RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography

Hao Cheng[1] Johann Großschädl[1]
Ben Marshall[2]
Dan Page[3] Thinh Pham[3]

[1] Department of Computer Science, University of Luxembourg.
{hao.cheng,johann.groszschaedl}@uni.lu
[2] PQShield Ltd, Oxford, UK.
ben.marshall@pqshield.com
[3] Department of Computer Science, University of Bristol.
{daniel.page,th.pham}@bristol.ac.uk

May 9, 2022

(Full disclosure: one of the authors is a member of the SPARKLE group [3])

Definition and Motivation

- Lightweight cryptography
  - “Cryptography tailored for resource-constrained devices” [10, Section 1]
  - Efficient on constrained hard/software platforms (vs. existing NIST standards)
  - Efficient for short messages
  - Amenable to countermeasures against implementation attacks
- Efficiency in software
  - Fast execution time, small code size, low RAM footprint, ...
  - Largely determined by instruction set (“HW/SW boudary”)
  - Idea: Customize/tailor instruction set for target algorithm
  - Goal: Improve SW efficiency at low HW overhead
- Instruction Set Extensions (ISEs) for cryptography
  - Omnipresent for AES and SHA2 family
  - Question: How efficient will ISE for NIST LWC standard be?

RISC-V

- Open Instruction Set Architecture (ISA)
  - Originally developed at UC Berkeley (2010)
  - Provided under open-source licenses (no fees!)
  - Based on well-established RISC principles
- Modular ISA design
  - Minimalist base ISA (RV32I) with only 40 instructions (no rotates!)
  - Optional extensions (many still in development)
  - E.g. extension for bit-manipulation (42 instr.) and scalar cryptography (49 instr.)
- Main differences to ARM
  - 32 instead of 16 integer registers
  - No condition codes (i.e. no add-with-carry, etc)
  - No flexible second operand (i.e. no implicit shift/rotate)
  - Fewer addressing modes, no multi-register load/store

RISC-V Standard Extension for Scalar Cryptography

- General-purpose cryptography instructions
  - Useful for a wide range of cryptographic algorithms
  - Instructions for rotations and permutations
  - Logic-with-negate instructions (e.g. `andn`), but no logic-with-shift/rotate
  - Carryless multiplication (e.g. AES-GCM)
- Special-purpose instructions
  - For particularly important algorithms
  - AES variants
  - SHA-256 and SHA-512
  - SM3 and SM4
- Entropy source interface
  - For generation of secret-key material
  - Full specification: `https://github.com/riscv/riscv-crypto/releases`

General ISE Design Principles

- Obey the wider RISC-V design principles
  - Support simple building-block operations
  - Instruction encodings must have at most *2 source and 1 destination* registers
  - Instruction length must be 32 bits
  - 3-register instruction can have 5-bit immediate value (optional)
- Use RISC-V scalar register file
  - Operands and result must be read from and written to general-purpose registers
- No special-purpose (micro-)architectural state
  - No special registers, caches, or scratch-pad memory
- Operation of instruction should be executed in 1 cycle in its HW module
  - Constant-time execution must be guaranteed

