Assembly or Optimized C for Lightweight Cryptography on RISC-V?

Fabio Campos[1], Lars Jellema[2], Mauk Lemmen[2], Lars Müller[1], Daan Sprenkels[2], and Benoit Viguier[2] *

[1] RheinMain University of Applied Sciences, Wiesbaden, Germany
campos@sopmac.de, mail@lars-mueller.com
[2] ICIS, Radboud University, Nijmegen, The Netherlands
M.Lemmen@student.ru.nl
lars.jellema@gmail.com
daan@dsprenkels.com
b.viguier@cs.ru.nl

Abstract. A major challenge when applying cryptography on constrained environments is the trade-off between performance and security. In this work, we analyzed different strategies for the optimization of several candidates of NIST's lightweight cryptography standardization project on a RISC-V architecture. In particular, we studied the general impact of optimizing symmetric-key algorithms in assembly and in plain C. Furthermore, we present optimized implementations, achieving a speed-up of up to 81% over available implementations, and discuss general implementation strategies.

Keywords: RISC-V · lightweight cryptography · software optimization · NIST.

1 Introduction

The enormous growth of the "Internet of Things" (IoT) is changing the world. Forecasts [30] project the number of interconnected embedded devices to around 50 billion worldwide by 2030, a five-fold increase in the next ten years. Driven by the lack of cryptographic algorithms which are more suitable for such constrained environments, NIST started in 2015 a project[1] (NIST-LWC) to solicit, evaluate, and eventually standardize lightweight authenticated encryption algorithms with associated data (AEAD) and hashing. In August 2019, NIST selected 32 candidates for round 2, which is expected to last one year. Lightweight cryptography (LWC), a sub-field of cryptography, covers cryptographic algorithms intended for use in constrained hardware and software environments. The main

* Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported in part by Continental AG; Elektrobit Automotive GmbH; and the European Commission through the ERC Starting Grant 805031 (EPOQUE). Date: December 6, 2020

[1] https://csrc.nist.gov/Projects/lightweight-cryptography

goal of NIST's project is to provide algorithms that are more suitable for use on constrained devices where the performance of current NIST cryptographic standards is not acceptable. Thereby, performance figures should be considered on a wide range of 8-bit, 16-bit and 32-bit microcontroller architectures.

On the hardware side, we are facing challenges where critical vulnerabilities ([27,29]) cannot be tracked back due to the lack of transparency. The RISC-V project, with roots in academia and research (University of California, Berkeley), has initiated a fundamental shift in the technical and business models for microprocessors. RISC-V [36], a royalty-free and open-source reduced instruction-set architecture (ISA), provides a competitive advantage and the required degree of flexibility to develop secure microprocessors with addresses of 32-, 64-, and 128-bits in length.

Contribution of this paper. This paper aims at comparing optimization at different levels of round-2 NIST lightweight candidates algorithms on a RISC-V architecture. To achieve this, we first present optimized RISC-V implementations of several cryptographic algorithms. Further, we study the impact of implementing these primitives on RISC-V in assembly compared to in C. Based on this, general implementation strategies are derived and discussed.

Related work. Many aspects regarding the optimization of lightweight cryptographic algorithms have been studied in the literature. In [31], generic security, efficiency, and some considerations for cryptographic design of lightweight constructions were explored. Cruz, Reis, Aranha, López, and Patil [17] discussed techniques for efficient and secure implementations of lightweight encryption on ARM devices. The modular and reusable architecture of RISC-V facilitates a variety of designs for the implementation of accelerators, ranging from loosely [35] to tightly coupled designs [2,25]. However, only few works focused on the optimization of cryptographic algorithms on the standard RISC-V instruction set. Stoffelen [34] presented the first optimized assembly implementations of AES, ChaCha, and the Keccak-f[1600] permutation for the RISC-V instruction set. In [32] the 32 second round finalists from the NIST-LWC were evaluated on RISC-V without further optimization.

Organization of this paper. This paper is structured as follows. Section 2 provides background information on the RISC-V 32-bit architecture and instruction set. We also give the necessary background on the platforms used for benchmarking. In Section 3, we briefly recall the selected algorithms and present our optimization strategies, before we describe the benchmarking setup and discuss the achieved results in Section 4. Finally, in Section 5, we conclude the paper.

Availability of implementations. We place all software and hardware implementations described in this paper into the public domain to maximize reusability of our results. They are available at `https://github.com/AsmOptC-RiscV/Assembly-Optimized-C-RiscV`.

2 RISC-V

In Section 2.1 we describe in more details the RISC-V 32-bit architecture before detailing the associated instruction set (Section 2.2). We then discuss different approaches to execute code targeting RISC-V platform (Section 2.3).

2.1 Architecture

The RISC-V architecture uses 32 32-bit registers numbered from `x0` through `x31`. To ease their use, they also have aliases. `zero` (`x0`) is hard-wired to the value 0; `ra` (`x1`) corresponds to the return address; `sp` (`x2`) to the stack pointer; `gp` (`x3`) to the global pointer; `tp` (`x4`) to the thread pointer. `a0-a7` (`x10-x17`) are function arguments with `a0` and `a1` also functioning as return values. `s0-s11` (`x8-x9`, `x18-x27`) are saved registers. Finally, `t0-t6` (`x5-x7`, `x28-x31`) are temporary registers.

The caller has the responsibility for the saved registers `s0-s11` while the callee is able to freely modify the arguments (`a0-a7`) and temporary registers (`t0-t6`).

Excluding the `zero`, `ra`, `sp`, `gp`, and `tp` registers, we are left with 27 freely usable 32-bit registers. This is twice of what is available in the Cortex-M3 and Cortex-M4 architectures; and it enables us to easily take care of register allocation.

2.2 Instruction set

The RISC-V base instruction set contains a small number of instructions which we briefly describe here.

Bitwise and arithmetic instructions such as `add`, `addi`, `and`, `andi`, `or`, `ori`, `sub`, `xor`, `xori` take three register operands, or if postfixed by `i`, two registers and one 12-bit sign-extended immediate.

We soon notice missing instructions. *e.g.*, `mov rd, rs` is implemented by taking advantage of the zero register as `add rd, zero, rs`. Similarly, the two's complement negation `neg rd, rs` is replaced by `sub rd, zero, rs` and the one's complement negation `not rd, rs` as `xori rd, rs, -1`. Subtract immediate (`subi`) is written as `addi` with a negative immediate.

The base ISA does not provide rotation instructions but logical and arithmetic shifts: `sll`, `slli`, `srl`, `srli`, `sra`, and `srai`. Those instructions are read as shift [left|right] [logical|arithmetic].

Load of constants is done with two instructions: `lui` and `addi`. Load upper immediate `lui` takes a 20 bit unsigned immediate and places it in the upper 20 bits of the destination register. The lowest 12 bits are filled with zeros.

```
.equ UART_BASE, 0x40003000

        lui a0,      %hi(UART_BASE)
        addi a0, a0, %lo(UART_BASE)
```

In order to load words, half-words (unsigned), or bytes (unsigned) from memory, the instructions `lw`, `lh`, `lhu`, `lb`, `lbu` are used. Similarly `sw`, `sh`, `shu`, `sb`, `sbu` are available to store values into the memory. For example `lw a5, 8(a2)` will load into `a5` the word located at address `a2 + 8`. Note that the offset has to be a constant. Additionally loads and stores of words have to be 32-bit aligned, *e.g.*, `lw a5, 3(a2)` will fail.

