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] University of Luxembourg [2] PQShield Ltd [3] University of Bristol

CHES 2023

Lightweight cryptography (LWC)

- Definition
  - “Cryptography tailored for resource-constrained devices”
  - Efficient on constrained hard/software platforms (vs. existing NIST standards)
  - Efficient for short messages
  - Amenable to countermeasures against implementation attacks
- NIST LWC standardization process
  - 08/2018: Call for Algorithms
  - 03/2021: Final Round begins (10 finalists)
  - 02/2023: NIST selects ASCON for standardization
  - At present: Drafting the LWC standard

Motivation

- Efficiency in software
  - Fast execution time, small code size, low RAM footprint, ...
  - Largely determined by instruction set ("HW/SW boundary")
  - Idea: Customize/tailor instruction set for target algorithm
  - Goal: Improve SW efficiency at low HW overhead
- Instruction Set Extensions (ISEs) for cryptography
  - Widespread availability for AES and SHA-2 (e.g., x86, ARM, POWER, RISC-V)
  - 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 integer instruction set (RV32I) with only 47 instructions (no rotates)
  - Optional extensions (many still in development)
  - E.g., extension for scalar cryptography (49 instr.)

RISC-V scalar cryptography extension

- General-purpose 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
- 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

LWC finalists overview

Submission	AEAD	Hash	Component(s)
Grain-128AEADv2	✔		Stream cipher
GIFT-COFB	✔		Block cipher
Romulus	✔	✔	(Tweakable) Block cipher
ASCON	✔	✔	Permutation
Elephant	✔		Permutation
ISAP	✔		Permutation
PHOTON-Beetle	✔	✔	Permutation
SCHWAEMM & ESCH	✔	✔	Permutation
TinyJAMBU	✔		(Keyed) Permutation
XOODYAK	✔	✔	Permutation

LWC finalists overview

Submission	AEAD	Hash	Component(s)
Grain-128AEADv2	✔		L/NFSRs
GIFT-COFB	✔		GIFT-128
Romulus	✔	✔	Skinny-128-384+
ASCON	✔	✔	ASCON-p
Elephant	✔		Spongent-$\pi[n]$ or Keccak-$f[m]$
ISAP	✔		ASCON-p or KECCAK-$f[m]$
PHOTON-Beetle	✔	✔	PHOTON_{256}
SCHWAEMM & ESCH	✔	✔	SPARKLE (incl. Alzette ARX-box)
TinyJAMBU	✔		P_n (incl. LFSR)
XOODYAK	✔	✔	XOODOO

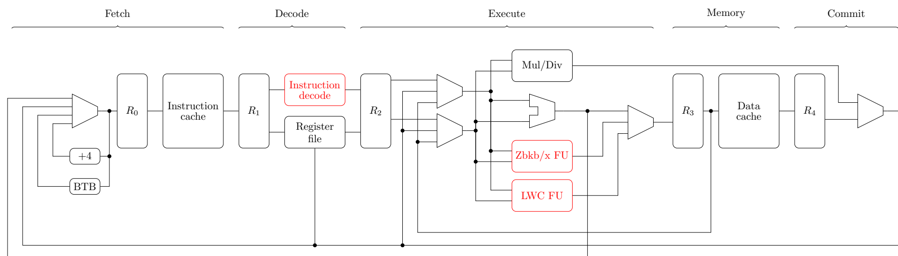
ISE design guidelines

- 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)
  - Disallow other approaches, e.g., tightly-coupled accelerator that processes entire state
- 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

ISE design flow

1. Identify kernel
   - Profiling tools can help to find the most performance-critical function
2. Implement kernel in assembly language using base ISA
   - Base ISA: RV32GC + Zbkb (crypto bitmanip) + Zbkx (crypto crossbar)
3. Design custom instructions
   - Follow basic ISE design guidelines explained before
4. Integrate custom instructions into toolchain
   - Assembler (GAS) and instruction-set simulator (Spike)
5. HW design of functional unit (LWC FU) and integration into base core
   - FPGA prototype of Rocket core
6. Implement kernel using ISE
7. Evaluate implementations (e.g., HW overhead, SW latency/throughput)

Extended 32-bit Rocket core



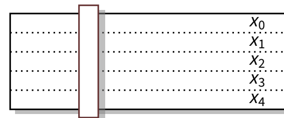
- Cores (on Xilinx Kintex-7 xc7k160tfbg676 FPGA)
  - **Unextended** core: RV32GC
  - **Base** core: RV32GC + Zbkb (crypto bitmanip) + Zbkx (crypto crossbar)
  - **Extended** core: RV32GC + Zbkb/x + LWC FU
- HW implementation of ISE
  - Modification of instruction decoder
  - Integration of ISE-specific functional unit into Rocket core