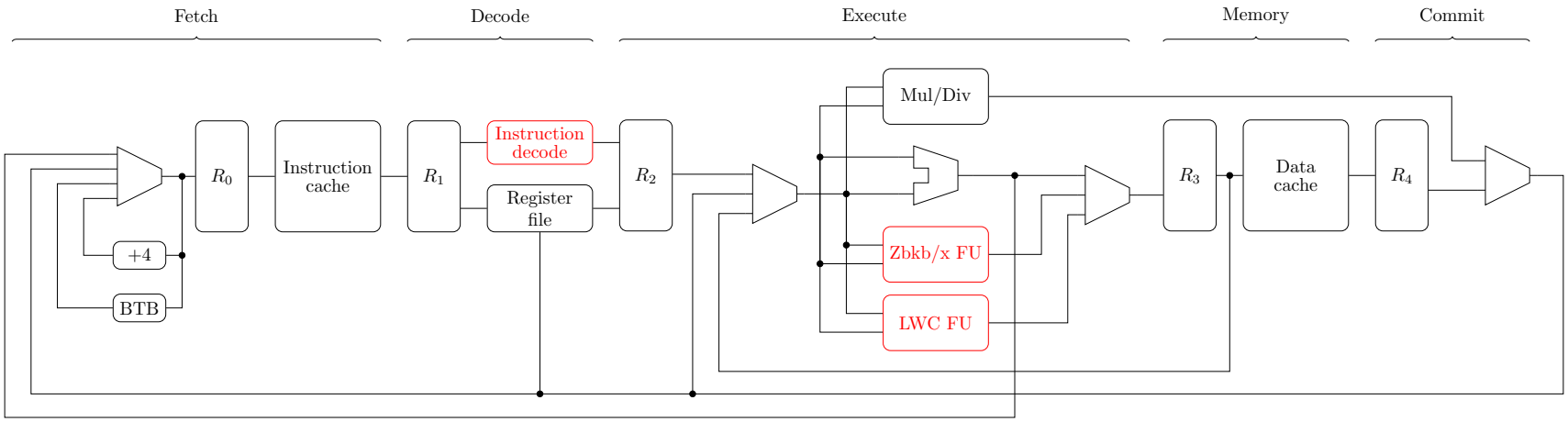
RISC-V Scalar Cryptography Extension: AES vs SHA2

- AES128 encryption
  - Most time-critical building block (“kernel”): round function
  - 2 custom instructions
    - ★ middle-round encryption: `aes32esmi rd, rs1, rs2, imm`
    - ★ final-round encryption: `aes32esi rd, rs1, rs2, imm`
  - Speed-up factor: 3.38x vs T-table AES on Rocket core
  - HW overhead factor: 1.06x vs base Rocket core (Kintex-7 FPGA)
  - Details: `https://tches.iacr.org/index.php/TCHES/article/view/8729`
- SHA256 hashing
  - Most time-critical building block (“kernel”): compression function
  - 4 custom instructions
    - ★ σ_0 and σ_1: `sha256sig0 rd, rs1` and `sha256sig1 rd, rs1`
    - ★ Σ_0 and Σ_1: `sha256sum0 rd, rs1` and `sha256sum1 rd, rs1`
  - Speed-up factor: roughly 2.0x on Rocket core
  - HW overhead factor: 1.07x vs base Rocket core (Kintex-7 FPGA)

ISE Design Flow

- Identify Kernel
  - Profiling tools can help to find the most performance-critical function
- Implement kernel in Assembly language using base ISA
  - Base ISA = RV32I + BitManip (Zbkb) + ScalarCrypto (Zbkx)
- Design custom instructions
  - Follow basic ISE design principles explained before
- Integrate custom instructions into toolchain
  - Assembler (GAS) and instruction-set simulator (Spike)
- HW design of functional unit and integration into RV32I core
  - FPGA prototype of Rocket core
- Implement kernel using ISE
- Detailed evaluation of performance, code size, HW-cost, ...

Integration of Custom Instructions into 32-bit Rocket Core



- HW implementation of ISE
  - Modification of instruction decoder
  - integration of ISE-specific functional unit into Rocket core
- Base RV32I Rocket core: 3303 LUTs (Xilinx Kintex-7 xc7k160tfbg676)
- Base core + BitManip (Zbkb) + ScalarCrypto (Zbkx): 3764 LUTs (1.14x)