Text labels are used as targets for branches, unconditional jumps and symbol offsets. They are added to the symbol table of the compiled module. Numeric labels are used for local references. When used in jumps and similar instructions, they are suffixed with 'f' for a forward reference or 'b' for a backwards reference.

```
loop:                          1:
        ...                            ...
        j loop                         j 1b

        j func                         j 2f
        ...                            ...
fun:                           2:
```

In addition to the `jal` and `jalr` unconditional jump –relative to the program counter or as an absolute address in a register– the instruction `beq`, `blt`, `bltu`, `bge`, `bgeu` are used for conditional jumps. They take three arguments, the first two are used in the comparison while the third one is the destination –label– encoded later as an offset relative to the program counter.

To perform our benchmarks we use the `csrr` instruction (control and status register) to read the 64-bit cycle-counter. On the RV32I architecture, it is split into two 32-bit words (`cycle` and `cycleh`).

2.3 Executing code

To write optimized code for a specific architecture, we need ways to measure improvements or regressions. Below, we describe 3 test platforms which allowed us to benchmark our code.

SiFive E31 core. We use a HiFive1 development board. They are easily available and contain the FE310-G000 SoC with an E31 core. The CPU implements the RV32IMAC instruction set. This corresponds to the RV32I base ISA with the extensions for multiplications, atomic instructions and compressed instructions.

It has to be noted that RISC-V does not specify how many cycles an instruction may take or the kind of memory used. As a result benchmarks between different RISC-V cores have to be carefully compared.

The E31 runs at 320+ MHz and uses a 5-stage single-issue in-order pipeline. Additionally it uses a 16KB, 2-way instruction cache. Fetching an instruction from the cache takes only 1 cycle. Most instruction execution takes 1 cycle with a few exceptions. For example, if there is a cache hit, load word (`lw`) takes 2 cycles, loads of half word (`lh`) and bytes (`lb`) a 3-cycle latency. In the case of a cache miss, the latency is highly dependent on the flash controller's clock frequency. To prevent such unpredictability, we fill up the cache before any benchmarks.

The E31 comes with a 1-cycle latency branch predictor. It uses a 28-entry branch target buffer (BTB), a 512-entry branch history table (BHT) for the direction of conditional branches, and a 6-entry return-address stack (RAS). A correctly predicted control-flow instruction results in no penalty while mispredictions incur a 3-cycle penalty.

The RISC-V specification requires a 64-bit cycle counter accessible via two CSR registers which we will use to benchmark code. Occasionally measurements may end up taking much longer than expected, we ignore these odd values.

VexRiscv simulator. VexRiscv is a 32-bit RISC-V CPU implementation written in SpinalHDL. Although it is possible to load the core onto an FPGA; we use the Verilator simulator to emulate a core and flash binaries to it. This process allows us to have cycle counts and to evaluate how each algorithm is performing.

The core features the RV32IM instruction set. This corresponds to the base ISA with the extension for multiplications. We initialize the simulator with 256 KiB of RAM and 128 KB of sRAM.

Similarly to the E31 core, the VexRiscv makes use of a 5-stage pipeline. The absence of a branch predictor and an instruction cache give a significant advantage to algorithms which have been unrolled either by hand or the compiler. This explains major cycle-counts differences in the execution of different implementations of a same algorithm.

riscvOVPsim simulator. Finally, as opposed to executing code on a board or on a fully simulated core, we use the Open Virtual Platforms developed by Imperas Software, Ltd. Their RISC-V simulator uses Just-in-Time Code Morphing and executes RISC-V code on a Linux or Windows host computer.

This simulator implements the full Instruction Set and permits us to enable or disable specific extensions such as Vector instructions or Bit manipulations. The B extension gives us access to more advanced instructions such as rotations (`rori`, `roli`), packing (`pack`, `packu`), and many others.

Unfortunately, this approach simulates neither pipeline nor cache. While it allows us to execute RISC-V binary files, the results may be biased towards some optimization practices, leading to significant differences between implementations as shown in our benchmarks (see Section 4).

3 Optimized Algorithms

Optimized cryptographic implementation are usually written directly in assembly with the idea to prevent the compiler from introducing bugs or weaknesses. By making sure we do not branch on secret data and considering the small size of the RISC-V ISA, we trust the compiler to match our implementations.

We call "Optimized C" the translation of an assembly implementation back into C, making use of `uint32_t` such that the C code mimics the assembly instructions. The underlying idea is to have the compiler further optimize our code and take care of the register allocation.

We now describe the algorithms we optimized and some of the implementation strategies we used.

3.1 Gimli

Gimli [7] is a lightweight scheme proposed by Bernstein, Kölbl, Lucks, Massolino, Mendel, Nawaz, Schneider, Schwabe, Standaert, Todo, and Viguier. It makes use of a sponge construction and is based on a 384-bit permutation. Its design puts an emphasis on cross-platform performance and simplicity. The code is compact and uses only logical operations and shifts. The absence of additions allows to "interleave" implementations for platform with different register size than 32 bits. An implementation for RISC-V-64 with the B extension would likely be using such strategy.

The permutation. The 24-round permutation operates on a 384-bit state seen as a 3×4 matrix of 32-bit words. Gimli works first locally on the four 96-bit columns; and, to ensure diffusion through the full state, a 2-word swap is applied on the upper 128-bit row of the state every 2 rounds. The symmetries in the state are broken by the addition of an incrementing round constant every 4 rounds.

Using a sponge construction [9], the designers created two variations: a hash function Gimli-Hash and an authenticated cipher Gimli-Cipher.

Gimli-Hash and Gimli-Cipher. Gimli-Hash initializes a 48-byte state to all-zero before reading sequentially through a variable-length input as a series of 16-byte input blocks. Each full 16-byte input block is absorbed into the state. The final non-full (empty or partial) block is padded with a byte `0x01` before its absorption while a domain separation byte `0x01` is XORed in the last byte (47^{th}) of the state. The 32-byte digest output is extracted by blocks of 16-bytes. Each absorption or extraction of blocks is interweaved with calls to the Gimli permutation.

After initializing the state with a nonce and a key, Gimli-Cipher processes the additional data in the same way as Gimli-Hash. The message is processed in a similar fashion with the exception that after each absorption of a block, the modified first 16 bytes of the state are produced as cipher text. Once the last non-full block is processed; the 16-byte authentication tag is generated from the first 16 bytes of the state.

We are able to get speed-ups on both Gimli-Hash and Gimli-Cipher by optimizing the underlying permutation Gimli. We rescheduled the order of instructions to avoid swap operations.

Bounds and optimizations. We optimize Gimli by first having a deeper look at the inner permutation and by computing the lower bound of the number of cycles used. Gimli's state representation uses twelve 32-bit words which are easily contained in the 27 general-use 32-bit registers. [7] shows that only 2 additional registers are required in order to compute the column operations; as a result, in a fully unrolled implementation, the only cycles necessary in the computation are the ones required by the logical operations.

Gimli uses 2 rotations, 6 XORs, 2 ANDs, 1 OR, and 4 shifts. All logical operations have a latency of 1 cycle, except for rotates which have a 3-cycle latency. A column operation requires thus 19 cycles; iterated over 4 columns and 24 rounds, this totals to 1824 cycles.

