r} \leftarrow \text{r} | ((\text{x} \land 0\times0033CC00) \gg 8) \\
8 & \text{r} \leftarrow \text{r} | ((\text{x} \land 0\times0000C000) \gg 18) \\
9 & \}
10 & \text{else if} (\text{imm} == 1) \{ \\
11 & \text{r} \leftarrow ((\text{x} \gg 22) \land 0\timesCC0000CC) \\
12 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 16) \land 0\times3300CC00) \\
13 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 24) \land 0\times00CC3300) \\
14 & \}
15 & \text{else if} (\text{imm} == 2) \{ \\
16 & \text{r} \leftarrow ((\text{x} \gg 12) \land 0\times000000CC) \\
17 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 10) \land 0\times00003333) \\
18 & \}
19 & \text{else if} (\text{imm} == 3) \{ \\
20 & \text{r} \leftarrow ((\text{x} \gg 12) \land 0\times0000CC33) \\
21 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 16) \land 0\times000000CC) \\
22 & \}
23 & \text{else if} (\text{imm} == 4) \{ \\
24 & \text{r} \leftarrow ((\text{x} \gg 16) \land 0\times000000CC) \\
25 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 8) \land 0\times0000CC00) \\
26 & \}
27 & \text{else if} (\text{imm} == 5) \{ \\
28 & \text{r} \leftarrow ((\text{x} \gg 8) \land 0\times000000CC) \\
29 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 30) \land 0\times000000CC) \\
30 & \}
31 & \text{else if} (\text{imm} == 6) \{ \\
32 & \text{r} \leftarrow ((\text{x} \gg 10) \land 0\times0000CC00) \\
33 & \text{r} \leftarrow \text{r} | ((\text{x} \gg 16) \land 0\times000000CC) \\
34 & \}
35 & \}
36 & \text{GPR[rd]} \leftarrow \text{r}
\end{align*}

*romulus.tkupd.0 rd, rs1, imm*

```plaintext
\begin{align*}
1 & \text{x} \leftarrow \text{GPR[rs1]} \\
2 & \text{if} (\text{imm} == 0) \{ \\
3 & \text{r} \leftarrow ((\text{x} \gg 26) \land 0\timesC3C3C3C3) \\
4 & \}
5 & \text{else if} (\text{imm} == 1) \{ \\
6 & \text{r} \leftarrow ((\text{x} \gg 16) \land 0\times0F0F0F0F) \\
7 & \}
8 & \text{else if} (\text{imm} == 2) \{ \\
9 & \text{r} \leftarrow ((\text{x} \gg 10) \land 0\timesC3C3C3C3) \\
10 & \}
11 & \text{GPR[rd]} \leftarrow \text{r}
\end{align*}

*romulus.tkupd.1 rd, rs1, imm*

```plaintext
\begin{align*}
1 & \text{x} \leftarrow \text{GPR[rs1]} \\
2 & \text{if} (\text{imm} == 0) \{ \\
3 & \text{r} \leftarrow ((\text{x} \gg 28) \land 0\times03030303) \\
4 & \text{r} \leftarrow ((\text{x} \gg 12) \land 0\times0C0C0C0C) \\
5 & \}
6 & \text{else if} (\text{imm} == 1) \{ \\
7 & \text{r} \leftarrow ((\text{x} \gg 14) \land 0\times03030303) \\
8 & \text{r} \leftarrow ((\text{x} \gg 6) \land 0\times0C0C0C0C) \\
9 & \}
10 & \text{else if} (\text{imm} == 2) \{ \\
11 & \text{r} \leftarrow ((\text{x} \gg 12) \land 0\times03030303) \\
12 & \text{r} \leftarrow ((\text{x} \gg 28) \land 0\times0C0C0C0C) \\
13 & \}
14 & \text{else if} (\text{imm} == 3) \{
```
```c
16  r ← ( ( x >> 30 ) & 0x30303030 )
17  r ← r | ( ( x >> 22 ) & 0x00C0C0C0 )
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 ) | ( ( ( r << 1 ) & 0xAAAAAAAA ) )
  5  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 ) | ( ( ( r << 1 ) & 0xAAAAAAAA ) )
  5  GPR[rd] ← r
```
G Additional ISE design detail: Sparkle

G.1 Additional notation

Define the look-up tables

\[ \text{ROT}_0 = \{31, 17, 0, 24\} \]
\[ \text{ROT}_1 = \{24, 17, 31, 16\} \]
\[ \text{RCON} = \{0x87E15162, 0xB7F715880, 0x3884D456, 0x324E7738, \\
0xBB1185EB, 0x4F7C7B57, 0xCF9BA1C8, 0x32B3293D\} \]

Define the function

\[ \text{ELL}(x) = \{ \text{return} \ (x \ ^\ (x \ll 16)) \gg 16 \} \]

G.2 \( \wp_{32}^* \) (i.e., common across all variants)

Instruction encoding.