LWC Finalists Overview

Name	Specification	AEAD	Hash	Component(s)
Grain128-AEAD	[9]	✓		Stream cipher
GIFT-COFB	[1]	✓		Block cipher
Romulus	[8]	✓	✓	(Tweakable) Block cipher
Ascon	[6]	✓	✓	Permutation
Elephant	[4]	✓		Permutation
PHOTON-Beetle	[2]	✓	✓	Permutation
Schwaemm & Esch	[3]	✓	✓	Permutation
Xoodyak	[5]	✓	✓	Permutation
ISAP	[7]	✓		Permutation
TinyJAMBU	[11]	✓		(Keyed) Permutation

LWC Finalists Overview

Name	Specification	AEAD	Hash	Component(s)
Grain128-AEAD	[9]	✓		L/NFSRs
GIFT-COFB	[1]	✓		GIFT-128
Romulus	[8]	✓	✓	Skinny-128-384+
Ascon	[6]	✓	✓	Ascon-p
Elephant	[4]	✓		Spongent-$\pi[n]$ or Keccak-$f[m]$
PHOTON-Beetle	[2]	✓	✓	$\texttt{PHOTON}_{256}$
Schwaemm & Esch	[3]	✓	✓	Sparkle (incl. Alzette ARX-box)
Xoodyak	[5]	✓	✓	Xoodoo
ISAP	[7]	✓		Ascon-p or Keccak-$f[m]$
TinyJAMBU	[11]	✓		P_n (incl. LFSR)

Kernels and Custom Instructions (1/2)

- Ascon
  - Kernel: `P[6,12]` → 2 custom instructions
- Elephant
  - Kernel: `permutation` → 3 custom instructions
- GIFT-COFB
  - Fix-sliced kernel: `giftb128` → 7 custom instructions
  - Bit-sliced kernel: `giftb128` → 2 custom instructions
- Grain-128AEADv2
  - Kernel: `grain_keystream32` → 10 custom instructions
- PHOTON-Beetle
  - Kernel: `PHOTON_Permutation` → 1 custom instruction

Kernels and Custom Instructions (2/2)

- Romulus
  - Table-based kernel: `Skinny_128_384_plus_enc` → 6 custom instructions
  - Fix-sliced kernel: `Skinny128_384_plus` → 8 custom instructions
- SPARKLE (Schwaemm + Esch)
  - Kernel: `sparkle_opt` → 4 custom instructions
  - Two alternative approaches: 9 and 4 custom instructions
- TinyJAMBU
  - Kernel: `state_update` → 1 custom instruction
  - Alternative approach: 4 custom instructions
- Xoodyak
  - Kernel: `Xoodoo_Permute_12rounds` → 1 custom instruction
- ISAP
  - Not implemented (same kernel as Ascon!)

Hardware-Oriented Evaluation of Kernels

Submission	Base core	Base core + Zbkb/x	Base core + Zbkb/x + $\mathcal{V}_0^{32}$	Base core + Zbkb/x + $\mathcal{V}_1^{32}$	Base core + Zbkb/x + $\mathcal{V}_2^{32}$
Ascon	3303 (1.000×)	3764 (1.140×)	4234 (1.282×)		
Elephant			3938 (1.192×)		
GIFT-COFB (BS)			3906 (1.183×)		
GIFT-COFB (FS)			4370 (1.323×)		
Grain-128AEADv2			4271 (1.293×)		
PHOTON-Beetle			3892 (1.178×)		
Romulus (TB)			3998 (1.210×)		
Romulus (FS)			4205 (1.273×)		
Sparkle			3998 (1.210×)	3986 (1.207×)	4483 (1.357×)
TinyJAMBU			3953 (1.197×)	3863 (1.170×)	
Xoodyak			3814 (1.155×)		

- Results are LUTs of Xilinx Kintex-7 FPGA (overhead factor in parentheses)
- Base core is RV32I Rocket core
- BitManip (Zbkb) and ScalarCrypto (Zbkx) introduce overhead of 1.14x

Software-Oriented Evaluation of Kernels (Latency + Code Footprint)