Gimli uses 6 constants (loaded in 2 cycles) derived every 4 rounds (an additional 5 cycles) before being XORed into the state (6 XORs, thus 6 cycles). When the permutation is not directly inlined and used as a function, it requires 12 loads and 12 stores to get the state into registers for an additional 48 cycles. Excluding the cycles needed to preserve some of the callee registers, we have a total of 1885 cycles.

As a base line, the reference C code runs at 2178 cycles. By using careful scheduling of the instructions, and using a minimum number of register – saving into the stack only 4 callee–, our assembly implementation runs at 2092 cycles. The Optimized C version runs in 2132 cycles. This timing difference is explained by the compiler's use of the 12 callee registers, inducing a 40-cycle penalty.

By unrolling in C –the same approach could have been applied in assembly– over 8 rounds and propagating the swapping by renaming variable to avoid move operations, the compiler manages to achieve further speed-ups by getting down to 1900 cycles. Using this last implementation, we get a 19% speed-up for Gimli-Hash and Gimli-Cipher (Table 1).

Table 1. Cycle counts for different Gimli implementations on the SiFive board; Gimli-Hash over 128 bytes of data, Gimli-Cipher over a 128 bytes message with 128 bytes of associated data. Compiled with Clang-10 and `-O3`

	C-ref	Assembly	Optimized C	8-round Optimized C
Gimli	2178	2092 (−4%)	2132 (−2%)	1900 (−13%)
Gimli-Hash	23120	20812 (−10%)	21055 (−9%)	18678 (−19%)
Gimli-Cipher	44423	39583 (−10%)	40816 (−8%)	35853 (−19%)

3.2 Sparkle

Sparkle [4] is a family of cryptographic permutations based on the block cipher Sparx [22]. Schwaemm (an AEAD cipher scheme) and Esch (a hash function) follow a not hermetic design approach, and share Sparkle as the underlying permutation. The Sparkle permutation is a classic ARX design, which, unlike most ARX constructions, provides security guarantees with regard to differential and linear cryptanalysis based on the long trail strategy (LTS) [22]. Schwaemm and Esch work on a relatively small state, which is only 256 bits for the most lightweight instance of Schwaemm and 384 bits for the lightest variant of Esch. The biggest possible state size with 512 bits, can be applied by both schemes. Both algorithms employ the sponge construction.

Two instances for hashing were proposed in [4], i.e., Esch256 and Esch384, which allow to process messages of arbitrary length and output a digest of 256 bit, and 384 bit, length, respectively. Esch256, the main instance of Esch and the one considered in our work, uses the 384-bit Sparkle permutation and has a claimed security level of 128 bits.

All the four instances for authenticated encryption with associated data proposed in [4], i.e., Schwaemm128-128, Schwaemm256-128, Schwaemm192-192 and Schwaemm256-256 use a variation of the Beetle mode of operation first presented in [16], which in turn is based on the duplexed sponge construction. We focus again on the main version Schwaemm256-128, which uses the 384 bit Sparkle (Sparkle384) permutation, with a rate of $r = 256$ bit and a capacity of $c = 128$ bit, claiming a security level of 120 bits.

Sparkle384 requires 50 rotations, 68 XORs, 24 ADDs, and 2 shifts for a single round. With the exception of rotation (3 cycles), all operations have a latency of 1 cycle. Thus, iterated over 7 or 11 rounds this totals to an estimated lower bound of 1708 cycles, and 2684 cycles respectively. For further details, we refer to the specification [4].

Loop unrolling. Although unrolling the main loop within the Sparkle permutation over 7 or 11 rounds results in a significant speed-up (see Table 3) when using instruction cache (like the SiFive core used in this work, see Section 2.3), this leads to significantly worse results in the case of AEAD (see Table 2).

Round Constants. In this optimization, we speed-up the permutation by increasing the space required. In every round of the permutation, each of the six ARX-boxes uses the same round constant in their computations. The idea is to avoid the loading of the constants for the ARX-boxes in every round by loading and saving these 6 constants in the registers before the transformation. This comes with the cost of dedicating 6 registers to these constants.

This optimization can be applied in the loop as well as in the unrolled variant of the implementation. In the unrolled implementation, we further reduce the loading of round constants, since these 6 constants are also being used as the round constants that are added to the state every round. In the 7-round variant of the permutation, we save the loading of the first 6 round constants and only have to load the 7th constant. In the 11-round variant of the permutation, we only have to load the 8th constant extra. The other three are already loaded because there are only 8 round constants defined and the selection index is calculated modulo 8. In the loop unrolled implementation we reduce the instruction count for 7 rounds by 72 instructions and for 11 rounds by 126 instructions.

Table 2 shows the achieved speed-up for Schwaemm256-128, Table 3 presents the achieved results for Esch256.

3.3 Saturnin

Saturnin [15] is the NIST lightweight candidate designed by Canteaut, Duval, Leurent, Naya-Plasencia, Perrin, Pornin, and Schrottenloher. By building on top of a 256-bit block cipher with a 256-bit key, they describe three constructions for hashing (Saturnin-Hash) and authenticated encryption of small (Saturnin-Short) and large data segment (Saturnin-Cipher). This last AEAD scheme uses the counter mode and a separate MAC.

We ported to our benchmark platform the reference implementation and both the 32-bit optimized “bs32” and “bs32x” C implementations [15, Section 3.4.2].

Table 2. Cycle counts for different Schwaemm256-128 implementations on the SiFive board; encryption over a 128 bytes of message with 128 bytes of associated data.

platform	Compiler	Opt.	Opt. C	looped + round cst ASM	loop-unrolled Opt. C
SiFive	Clang-10	`-O3`	72286	43877 (−40%)	1059813 (+94%)
SiFive	Clang-9	`-O3`	73387	44558 (−40%)	1709958 (+95%)
SiFive	GCC	`-O3`	71271	42634 (−40%)	1790566 (+95%)
riscvOVPsim	Clang-10	`-O3`	20842	20840 (±0%)	20277 (−3%)
riscvOVPsim	Clang-9	`-O3`	20842	20840 (±0%)	20277 (−3%)
riscvOVPsim	GCC	`-O3`	20762	20161 (−2%)	20010 (−3%)
VexRiscv	GCC	`-O2`	25464	27018 (+6%)	24769 (−3%)

Table 3. Esch256 cycle counts on each platform. The hashing operation hashes 128 bytes of data.

platform	Compiler	Opt.	Opt. C	loop-unrolled Opt. C
SiFive	Clang-10	`-O3`	62734	34664 (−44%)
SiFive	Clang-9	`-O3`	63893	28952 (−54%)
SiFive	GCC	`-O3`	58193	33331 (−42%)
riscvOVPsim	Clang-10	`-O3`	17439	16552 (−5%)
riscvOVPsim	Clang-9	`-O3`	17439	16552 (−5%)
riscvOVPsim	GCC	`-O2`	17849	17231 (−3%)
VexRiscv	GCC	`-O2`	18874	17753 (−6%)

The "bs32" and "bs32x" implementations both implement Saturnin in a 32× bitsliced fashion. Their difference is that "bs32" bitslices *inside* of blocks, whereas "bs32x" bitslices *across* blocks. When comparing the two bitsliced implementations, "bs32" showed a consistently better performance than the other, albeit sometimes with a small margin. We decided that "bs32" would be the preferred implementation to use on our platforms.