| 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 |
|------------------|------------------|------------------|------------------|------------------|
| 00000010         | rs2              | rs1              | 111              | rd               | 11101111         | sparkle.ell     |
| 0000             | imm              | rs2              | rs1              | 110              | rd               | 1011011         | sparkle.rcon   |

Instruction semantics.

- \( \text{sparkle.ell rd, rs1, rs2} \)

1. \( x \leftarrow \text{GPR[rs1]} \)
2. \( y \leftarrow \text{GPR[rs2]} \)
3. \( r \leftarrow \text{ELL}(x \ ^\ y) \)
4. \( \text{GPR[rd]} \leftarrow r \)

- \( \text{sparkle.rcon rd, rs1, imm} \)

1. \( x \leftarrow \text{GPR[rs1]} \)
2. \( r \leftarrow x ^\ (\text{RCON[imm]}) \)
3. \( \text{GPR[rd]} \leftarrow r \)

G.3 \( \wp_0^{32} \)

Instruction encoding.

| 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               | 01010111         | sparkle.addrori |
| 10               | imm              | rs2              | rs1              | 111              | rd               | 01010111         | sparkle.xorriri |

Instruction semantics.

- \( \text{sparkle.addrori rd, rs1, rs2, imm} \)

1. \( x \leftarrow \text{GPR[rs1]} \)
2. \( y \leftarrow \text{GPR[rs2]} \)
3. \( r \leftarrow x + (y \gg imm) \)
4. \( \text{GPR[rd]} \leftarrow r \)

- \( \text{sparkle.xorriri rd, rs1, rs2, imm} \)

1. \( x \leftarrow \text{GPR[rs1]} \)
2. \( y \leftarrow \text{GPR[rs2]} \)
3. \( r \leftarrow x ^\ (y \gg imm) \)
4. \( \text{GPR[rd]} \leftarrow r \)
### G.4 \( V_1^{32} \)

#### Instruction encoding.

| Instruction Memory Operation | 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  |
|------------------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sparkle.addrori.31 rs2, rs1, rs2 | 0100000 | rs2 | rs1 | 111 | rd  | 1111011 |
| sparkle.addrori.17 rs2, rs1, rs2 | 0100001 | rs2 | rs1 | 111 | rd  | 1111011 |
| sparkle.addrori.24 rs2, rs1, rs2 | 0100110 | rs2 | rs1 | 111 | rd  | 1111011 |
| sparkle.xorrori.31 rs2, rs1, rs2 | 0100111 | rs2 | rs1 | 111 | rd  | 1111011 |
| sparkle.xorrori.17 rs2, rs1, rs2 | 0101000 | rs2 | rs1 | 111 | rd  | 1111011 |
| sparkle.xorrori.24 rs2, rs1, rs2 | 0101001 | rs2 | rs1 | 111 | rd  | 1111011 |