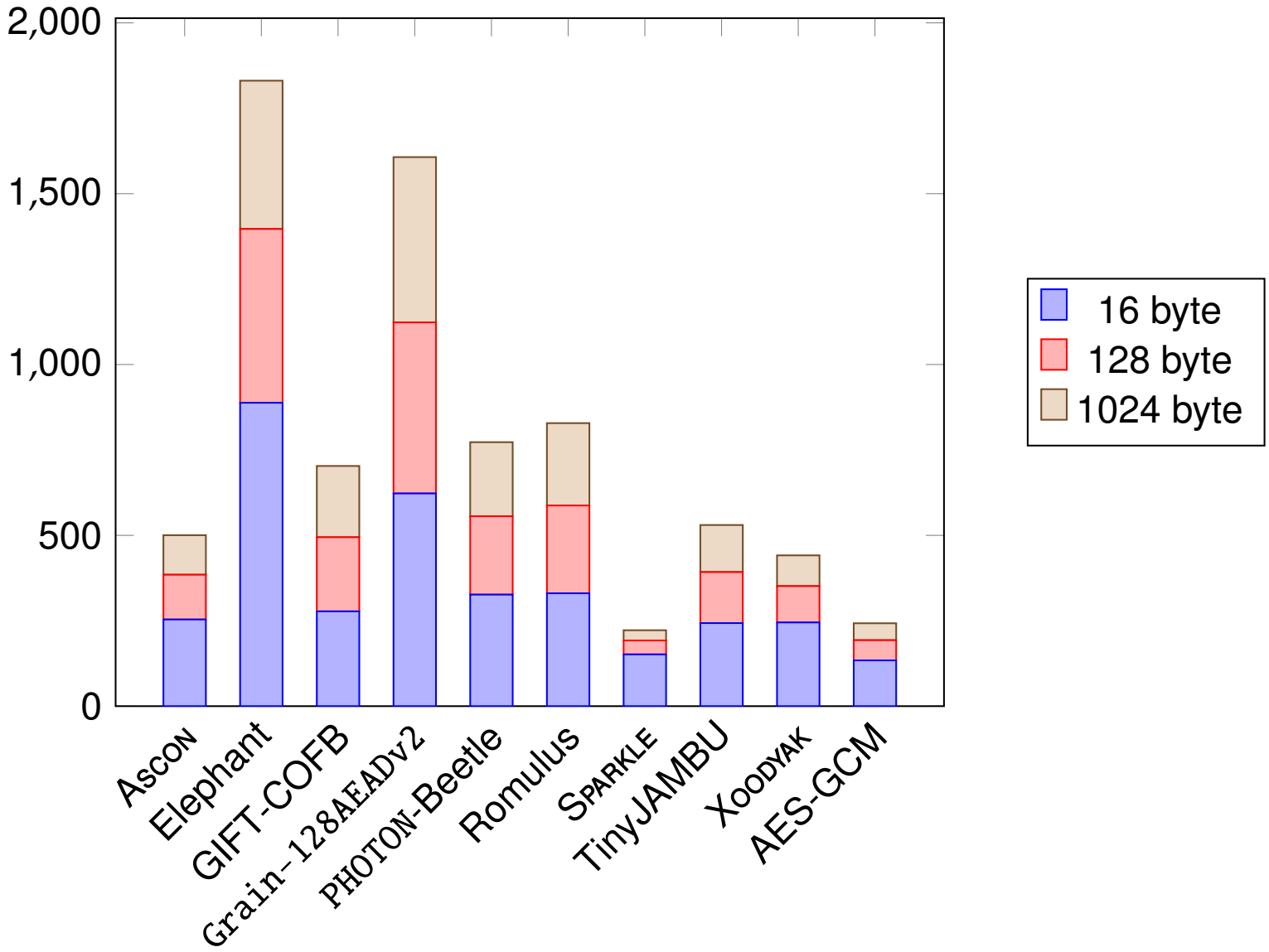
Submission	Kernel	Metric	Replacement kernel implementation: RV32GC + Zbkb/x	RV32GC + Zbkb/x + $\mathcal{V}_0^{32}$	RV32GC + Zbkb/x + $\mathcal{V}_1^{32}$	RV32GC + Zbkb/x + $\mathcal{V}_2^{32}$
Ascon	P6	latency	700 (1.00×)	280 (2.50×)		
		footprint	2718 (1.00×)	1050 (2.59×)		
Elephant	permutation	latency	15804 (1.00×)	1944 (8.13×)		
		footprint	25662 (1.00×)	7702 (3.33×)		
GIFT-COFB (BS)	giftb128	latency	1481 (1.00×)	641 (2.31×)		
		footprint	5770 (1.00×)	2410 (2.39×)		
GIFT-COFB (FS)	giftb128	latency	1386 (1.00×)	972 (1.43×)		
		footprint	4888 (1.00×)	3412 (1.43×)		
	precompute_rkeys	latency	1306 (1.00×)	251 (5.20×)		
		footprint	4830 (1.00×)	768 (6.29×)		
Grain-128AEADv2	grain_keystream32	latency	235 (1.00×)	86 (2.73×)		
		footprint	858 (1.00×)	262 (3.27×)		
PHOTON-Beetle	PHOTON_Permutation	latency	67035 (1.00×)	1473 (45.51×)		