In all the implementations, we tweaked the code to make sure that any constants would be loaded from SRAM, instead of (the relatively slow) SPI flash. This considerably improved the performance of the bitsliced implementations.

In the end, we see that the Optimized C implementation is considerably faster than the reference implementation in terms of performance, with generally a speed-up by a factor of 2. Another interesting property from the results in Tables 4 and 5 is the performance stability of the implementations across compilers. Where the "bs32" performance is very stable—with cycle counts generally varying less than 10%—the performance of the reference implementation varies a lot with different compiler versions. Nonetheless, we see that newer compiler versions seem to produce faster code.

Table 4. Saturnin-Hash cycle counts on each platform. The hashing operation hashes 128 bytes of data.

platform	Compiler	Opt.	Ref.	bs32
SiFive	Clang-10	`-O3`	49433	28199 (−43%)
SiFive	Clang-9	`-O3`	52868	30483 (−42%)
SiFive	GCC	`-O3`	78110	30321 (−61%)
riscvOVPsim	Clang-10	`-O3`	46946	27070 (−42%)
riscvOVPsim	Clang-9	`-O3`	48785	27972 (−43%)
riscvOVPsim	GCC	`-O3`	76211	29030 (−61%)
VexRiscv	GCC	`-O2`	103325	32169 (−69%)

Table 5 illustrates the fact that the greedy unrolling and inlining by GCC with `-O3` results in major speed-up on simulators. However once tested on a physical device such as the SiFive development board (2.3), this results in a code too large for the 16KB cache, inducing in a slowdown by a factor of 5.

Table 5. Saturnin-Cipher cycle counts on each platform. The cipher encrypts 128 AD bytes and 128 message bytes.

platform	Compiler	Opt.	Ref.	bs32	bs32x
SiFive	Clang-10	`-O3`	121651	59368 (−51%)	68792 (−43%)
SiFive	Clang-9	`-O3`	106665	62743 (−41%)	91511 (−14%)
SiFive	GCC	`-O3`	151428	60817 (−60%)	5210541 (×34)
SiFive	GCC	`-Os`	183464	65469 (−64%)	138187 (−24%)
riscvOVPsim	Clang-10	`-O3`	93184	55154 (−41%)	61077 (−34%)
riscvOVPsim	Clang-9	`-O3`	96540	55617 (−42%)	63767 (−33%)
riscvOVPsim	GCC	`-O3`	145734	57366 (−61%)	75646 (−48%)
VexRiscv	GCC	`-O2`	202226	65015 (−68%)	88278 (−56%)

3.4 Ascon

Ascon [23] is a scheme proposed by Dobraunig, Eichlseder, Mendel and Schläffer. It uses a very small 320-bit state which allows it to fit in registers on most systems. The authors introduce multiple variants of Ascon AEAD as well as a hashing scheme. We focus our efforts on the Ascon-128 AEAD variant. We expect that our results translate fairly well to the other variants and the hashing scheme as they are very similar.

We use the Ascon C [3] repository as a base line, more specifically we use the reference, the 64-bit optimized, and the 32-bit interleaved implementations as starting point for our optimizations.

Improved formula. First we optimize the inner permutation by improving the Ascon S-box formula (Figure 1). We reduce the number of required instructions from 22 to 17 and the number of temporary registers from 5 to 3 at the cost of less potential for parallelism. Instruction-level parallelism —such as out-of-order execution— is common in high-end CPUs but not so common in lightweight platforms like our RISC-V targets. This optimization gives us a 10% speed-up for both the assembly and Optimized C implementations (Table 6).

Fig. 1. These formulas compute the Ascon S-box in 17 operations (once duplicate operations are taken out); o_n indicates output bit n and i_n indicates input bit n.

$$
\begin{aligned}
o_0 &= i_3 \oplus i_4 \oplus (i_1 \vee (i_0 \oplus i_2 \oplus i_4)) \\
o_1 &= i_0 \oplus i_4 \oplus ((i_1 \oplus i_2) \vee (i_2 \oplus i_3)) \\
o_2 &= i_1 \oplus i_2 \oplus (i_3 \vee \neg i_4) \\
o_3 &= i_1 \oplus i_2 \oplus (i_0 \vee (i_3 \oplus i_4)) \\
o_4 &= i_3 \oplus i_4 \oplus (i_1 \wedge \neg(i_0 \oplus i_4))
\end{aligned}
$$

Table 6. Cycle counts for the different Ascon's round functions over 6 rounds; Compiled with Clang-10 and `-O3`

Platforms	C-ref	Assembly	Optimized C
SiFive	832	750 (−10%)	750 (−10%)
riscvOVPsim	830	748 (−10%)	748 (−10%)

Bit Interleaving. We also compare the C implementation optimized for 32-bit interleaving. It performs the worst of all others including the baseline implementation. Bit interleaving allows 32-bit rotations to model 64-bit rotations efficiently, unfortunately our targets does not support 32-bit rotations. We expect this implementation will perform better when targeting RISC-V cores comes with the B extension, which adds rotation instructions.

Optimized 64 bits. Finally, we compare the C implementation optimized for 64-bit processors. On RISC-V cores without the B extension, the 64-bit operations are compiled to 32-bit operations in a straightforward manner and the compiler has no trouble with it. As RISC-V does not support misaligned memory access, we had to modified the code to handle the authentication tag.

While on the RISC-V OVP simulator the 64-bit optimized version is 7% faster than the baseline, testing it on the SiFive board reveals significant slowdowns due to the code not fitting in the 16KB instruction cache.

Our final implementation makes use of the improved S-box formula in a 6-round unrolled Optimized C permutation. By folding the processing of associated data and message we are able to reuse the code and have to compiled code fit in the instruction cache. Applying these modifications, we achieve our best results: 15% faster than the baseline.

Table 7. Cycle counts for different Ascon implementations in OVP sim for encrypting 128 bytes of message and 128 bytes of associated data; compiled with Clang-10 and `-O3`

Implementation	OVP sim	SiFive
ref. & default permutation	31990	32038
ref. & asm permutation	28988 (−9%)	29036 (−9%)
ref. & inlined Optimized C perm.	27489 (−14%)	27703 (−14%)
bit interleaved inline permutation	32001 (±0%)	1559691 (×49)
opt. 64-bit & default unrolled perm.	29646 (−7%)	1191702 (×37)
opt. 64-bit & asm permutation	29090 (−9%)	29170 (−9%)
opt. 64-bit & fully unrolled Opt. C perm.	27589 (−14%)	809631 (×25)
opt. 64-bit & 6-round unrolled Opt. C perm.	27184 (−15%)	27271 (−15%)

3.5 Delirium

Elephant [12] is a family of lightweight authenticated encryption schemes. The mode of Elephant is a nonce-based encrypt-then-MAC construction, where encryption is performed using counter mode based on permutation masked using LFSRs. One of the instances of Elephant is Elephant-Keccak-$f[200]$, also called Delirium, which uses Keccak as its permutation primitive. Delirium has a state size of 200 bits and claimed a security level of 127 bits. We optimize Delirium by exploiting Elephant's possibility for parallelization by using bit-interleaving.