#### Instruction semantics.

- **sparkle.addrori.31 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x + ( y >>> 31 )
4 GPR[rd] ← r
```

- **sparkle.addrori.17 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x + ( y >>> 17 )
4 GPR[rd] ← r
```

- **sparkle.addrori.24 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x + ( y >>> 24 )
4 GPR[rd] ← r
```

- **sparkle.xorrori.31 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x ^ ( y >>> 31 )
4 GPR[rd] ← r
```

- **sparkle.xorrori.17 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x ^ ( y >>> 17 )
4 GPR[rd] ← r
```

- **sparkle.xorrori.24 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x ^ ( y >>> 24 )
4 GPR[rd] ← r
```

- **sparkle.xorrori.16 rd, rs1, rs2**

```plaintext
1 x ← GPR[rs1]
2 y ← GPR[rs2]
3 r ← x ^ ( y >>> 16 )
4 GPR[rd] ← r
```
G.5 \( \nu_{2}^{32} \)

Instruction encoding.

<table>
<thead>
<tr>
<th>1000</th>
<th>imm</th>
<th>rs2</th>
<th>rs1</th>
<th>111</th>
<th>rd</th>
<th>1011011</th>
<th>sparkle.whole.enci.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>imm</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>1011011</td>
<td>sparkle.whole.enci.y</td>
</tr>
</tbody>
</table>

Instruction semantics.

- \textbf{sparkle.whole.enci.x rd, rs1, rs2, imm}

\begin{verbatim}
  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
\end{verbatim}

- \textbf{sparkle.whole.enci.y rd, rs1, rs2, imm}

\begin{verbatim}
  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
\end{verbatim}
H Additional ISE design detail: TinyJAMBU

H.1 $\mathcal{V}_0^{32}$

Instruction encoding.

| 00 | imm | rs2 | rs1 | 111 | rd | 0101011 | jambu.fsri |

Instruction semantics.

- \texttt{jambu.fsri rd, rs1, rs2, imm}
  
  1. $x_{hi} \leftarrow \text{GPR[rs2]}$
  2. $x_{lo} \leftarrow \text{GPR[rs1]}$
  3. $r \leftarrow (x_{hi} | x_{lo}) \gg imm$
  4. \text{GPR[rd]} \leftarrow r_{\{31..0\}}$

H.2 $\mathcal{V}_1^{32}$

Instruction encoding.

| 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 |

Instruction semantics.

- \texttt{jambu.fsr.15 rd, rs1, rs2}
  
  1. $x_{hi} \leftarrow \text{GPR[rs2]}$
  2. $x_{lo} \leftarrow \text{GPR[rs1]}$
  3. $r \leftarrow (x_{hi} | x_{lo}) \gg 15$
  4. \text{GPR[rd]} \leftarrow r_{\{31..0\}}$

- \texttt{jambu.fsr.6 rd, rs1, rs2}
  
  1. $x_{hi} \leftarrow \text{GPR[rs2]}$
  2. $x_{lo} \leftarrow \text{GPR[rs1]}$
  3. $r \leftarrow (x_{hi} | x_{lo}) \gg 6$
  4. \text{GPR[rd]} \leftarrow r_{\{31..0\}}$

- \texttt{jambu.fsr.21 rd, rs1, rs2}
  
  1. $x_{hi} \leftarrow \text{GPR[rs2]}$
  2. $x_{lo} \leftarrow \text{GPR[rs1]}$
  3. $r \leftarrow (x_{hi} | x_{lo}) \gg 21$
  4. \text{GPR[rd]} \leftarrow r_{\{31..0\}}$

- \texttt{jambu.fsr.27 rd, rs1, rs2}
  
  1. $x_{hi} \leftarrow \text{GPR[rs2]}$
  2. $x_{lo} \leftarrow \text{GPR[rs1]}$
  3. $r \leftarrow (x_{hi} | x_{lo}) \gg 27$
  4. \text{GPR[rd]} \leftarrow r_{\{31..0\}}$
I Additional ISE design detail: XooDYAK

1.1 $V_0^{32}$

Instruction encoding.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0101011</td>
<td>xooDYAK.xorrol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction semantics.