		footprint	82486 (1.00×)	3466 (23.80×)		
Romulus (TB)	Skinny_128_384_plus_enc	latency	14268 (1.00×)	1502 (9.50×)		
		footprint	23508 (1.00×)	4612 (5.10×)		
Romulus (FS)	Skinny128_384_plus	latency	6208 (1.00×)	2156 (2.88×)		
		footprint	17402 (1.00×)	7274 (2.39×)		
	precompute_rtk1	latency	867 (1.00×)	200 (4.34×)		
		footprint	2814 (1.00×)	610 (4.61×)		
	precompute_rtk2_3	latency	3402 (1.00×)	1557 (2.18×)		
		footprint	11290 (1.00×)	5186 (2.18×)		
Sparkle	Sparkle_opt	latency	1647 (1.00×)	1185 (1.39×)	1185 (1.39×)	525 (3.14×)
		footprint	5908 (1.00×)	4456 (1.33×)	4456 (1.33×)	1816 (3.25×)
TinyJAMBU	state_update (P1024)	latency	575 (1.00×)	319 (1.80×)	319 (1.80×)	
		footprint	2208 (1.00×)	1184 (1.86×)	1184 (1.86×)	
Xoodyak	Xoodoo_Permute_12rounds	latency	873 (1.00×)	777 (1.12×)		
		footprint	3394 (1.00×)	3010 (1.13×)		