Bit Interleaving. In order to make full use of the 32-bit registers, we combine four blocks of byte-sized elements into one block of 4-byte elements. Thus, we can process four blocks at the same time and our state representation changes to an array of 25 32-bit words (5-by-5-by-32) with a total size of 800 bits. In this new representation, one block amounts to four blocks in the standard representation.

There are two possible cases when transforming blocks before encrypting/decrypting to the new representation. The first and the easiest case is when the amount of blocks that need to be transformed is a multiple of four. This means that all groups of four blocks consisting of 8-bit words can be interleaved to make one block of 32-bit words. The second case is when the amount of blocks is not a multiple of four. Since the new representation needs four "old" blocks to transform into one new block, we have to use padding blocks filled with zero values to add to make the amount of blocks to a multiple of four.

After encryption/decryption, when transforming back to a byte representation of the data, we have to de-interleave each interleaved 32-bit block back to four blocks of bytes. Since it is possible that the amount of original blocks was not a multiple of four, we need to make sure none of the data from the added padding blocks gets joined in the output data. This can be done by cutting off any output data which exceeds the message-length variable.

As shown in Table 8, we note that shorter inputs perform worse in the optimized implementation. This is because the effort of interleaving data to process four blocks simultaneously is wasted if there are very few blocks to process.

Table 8. Cycle counts for different Elephant-Keccak-f[200] implementations on the SiFive board; encryption over a 32/64/128 bytes message with 32/64/128 bytes of associated data.

platform	Compiler	message length	data length	C-ref	bit interleaved
SiFive	GCC	16	16	66541	73989 (+11%)
SiFive	GCC	32	32	91837	74385 (−19%)
SiFive	GCC	64	64	143181	74890 (−47%)
SiFive	GCC	128	128	245100	113031 (−53%)
SiFive	Clang-9	128	128	294643	160494 (−46%)
SiFive	Clang-10	128	128	241975	145936 (−40%)
riscvOVPsim	GCC	32	32	64651	66690 (+3%)
riscvOVPsim	GCC	64	64	102138	66805 (−35%)
riscvOVPsim	GCC	128	128	176086	101966 (−42%)
riscvOVPsim	Clang-9	128	128	168313	106904 (−36%)
riscvOVPsim	Clang-10	128	128	163973	103631 (−37%)

3.6 Xoodyak

Xoodyak [20], based on the Xoodoo permutation [19,18], is a cryptographic scheme that is suitable for several symmetric-key functions, including hashing, encryption, MAC computation and authenticated encryption. Xoodoo, according to its authors [19], can be seen as a porting of the Keccak-p [10,11] design approach to a Gimli-shaped [7] state.

Xoodoo iteratively applies 12 rounds to a 384-bit state, which can be treated as 3 horizontal planes, each one consisting of 4 parallel 32-bit lanes. The choice of 12 rounds justifies a security claim in the hermetic philosophy. The claimed security strength for Xoodyak is 128 bits.

An estimated lower bound for cycles taken by Xoodoo can be calculated as follows. It requires 24 rotations, 37 XORs, 12 ANDs, and 12 NOTs for a single round. With the exception of rotation (3 cycles), all operations take 1 cycle. Thus, iterated over 12 rounds this totals to 1596 cycles.

Lane Complementing. The idea behind lane complementing, first proposed in the Keccak implementation overview [11], is to reduce the number of NOT instructions by complementing certain lanes before the transformation.

In Xoodoo the state is ordered in 4 sheets, each containing 3 lanes with a width of 32-bit. The χ layer computes 3 XOR, 3 AND and 3 NOT operations for every sheet in the state. This sums up to 12 NOT operations per round and 144

NOT operations in total. In the default case, the χ transformation for every lane $a[i]$ in a sheet, with $0 \leq i \leq 2$ and index calculation mod 3, can be calculated as shown in equation (1).

$$a[i] \leftarrow a[i] \oplus (\overline{a[i+1]} \wedge a[i+2]) \tag{1}$$

For example, we now want to complement lane $a[2]$. Thus, the equation of lane $a[0]$ gets rearranged as follows:

$$a[0]' = a[0] \oplus (\overline{a[1]} \wedge \overline{\overline{a[2]}}) = a[0] \oplus \overline{(a[1] \vee \overline{a[2]})} = \overline{a[0] \oplus (a[1] \vee \overline{a[2]})},$$
$$\overline{a[0]'} = a[0] \oplus (a[1] \vee \overline{a[2]}).$$

The complementation of $a[2]$ results in the cancellation of the negation of $a[1]$, the switch from an AND to an OR operation and the complement of $a[0]'$. Now we calculate all three lanes of a sheet with the complement of the lane $a[2] \leftarrow \overline{a[2]}$:

$$\begin{aligned} a[0] \leftarrow \overline{a[0]'} &= a[0] \oplus (a[1] \vee a[2]), \\ a[1] \leftarrow a[1]' &= a[1] \oplus (a[2] \wedge \overline{a[0]}), \\ a[2] \leftarrow \overline{a[2]'} &= a[2] \oplus (a[0] \wedge a[1]). \end{aligned}$$

It can be observed that we only need one complementation for this sheet, instead of three. For the computation of $a[1]$, $a[0]$ is complemented to be positive, because $a[0]$ was negated before. This example of lane complementing comes with the cost of applying the input mask $\overline{a[2]}$ and output mask $\overline{a[0]}$, $\overline{a[2]}$.

The possible transformations of the boolean equations for a sheet are not fixed to one. Thus, there are multiple boolean equations that are still logically congruent, but may differ in the input and output mask. We want to find the boolean equations and input mask with the lowest possible number of NOT-instructions. To simplify this problem, we set the boolean equations to a fixed set and only care about the possible input patterns. Therefore, we employ an algorithm for finding the minimum NOT instruction count for a certain set of boolean equations. We test all 2^{12} possible combinations of input masks. For every input mask, we follow the complements propagation through the 12 rounds of the permutation as a symmetric difference pattern in the state and count the NOT instructions.

After the application of the algorithm, we obtain an input mask and a sequence of boolean equations. This input mask is 2-round invariant, meaning that the input mask is the always same after every two rounds. Hence, it can be implemented as a loop and therefore have a smaller code size. The obtained input mask P is the following (denoted in x, y coordinates):

$$P = \{(0,0), (1,0), (2,0), (3,0)\}.$$

We reduce the number of NOT operations to exactly 33% over 12 rounds. The application of our input and output mask, each costs 4 NOT operations. Due to a larger number of lanes in KECCAK, Stoffelen [34] achieved a reduction to 20%.

Lane complementing is not an assembly-specific optimization. As shown in tables 9 to 11, we achieve a very similar speed-up in assembly and in C.

Table 9. Cycle counts for different implementations of Xoodyak in AEAD mode GCC compiled with `-O2` in riscvOVPsim for encrypting 128 bytes of message and 128 bytes of associated data.