- $xooDYAK.xorrol \; rd, \; rs1, \; rs2$

1. $x \leftarrow \text{GPR}[rs1]$
2. $y \leftarrow \text{GPR}[rs2]$
3. $r \leftarrow (x \ll 5) \oplus (y \ll 14)$
4. $\text{GPR}[rd] \leftarrow r$
### J Additional evaluation results

Table 5: Results of software-oriented evaluation, i.e., utilisation of each ISE design: the per-algorithm results detail latency measured in clock cycles (plus overhead versus baseline in parentheses) associated with use of the AEAD API (i.e., encryption and decryption via `aead_encrypt` and `aead_decrypt`, using 16B plaintext, ciphertext, and associated data) as supported by the original and replacement kernel implementations.

<table>
<thead>
<tr>
<th>Submission</th>
<th>Functionality</th>
<th>Original kernel implementation</th>
<th>Replacement kernel implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RV32GC</td>
<td>RV32GC + Zbkbs/</td>
</tr>
<tr>
<td>ASCON</td>
<td>aead_encrypt</td>
<td>14401 (1.00×)</td>
<td>7839 (1.09×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>14523 (1.00×)</td>
<td>7962 (1.85×)</td>
</tr>
<tr>
<td>Elephant</td>
<td>aead_encrypt</td>
<td>3487678 (1.00×)</td>
<td>87608 (39.81×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>348769 (1.00×)</td>
<td>87608 (39.81×)</td>
</tr>
<tr>
<td>GIFT-COFB</td>
<td>aead_encrypt</td>
<td>118062 (1.00×)</td>
<td>6955 (16.97×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>118058 (1.00×)</td>
<td>6926 (17.05×)</td>
</tr>
<tr>
<td>GIFT-COFB</td>
<td>aead_encrypt</td>
<td>118062 (1.00×)</td>
<td>7957 (14.84×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>118058 (1.00×)</td>
<td>7919 (14.91×)</td>
</tr>
<tr>
<td>Grain-128AEADv2</td>
<td>aead_encrypt</td>
<td>19411 (1.00×)</td>
<td>19225 (1.03×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>15389 (1.00×)</td>
<td>14988 (1.03×)</td>
</tr>
<tr>
<td>PHOTON-Beetle</td>
<td>aead_encrypt</td>
<td>1407143 (1.00×)</td>
<td>203088 (6.93×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>1407754 (1.00×)</td>
<td>203254 (6.93×)</td>
</tr>
<tr>
<td>Romulus (TB)</td>
<td>aead_encrypt</td>
<td>161068 (1.00×)</td>
<td>33251 (4.84×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>161103 (1.00×)</td>
<td>33453 (4.82×)</td>
</tr>
<tr>
<td>Romulus</td>
<td>aead_encrypt</td>
<td>29668 (1.00×)</td>
<td>36613 (0.81×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>30093 (1.00×)</td>
<td>36458 (0.83×)</td>
</tr>
<tr>
<td>SPARKLE</td>
<td>aead_encrypt</td>
<td>131441 (1.00×)</td>
<td>5829 (2.26×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>131466 (1.00×)</td>
<td>5818 (2.20×)</td>
</tr>
<tr>
<td>TinyJAMBU</td>
<td>aead_encrypt</td>
<td>79528 (1.00×)</td>
<td>6418 (1.18×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>7978 (1.00×)</td>
<td>6761 (1.18×)</td>
</tr>
<tr>
<td>Xoddvay</td>
<td>aead_encrypt</td>
<td>57766 (1.00×)</td>
<td>4191 (13.78×)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>57775 (1.00×)</td>
<td>4200 (13.76×)</td>
</tr>
</tbody>
</table>
Table 6: Results of software-oriented evaluation, i.e., utilisation of each ISE design: the per-algorithm results detail latency measured in clock cycles (plus overhead versus baseline in parentheses) associated with use of the AEAD API (i.e., encryption and decryption via `aead_encrypt` and `aead_decrypt`, using 1024 B plaintext, ciphertext, and associated data) as supported by the original and replacement kernel implementations.