AEAD Results (Latency + Code Footprint) for 128 bytes of Data/AssocData

Submission	Functionality	Original kernel implementation	Replacement kernel implementation			
		RV32GC	RV32GC + Zbkb/x	RV32GC + Zbkb/x + $\mathcal{V}_0^{32}$	RV32GC + Zbkb/x + $\mathcal{V}_1^{32}$	RV32GC + Zbkb/x + $\mathcal{V}_2^{32}$
Ascon	aead_encrypt	43005 (1.00×)	32316 (1.33×)	16775 (2.56×)		
	aead_decrypt	43414 (1.00×)	32694 (1.33×)	17159 (2.53×)		
Elephant	aead_encrypt	16044010 (1.00×)	401543 (39.96×)	65118 (246.38×)		
	aead_decrypt	16044075 (1.00×)	402787 (39.83×)	65079 (246.53×)		
GIFT-COFB (BS)	aead_encrypt	687611 (1.00×)	42048 (16.35×)	27774 (24.76×)		
	aead_decrypt	687543 (1.00×)	42093 (16.33×)	27819 (24.71×)		
GIFT-COFB (FS)	aead_encrypt	687611 (1.00×)	41884 (16.42×)	33763 (20.36×)		
	aead_decrypt	687543 (1.00×)	41749 (16.47×)	33642 (20.44×)		
Grain-128AEADv2	aead_encrypt	87682 (1.00×)	85826 (1.02×)	64083 (1.37×)		
	aead_decrypt	86656 (1.00×)	84897 (1.02×)	63148 (1.37×)		
PHOTON-Beetle	aead_encrypt	8065027 (1.00×)	1149521 (7.02×)	29372 (274.58×)		
	aead_decrypt	8063672 (1.00×)	1150013 (7.01×)	29407 (274.21×)		
Romulus (TB)	aead_encrypt	1018364 (1.00×)	213180 (4.78×)	32880 (30.97×)		
	aead_decrypt	1017990 (1.00×)	213444 (4.77×)	33049 (30.80×)		
Romulus (FS)	aead_encrypt	177043 (1.00×)	203476 (0.87×)	40351 (4.39×)		
	aead_decrypt	177326 (1.00×)	203444 (0.87×)	41257 (4.30×)		
Sparkle	aead_encrypt	30033 (1.00×)	12883 (2.33×)	9724 (3.09×)	9721 (3.09×)	5218 (5.76×)
	aead_decrypt	30053 (1.00×)	12910 (2.33×)	9756 (3.08×)	9756 (3.08×)	5268 (5.70×)
TinyJAMBU	aead_encrypt	39851 (1.00×)	33574 (1.19×)	19118 (2.08×)	19118 (2.08×)	
	aead_decrypt	40432 (1.00×)	34033 (1.19×)	19562 (2.07×)	19562 (2.07×)	
Xoodyak	aead_encrypt	192338 (1.00×)	14579 (13.19×)	13616 (14.13×)		
	aead_decrypt	192149 (1.00×)	14397 (13.35×)	13429 (14.31×)		

Baseline is the optimized (resp. reference) implementation of the designers

AEAD Throughput (Cycles/Byte) for 16/128/1024 bytes of Data/AssocData



Concluding Remarks

- Main observations
  - HW-oriented candidates achieve a higher speed-up than SW-oriented ones
  - Nonetheless, SW-oriented candidates remain the top-performers
  - ISE narrow the gap between the fastest and slowest candidate
  - Only SPARKLE is able to outperform AES-GCM
- Source code
  - Available on GitHub: `https://github.com/scarv/lwise`
  - Feedback is welcome!
  - Proposals for better ISE are welcome!
  - ISE proposals have to follow design guidelines

References

[1] S. Banik et al. GIFT-COFB. Submission to NIST (version 1.1). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf. 2021.

[2] Z. Bao et al. PHOTON-Beetle. Submission to NIST. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf. 2021.

[3] C. Beierle et al. *Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the* SPARKLE *Permutation Family*. Submission to NIST (version 1.2). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf. 2021.

[4] T. Beyne et al. *Elephant*. Submission to NIST (version 2.0). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf. 2021.

[5] J. Daemen et al. *Xoodyak, a lightweight cryptographic scheme*. Submission to NIST (version 2.0). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf. 2021.

[6] C. Dobraunig et al. *Ascon*. Submission to NIST (version 1.2). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf. 2021.

[7] C. Dobraunig et al. *ISAP*. Submission to NIST (version 2.0). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/isap-spec-final.pdf. 2021.

[8] C. Guo et al. Romulus. Submission to NIST (version 1.3). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf. 2021.

[9] M. Hell et al. Grain-128AEADv2. Submission to NIST (version 2.0). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf. 2021.

[10] National Institute of Standards and Technology (NIST). *Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process*. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf. 2018.

[11] H. Wu and T. Huang. *TinyJAMBU*. Submission to NIST (version 2.0). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf. 2021.