Implementation	riscvOVPsim	Relative
reference	105463	
loop unrolled + lane complementing assembly	29574	−71%
loop unrolled + lane complementing Optimized C	28672	−72%

Table 10. Cycle counts for Xoodyak in hash mode on each platform, compiled with `-O3`. The hashing operation hashes 128 bytes of data.

platform	Compiler	Ref.	unrolled & lane comp.
SiFive	Clang-10	81349	17963 (−78%)
SiFive	Clang-9	88451	18865 (−79%)
SiFive	GCC	82741	17063 (−79%)
riscvOVPsim	Clang-10	18114	16845 (−7%)
riscvOVPsim	Clang-9	18059	16898 (−6%)
riscvOVPsim	GCC	23247	16614 (−29%)
VexRiscv	GCC `-O2`	261678	38378 (−85%)

Table 11. Cycle counts for Xoodyak in AEAD mode on each platform, compiled with `-O3`, for encrypting 128 bytes of message and 128 bytes of associated data.

platform	Compiler	Ref.	unrolled & lane comp.
SiFive	Clang-10	103717	26246 (−75%)
SiFive	Clang-9	112414	27392 (−76%)
SiFive	GCC	103522	23238 (−78%)
riscvOVPsim	Clang-10	25002	23429 (−6%)
riscvOVPsim	Clang-9	25002	23429 (−6%)
riscvOVPsim	GCC	29775	21668 (−28%)
VexRiscv	GCC `-O2`	261678	38378 (−85%)

3.7 AES

In [34], Stoffelen proposes two assembly implementations of AES: the first one is based on lookup tables, and the second one uses a bitsliced approach.

With a lookup table. When encrypting a single block of 16 bytes, multiple steps of the round function can be combined in a lookup table, also called T-table by Daemen and Rijmen in [21]. Note that this type of implementation is usually vulnerable to cache attacks [5,13,33]. Because none of our benchmarking platforms have a data cache, we believe this implementation is likely "safe" to use.

For his table-based implementation, Stoffelen makes use of the baseline instructions described in [8]. Most of the proposed optimization by Bernstein and Schwabe are not applicable due to the small instruction set of the RISC-V architecture. The translation from assembly to C using `uint32_t` to simulate registers is straightforward, and the lookup table is converted to an array as `uint32_t variable[]`.

Table 12. Cycle counts for the Assembly of [34] and its translation to C on the SiFive board, compiled with Clang-10 and `-O3`.

	Assembly	Optimized C
Key schedule	342	342
1-block encryption	903	901

Note that if the table is declared as `const`, the compiler will place it in the `.rodata` segment. While this change does not have any impact on the verilator and the riscvOVPsim simulators, it induces a major slowdown in the case of the SiFive board as the SPI flash is significantly slower than the SRAM.

In order to prevent the compiler from messing with the pointer arithmetic, data pointers are kept in the `uint8_t*` type. This forces us to cast the pointer to `uint32_t*` before de-referencing to trigger the compiler to use the `lw` instruction.

```
Y0 = RK[0]; T0 = (uint32_t*)(LUT1 + ((*X0 & 0xff) << 4)); Y0 = Y0 ^ *T0;
Y1 = RK[1]; T1 = (uint32_t*)(LUT1 + ((*X1 & 0xff) << 4)); Y1 = Y1 ^ *T1;
Y2 = RK[2]; T2 = (uint32_t*)(LUT1 + ((*X2 & 0xff) << 4)); Y2 = Y2 ^ *T2;
Y3 = RK[3]; T3 = (uint32_t*)(LUT1 + ((*X3 & 0xff) << 4)); Y3 = Y3 ^ *T3;
```

Listing 1.1. Code fragment of AES encryption

Using a bitsliced approach. When using AES in CTR or GCM mode, multiple blocks can be processed in parallel using a bitsliced implementation [26,28]. This strategy is often more efficient and avoids lookup tables, making the implementation more resistant against timing attacks.

By using the same approach as with lookup tables, we translate the assembly from [34] back into C. As seen in Table 13 the key schedule it is slightly slower. However this translation approach gives us a 4% speed-up in the case of the encryption in CTR mode (Table 13).

Table 13. Cycle counts for the Assembly of [34] and its translation to C on the SiFive board, compiled with Clang-10 `-O3`.

	Assembly	Optimized C
Key schedule	1248	1256
Encryption of 128 blocks	260695	249813 (-4%)

3.8 Keccak

We now have a look at the KECCAK-f family permutation—designed by Bertoni, Daemen, Peeters and Van Assche [10]—, more precisely its 1600-bit instance found in the SHA-3 standard by NIST[24]. The permutation is used in multiple cryptographic constructions including future post-quantum candidates such as FrodoKEM[14], NewHope[1], SPHINCS+[6] and others.

Stoffelen [34] provides us with another optimized implementation for RISC-V inspired by the *Keccak implementation overview*[11]. KECCAK-f[1600] works on a state composed of 25 64-bit lanes, in other words a total of 50 32-bits words. This is more than the number of register made available by the ISA, preventing the state from completely fitting in the registers. By using bit interleaving and other techniques, Stoffelen manages to reduce the number of cycles used.

Table 14. Cycle counts for the Assembly of Keccak [34] and its translation to C on the SiFive board, compiled with GCC `-Os`.

	Assembly	Optimized C
KECCAK-f[1600]	13731	13336 (-3%)

We take his implementation and translate it back to C. We compile with GCC and `-Os` instead of `-O2` or `-O3` to get slightly faster results than the assembly implementation in [34] (Table 14).

4 Comparison with other implementations and additional benchmark

Some other implementations of lightweight candidates are publicly available; we chose to compare our work against the repository of Weatherley[2] as their implemented are "focused on good performance in plain C on 32-bit embedded microprocessors".

As Clang-10 generally produces faster results than GCC with `-O3`, we used it to compile and benchmark every optimized C implementation provided by Weatherley. We measure the cycle counts for encryption of AEAD schemes for 128-byte messages with 128 bytes of associated data, and provide our results in Appendix A.2, Table 17.

In Table 15, we summarize the performance of our software and Weatherley's implementations.

While on the OVP simulator most of our implementations produces just slightly better results with an average at −4% cycle counts; when using the SiFive board, the unrolled implementation of Weatherley suffer heavily from the 16KB instruction cache. This makes our RISC-V-optimized code on average 47.5% faster.

[2] https://github.com/rweather/lightweight-crypto, commit `52c8281`

Table 15. Cycle counts for AEAD mode on the SiFive and riscvOVPsim platform, compiled with Clang-10 `-O3`, for encrypting 128 bytes of message and 128 bytes of associated data.

Algorithm	Weatherley		our results	
	OVP	SiFive	OVP	SiFive
Gimli	37596	38530	35690 (-5%)	35853 (-7%)
Schwaemm256-128	20842	72286	20277 (-3%)	43877 (-40%)
Saturnin	55367	152803	55154 (-1%)	59368 (-61%)
Ascon	41228	42562	27184 (-34%)	27271 (-36%)
Delirium	110171	765235	103631 (-6%)	145936 (-81%)
Xoodyak	18852	64869	23451 (+24%)	26246 (-60 %)

5 Conclusion

We described how multiple lightweight NIST candidates such as Gimli, Sparkle, Saturnin, Ascon, Delirium, and Xoodyak can be efficiently implemented. With strategies such as loop unrolling, we are able to write assembly code close to the lower bound given by the number instructions arithmetic. By translating our assembly implementation back into C, we get the compiler to further optimize our results.

Using the AES and Keccak assembly implementations from Stoffelen [34], we also show that our approach is applicable to existing code bases, and may provide slightly improved results while increasing the readability and maintainability of the code.