<table>
<thead>
<tr>
<th>Submission</th>
<th>Functionality</th>
<th>Original kernel implementation</th>
<th>Replacement kernel implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RV32GC</td>
<td>RV32GC + Zkbb/x</td>
</tr>
<tr>
<td>Ascon</td>
<td>aead_encrypt</td>
<td>270239 (1.00x)</td>
<td>228119 (1.18x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>271905 (1.00x)</td>
<td>230828 (1.17x)</td>
</tr>
<tr>
<td>Elephant</td>
<td>aead_encrypt</td>
<td>109520728 (1.00x)</td>
<td>2749081 (39.84x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>109520700 (1.00x)</td>
<td>2746736 (39.87x)</td>
</tr>
<tr>
<td>GIFT-COFB (BS)</td>
<td>aead_encrypt</td>
<td>5221431 (1.00x)</td>
<td>322859 (16.21x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>5220757 (1.00x)</td>
<td>322841 (16.17x)</td>
</tr>
<tr>
<td>GIFT-COFB (FS)</td>
<td>aead_encrypt</td>
<td>5221431 (1.00x)</td>
<td>312881 (16.69x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>5220757 (1.00x)</td>
<td>312008 (16.73x)</td>
</tr>
<tr>
<td>Grain-128AEADv2</td>
<td>aead_encrypt</td>
<td>664938 (1.00x)</td>
<td>650688 (1.02x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>655304 (1.00x)</td>
<td>642831 (1.02x)</td>
</tr>
<tr>
<td>PHOTOS-Beetle</td>
<td>aead_encrypt</td>
<td>61215512 (1.00x)</td>
<td>8718676 (1.22x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>61215428 (1.00x)</td>
<td>8717466 (1.22x)</td>
</tr>
<tr>
<td>Romulus (TB)</td>
<td>aead_encrypt</td>
<td>7587976 (1.00x)</td>
<td>1609969 (1.74x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>7579477 (1.00x)</td>
<td>1605325 (4.73x)</td>
</tr>
<tr>
<td>Romulus (FS)</td>
<td>aead_encrypt</td>
<td>1282582 (1.00x)</td>
<td>1442906 (0.89x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>1287633 (1.00x)</td>
<td>1441150 (0.89x)</td>
</tr>
<tr>
<td>SPARKLE</td>
<td>aead_encrypt</td>
<td>185117 (1.00x)</td>
<td>783116 (2.36x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>185202 (1.00x)</td>
<td>78346 (2.36x)</td>
</tr>
<tr>
<td>TinyJAMBU</td>
<td>aead_encrypt</td>
<td>299603 (1.00x)</td>
<td>248522 (1.19x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>299601 (1.00x)</td>
<td>251993 (1.19x)</td>
</tr>
<tr>
<td>XOOGYAK</td>
<td>aead_encrypt</td>
<td>1307332 (1.00x)</td>
<td>985774 (13.26x)</td>
</tr>
<tr>
<td></td>
<td>aead_decrypt</td>
<td>1306193 (1.00x)</td>
<td>98744 (13.50x)</td>
</tr>
</tbody>
</table>
Figure 3: A graph summarising the data in Figure 5, Figure 4, and Figure 6, focusing on encryption aead_encrypt: for each algorithm, we select the most efficient ISE variant (with respect to execution latency) and plot the number of cycles per byte needed across the parameterisations considered (i.e., 16 B, 128 B, and 1024 B associated data and plaintext/ciphertext) plus a comparison with the ISE-supported implementation of AES-GCM discussed.
Figure 4: A graph summarising the data in Figure 5, Figure 4, and Figure 6, focusing on decryption aead_decrypt: for each algorithm, we select the most efficient ISE variant (with respect to execution latency) and plot the number of cycles per byte needed across the parameterisations considered (i.e., 16 B, 128 B, and 1024 B associated data and plaintext/ciphertext) plus a comparison with the ISE-supported implementation of AES-GCM discussed.