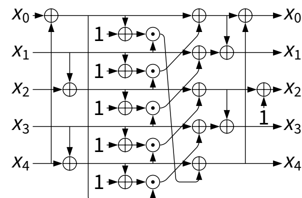
ISEs overview

Submission	Target AEAD	Kernel	# Instr.
Ascon	Ascon-128	P[6|12]	2
Elephant	Dumbo	permutation	3
GIFT-COFB (BS)	GIFT-COFB	giftb128	2
GIFT-COFB (FS)	GIFT-COFB	giftb128	7
Grain-128AEADv2	Grain-128AEADv2	grain_keystream32	10
ISAP	Isap-A-128a	Ascon_Permute_Nrounds	2
PHOTON-Beetle	PHOTON-Beetle-AEAD[128]	PHOTON_Permutation	1
Romulus (TB)	Romulus-N	Skinny_128_384_plus_enc	6
Romulus (FS)	Romulus-N	Skinny128_384_plus	8
Sparkle	Schwaemm256-128	Sparkle_opt	4
TinyJAMBU	TinyJAMBU-128	state_update	4
Xoodyak	Xoodyak	Xoodoo_Permute_12rounds	1

ISE for Ascon

- Addition of constants
  - 1 instr.
- S-Box
  - RV32GC: 17 instr. for half of the state
  - + Zbkb/x: 15 instr. (`andn`, `orn`) for half of the state
  - No custom instruction can help further





ISE for Ascon

- Linear layer
  - RV32GC (+ Zbkb/x): 16 instr. for one 64-bit word
  - 2 custom instr. for one $X_i := X_i \oplus (X_i \ggg m) \oplus (X_i \ggg n)$
    - Define the look-up tables for constant distances

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

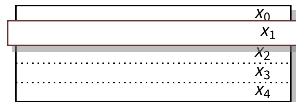
    - ascon.sigma.lo rd, rs1, rs2, imm

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

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

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

  - Bit-Interleaving (BI) has adverse impact due to conversions



$$X_0 := X_0 \oplus (X_0 \ggg 19) \oplus (X_0 \ggg 28)$$
$$X_1 := X_1 \oplus (X_1 \ggg 61) \oplus (X_1 \ggg 39)$$
$$X_2 := X_2 \oplus (X_2 \ggg 1) \oplus (X_2 \ggg 6)$$
$$X_3 := X_3 \oplus (X_3 \ggg 10) \oplus (X_3 \ggg 17)$$
$$X_4 := X_4 \oplus (X_4 \ggg 7) \oplus (X_4 \ggg 41)$$

Hardware-oriented evaluation

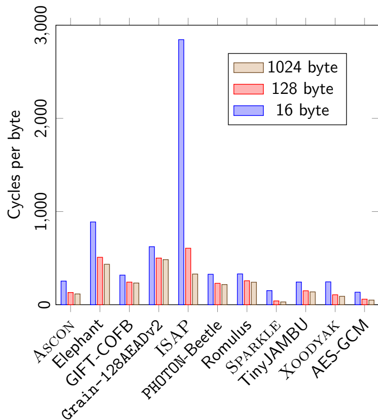
15% – 32% more LUTs (14% from Zbkb/x)

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

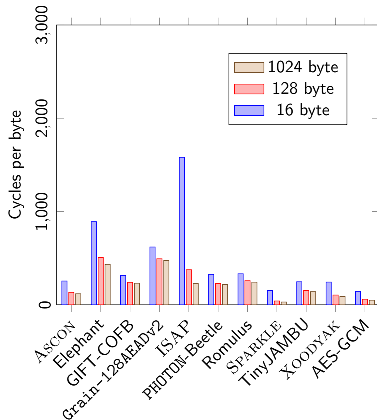
Software-oriented vertical evaluation (128-byte Data/AssocData)

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

Software-oriented horizontal evaluation (RV32GC + Zbkb/x + ISE)



(a) AEAD encryption throughput



(b) AEAD decryption throughput

Concluding remarks

- Main observations
  - Sort of open if/how ISE-based approaches should be accommodated in “official” platforms
  - Maybe should define a reference platform that could support work on ISEs
  - HW-oriented candidates achieve a higher speed-up than SW-oriented ones
  - Nonetheless, SW-oriented candidates remain the top-performers
- 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

Thank you for your attention!