We use the HiFive1 development board to illustrate that algorithms need to be tested on physical devices in order to guarantee useful optimized implementations (Table 17). Although strategies such as fully unrolled loops may work nicely in simulated environments such as riscvOVPsim; they will fail at length on physical devices with *e.g.*, a 16KB instruction cache.

As the NIST lightweight competition is currently taking place, we hope our results will be found useful by the candidates' implementers and designers. On the other side, RISC-V offers the opportunity to disrupt the processor industry by using a very collaborative approach offering more interoperability and partnership opportunities.

References

1. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange: A new hope. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 327–343, USA, 2016. USENIX Association. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_alkim.pdf. 17
2. Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):219–242, Jun. 2020. https://tches.iacr.org/index.php/TCHES/article/view/8589. 2
3. Ascon C repository on GitHub. https://github.com/ascon/ascon-c. 10
4. Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Lightweight AEAD and Hashing using the Sparkle Permutation Family. *IACR Transactions on Symmetric Cryptology*, 2020(S1):208–261, Jun. 2020. https://tosc.iacr.org/index.php/ToSC/article/view/8627. 7, 8
5. Daniel J. Bernstein. Cache-timing attacks on AES, 2005. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 15
6. Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ Signature Framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3319535.3363229. 17
7. Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli: a cross-platform permutation. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, 2017. https://eprint.iacr.org/2017/630. 6, 13
8. Daniel J. Bernstein and Peter Schwabe. New AES Software Speed Records. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008*, pages 322–336, Berlin, Heidelberg, 2008. Springer. https://www.cryptojedi.org/papers/aesspeed-20080926.pdf. 16
9. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT Hash Workshop*, volume 2007, 2007. https://keccak.team/files/SpongeFunctions.pdf. 6
10. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer. https://link.springer.com/chapter/10.1007/978-3-642-38348-9_19. 13, 17
11. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, 2013. https://keccak.team/files/Keccak-implementation-3.2.pdf. 13, 17
12. Tim Beyne, Yu Long Chen, Christof Dobraunig, and Bart Mennink. Elephant v1, 2019. https://www.esat.kuleuven.be/cosic/elephant/. 12
13. Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks against AES. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'06, pages 201–215, Berlin, Heidelberg, 2006. Springer. https://doi.org/10.1007/11894063_16. 15

14. Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1006–1018, New York, NY, USA, 2016. Association for Computing Machinery. https://dl.acm.org/doi/abs/10.1145/2976749.2978425. 17
15. Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *IACR Transactions on Symmetric Cryptology*, 2020(S1):160–207, Jun. 2020. https://tosc.iacr.org/index.php/ToSC/article/view/8621. 8
16. Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):218–241, May 2018. https://tches.iacr.org/index.php/TCHES/article/view/881. 8
17. Rafael Cruz, Tiago Reis, Diego F. Aranha, and Harsh Kupwade Patil. Lightweight cryptography on ARM. In *NIST Lightweight Cryptography Workshop. NIST*, 2016. http://www.africacrypt.com/presentations/lw-arm-speed.pdf. 2
18. Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38, Dec 2018. https://tosc.iacr.org/index.php/ToSC/article/view/7359. 13
19. Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. Cryptology ePrint Archive, Report 2018/767, 2018. https://eprint.iacr.org/2018/767. 13
20. Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme. *IACR Transactions on Symmetric Cryptology*, 2020(S1):60–87, Jun. 2020. https://tosc.iacr.org/index.php/ToSC/article/view/8618. 13
21. Joan Daemen and Vincent Rijmen. The Design of Rijndael: AES - The Advanced Encryption Standard. In *Information Security and Cryptography*. Springer, 2002. 15
22. Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design Strategies for ARX with Provable Bounds: Sparx and LAX. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 484–513, Berlin, Heidelberg, 2016. Springer. https://link.springer.com/chapter/10.1007/978-3-662-53887-6_18. 7
23. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2, 2016. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf. 10
24. Morris J Dworkin. FIPS 202: SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, National Institute of Standards and Technology, 2015. https://doi.org/10.6028/NIST.FIPS.202. 17
25. Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. Cryptology ePrint Archive, Report 2020/446, 2020. https://eprint.iacr.org/2020/446. 2
26. Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, pages 1–17, Berlin, Heidelberg, 2009. Springer. https://link.springer.com/chapter/10.1007/978-3-642-04138-9_1. 16

27. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. https://ieeexplore.ieee.org/document/8835233. 2
28. Robert Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, pages 187–202, Berlin, Heidelberg, 2008. Springer. https://link.springer.com/chapter/10.1007/978-3-540-79263-5_12. 16
29. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf. 2
30. S. Liu. IoT connected devices worldwide 2030, 2019. https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/. 1
31. Nicky Mouha. The Design Space of Lightweight Cryptography. In *NIST Lightweight Cryptography Workshop 2015*, Gaithersburg, United States, Jul 2015. https://hal.inria.fr/hal-01241013/file/session5-mouha-paper.pdf. 2
32. Gorkem Nisanci, Remzi Atay, Meltem Kurt Pehlivanoglu, Elif Bilge Kavun, and Tolga Yalcin. Will the Future Lightweight Standard be RISC-V Friendly?, 2019. https://csrc.nist.gov/CSRC/media/Presentations/will-the-future-lightweight-standard-be-risc-v-fri/images-media/session4-yalcin-will-future-lw-standard-be-risc-v-friendly.pdf. 2
33. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer. https://www.cs.tau.ac.il/~tromer/papers/cache.pdf. 15
34. Ko Stoffelen. Efficient Cryptography on the RISC-V Architecture. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 323–340, Cham, 2019. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-30530-7_16. 2, 14, 15, 16, 17, 18
35. Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and Embedded Systems. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, pages 523–550, Cham, 2020. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-38471-5_21. 2
36. Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.2, 2017. 2

A Further Results

A.1 RISC-V Bitmanip (Bit Manipulation) Extension

In table 16, we present for some of the primitives the impact of using the RISC-V bit manipulation extension[3] (B). The provided instructions in this extension are a work in progress. Thus, some instructions and their specification may change before being accepted as a standard by the RISC-V Foundation. The presented performance figures were compiled using the *riscv-bitmanip* branch of the GCC compiler[4] using the flags `-O2 -mcmodel=medany -march=rv64gcb -mabi=lp64d` and calculated with the Spike RISC-V ISA Simulator[5].

Table 16. Cycle counts comparison for some primitives using the RISC-V bit manipulation extension.

primitive	variant	w/o B ext.	with B ext.
Gimli-Hash	C-ref	27628	22688
Gimli-Hash	C-optimized	26771	22080
Gimli-Hash	8-round C-optimized	22224	16618
Esch256	C-ref	20605	13891
Esch256	loop unrolled	17585	11586
AES LUT	C-optimized	3647	1578
AES CTR-Bitsliced	C-optimized	1509	1431
Saturnin-Hash	C-ref	83516	80866
Saturnin-Hash	bs32	33087	30943
Xoodyak-Hash	C-ref	28492	22440
Xoodyak-Hash	unrolled & complementing	19123	14169
Keccak-f[1600]	C-optimized	14633	12402
Keccak-f[200]	C-optimized	9143	6119

A.2 Benchmark of other implementation

RISC-V hardware availability is scarce and most implementations are based on simulators. Having the availability of both –board and simulator–, we benchmark the work of Rhys Weatherley[6] and present our measures in table 17. As such, we illustrate the need of testing implementations on boards and constrained environment.

[3] https://github.com/riscv/riscv-bitmanip, commit a05231d
[4] https://github.com/riscv/riscv-gcc/tree/riscv-bitmanip, commit 8b86205
[5] https://github.com/riscv/riscv-isa-sim, commit 958dcdc
[6] https://github.com/rweather/lightweight-crypto, commit 52c8281

Table 17: Cycle counts for different ciphers implementations clang-10 compiled with `-O3` for encrypting 128 bytes of message and 128 bytes of associated data.

Implementation	riscvOVPSim	SiFive	Relative
ACE_AEAD_ENCRYPT	206040	1959318	+851%
ASCON128_AEAD_ENCRYPT	41228	42562	+3%
ASCON128A_AEAD_ENCRYPT	28284	29457	+4%
ASCON80PQ_AEAD_ENCRYPT	41264	42639	+3%
COMET_128_CHAM_AEAD_ENCRYPT	21648	22570	+4%
COMET_64_SPECK_AEAD_ENCRYPT	24429	25515	+4%
COMET_64_CHAM_AEAD_ENCRYPT	60197	61552	+2%
DRYGASCON128_AEAD_ENCRYPT	96365	98194	+2%
DRYGASCON256_AEAD_ENCRYPT	120862	123710	+2%
ESTATE_TWEGIFT_AEAD_ENCRYPT	50051	855204	+1609%
DUMBO_AEAD_ENCRYPT	1365382	2927910	+114%
JUMBO_AEAD_ENCRYPT	1465147	3265908	+123%
DELIRIUM_AEAD_ENCRYPT	110171	763144	+593%
FORKAE_PAEF_64_192_AEAD_ENCRYPT	543575	818122	+51%
FORKAE_PAEF_128_192_AEAD_ENCRYPT	349468	550627	+58%
FORKAE_PAEF_128_256_AEAD_ENCRYPT	349485	510584	+46%
FORKAE_PAEF_128_288_AEAD_ENCRYPT	457833	642481	+40%
FORKAE_SAEF_128_192_AEAD_ENCRYPT	350409	523108	+49%
FORKAE_SAEF_128_256_AEAD_ENCRYPT	350767	512754	+46%
GIFT_COFB_AEAD_ENCRYPT	28277	28642	+1%
GIMLI24_AEAD_ENCRYPT	37596	38530	+2%
GRAIN128_AEAD_ENCRYPT	71378	71687	-0%
HYENA_AEAD_ENCRYPT	37317	136055	+265%
ISAP_KECCAK_128A_AEAD_ENCRYPT	243243	640729	+163%
ISAP_ASCON_128A_AEAD_ENCRYPT	204619	222540	+9%
ISAP_KECCAK_128_AEAD_ENCRYPT	1190410	2467483	+107%
ISAP_ASCON_128_AEAD_ENCRYPT	585890	605107	+3%
KNOT_AEAD_128_256_ENCRYPT	51298	224600	+338%
KNOT_AEAD_128_384_ENCRYPT	30982	102381	+230%
KNOT_AEAD_192_384_ENCRYPT	69190	228613	+230%
KNOT_AEAD_256_512_ENCRYPT	97648	266707	+173%
LOTUS_AEAD_ENCRYPT	84658	1006509	+1089%
LOCUS_AEAD_ENCRYPT	86693	1008575	+1063%
ORANGE_ZEST_AEAD_ENCRYPT	84917	159953	+88%
ORIBATIDA_256_AEAD_ENCRYPT	104129	106399	+2%
ORIBATIDA_192_AEAD_ENCRYPT	118296	121481	+3%
PHOTON_BEETLE_128_AEAD_ENCRYPT	157558	298030	+89%
PHOTON_BEETLE_32_AEAD_ENCRYPT	591344	1124306	+90%
PYJAMASK_128_AEAD_ENCRYPT	287809	316105	+10%
PYJAMASK_MASKED_128_AEAD_ENCRYPT	1407899	1602733	+14%

PYJAMASK_96_AEAD_ENCRYPT	284920	308484	+8%
PYJAMASK_MASKED_96_AEAD_ENCRYPT	1391787	1500329	+8%
ROMULUS_N1_AEAD_ENCRYPT	213113	218841	+3%
ROMULUS_N2_AEAD_ENCRYPT	197988	201165	+2%
ROMULUS_N3_AEAD_ENCRYPT	166744	309093	+85%
ROMULUS_M1_AEAD_ENCRYPT	282764	325705	+15%
ROMULUS_M2_AEAD_ENCRYPT	270063	291863	+8%
ROMULUS_M3_AEAD_ENCRYPT	231497	238422	+3%
SKINNY_AEAD_M1_ENCRYPT	248751	251707	+1%
SKINNY_AEAD_M2_ENCRYPT	248747	251677	+1%
SKINNY_AEAD_M3_ENCRYPT	248721	251578	+1%
SKINNY_AEAD_M4_ENCRYPT	248717	251548	+1%
SKINNY_AEAD_M5_ENCRYPT	211871	215308	+2%
SKINNY_AEAD_M6_ENCRYPT	211820	215179	+2%
SCHWAEMM_256_128_AEAD_ENCRYPT	20842	72286	+247%
SCHWAEMM_128_128_AEAD_ENCRYPT	23918	111207	+365%
SCHWAEMM_192_192_AEAD_ENCRYPT	28112	98569	+251%
SCHWAEMM_256_256_AEAD_ENCRYPT	30452	107680	+254%
SPIX_AEAD_ENCRYPT	93090	348608	+274%
SUNDAE_GIFT_0_AEAD_ENCRYPT	44214	57268	+30%
SUNDAE_GIFT_64_AEAD_ENCRYPT	45838	58952	+29%
SUNDAE_GIFT_96_AEAD_ENCRYPT	45874	58996	+29%
SUNDAE_GIFT_128_AEAD_ENCRYPT	45906	58996	+29%
SATURNIN_AEAD_ENCRYPT	55367	152798	+176%
SATURNIN_SHORT_AEAD_ENCRYPT	42	55	+31%
SPOC_128_AEAD_ENCRYPT	69789	839487	+1103%
SPOC_64_AEAD_ENCRYPT	113672	1651442	+1353%
SPOOK_128_512_SU_AEAD_ENCRYPT	34778	285992	+722%
SPOOK_128_384_SU_AEAD_ENCRYPT	46390	195868	+322%
SPOOK_128_512_MU_AEAD_ENCRYPT	34792	285764	+721%
SPOOK_128_384_MU_AEAD_ENCRYPT	46409	195708	+322%
SUBTERRANEAN_AEAD_ENCRYPT	127707	132338	+4%
TINY_JAMBU_128_AEAD_ENCRYPT	34776	37952	+9%
TINY_JAMBU_192_AEAD_ENCRYPT	37590	40789	+9%
TINY_JAMBU_256_AEAD_ENCRYPT	40394	43865	+9%
WAGE_AEAD_ENCRYPT	788038	14336632	+1719%
XOODYAK_AEAD_ENCRYPT	18852	64869	+244%