Lightweight Cryptography with Ascon in `riscv64`

Paulo Pacitti *Julio López*

Technical Report - IC-PFG-23-41 - Relatório Técnico
December - 2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Lightweight Cryptography with Ascon in riscv64

Paulo Pacitti[*] Julio López[†]

Abstract

Sed quis lorem magna. Sed sit amet ullamcorper massa, sit amet placerat lectus. Suspendisse pulvinar ipsum sed enim commodo, ac malesuada lectus finibus. Aliquam eu eros eleifend, interdum nisi faucibus, viverra sapien. Vivamus lobortis a lectus eu rutrum. Quisque in est sit amet libero sollicitudin ornare a sed ipsum. Suspendisse potenti. Aliquam sit amet nisi sed nulla tincidunt imperdiet. Pellentesque elementum lacus eget dolor gravida lobortis. Sed placerat lacinia nisi, sed varius turpis facilisis ac.

1 Introduction

Inspired by the works of the UNICAMP's Laboratory of Security and Cryptography in the optimization of cryptographic algorithms for the ARM architecture [1], the NIST Lightweight Cryptography competition winner algorithm [2], and the RISC-V open architecture, this research aims to explore the Ascon family of algorithms [3] on the RISC-V 64-bit architecture and whether it's possible to optimize it for this architecture. There are other works that have explored the Ascon family of algorithms [4] that explores the Ascon algorithm in RISC-V, but in the 32-bit version and benchmarking in an FPGA chip. It was used in this paper for benchmarking a 64-bit RISC-V chip, the Allwinner D1.

The approach was to analyse the Ascon algorithm design and 3 different implementations. All the implementations tested are written in C. The first implementation `ref` is the reference implementation of Ascon, written by Ascon team [5]. The second one is `opt64`, an optimized implementation for a generic 64-bit architecture system, also developed by the Ascon team. The third implementation was the main objective of this research, named `ascon-v` [6], this implementation is focused on producing an optimized version for the RISC-V 64-bit architecture. The research was focused on trying to improve the basic blocks of the Ascon family of algorithms. Because of that, the analysis, optimizations, and results are focused on the Ascon-128, which is the *de facto* AEAD standard of the Ascon family.

2 Ascon

Ascon is a family of algorithms for lightweight cryptography, designed to be used in constrained environments, like embedding computing. Designed by cryptographers from Graz University of Technology, Infineon Technologies, Intel Labs, and Radboud University, Ascon has been selected as the new standard for lightweight cryptography in the 2019–2023 NIST Lightweight Cryptography competition. The Ascon family is mainly composed by 4 algorithms: Ascon-128, Ascon-128a, Ascon-Hash and Ascon-Hasha. There's also variants Ascon-80pq, Ascon-Xof, Ascon-Xofa,

[*]Institute of Computing, UNICAMP. `p185447@dac.unicamp.br`
[†]Associate Professor, Institute of Computing, UNICAMP. `jlopez@ic.unicamp.br`

Name	**Algorithms**	**key**	**nonce**	**tag**	**data block**	p^a	p^b
Ascon-128	E, $D_{128,64,12,6}$	128	128	128	64	12	6
Ascon-128a	E, $D_{128,128,12,8}$	128	128	128	128	12	8

Table 1: Ascon AEAD parameters

where the first it's a version of AEAD with an increased key size of 160 bits and the latter two are versions of the hash algorithm, but they produce hash outputs of arbitrary length, just changing the number of rounds necessary for it.

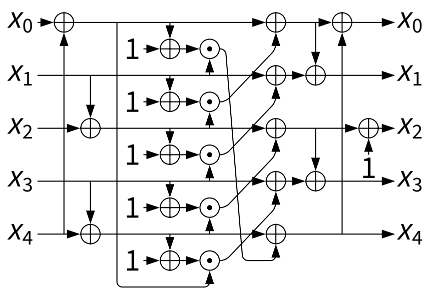
The Table 1 shows the parameters of the recommended AEAD schemes from the Ascon family of algorithms, where this article will focus on the Ascon-128. The algorithms use the encryption function $E_{k,r,a,b}$ and the decryption function $D_{k,r,a,b}$ where k is the key size, r is the rate (data block) size, a and b are the number of rounds used in p^a and p^b permutations used across the algorithms. The encryption $E_{k,r,a,b}(K, A, N, P) = (C, T)$ receives a key K, an associated data A, a nonce N and a plaintext P and returns a ciphertext C and a tag T. The decryption $D_{k,r,a,b}(K, A, N, C, T) \in \{P, \perp\}$ receives a key K, an associated data A, a nonce N, a ciphertext C and a tag T and returns the plaintext P if the verification of the tag is correct, otherwise it returns the $\perp$ error.

Ascon lightweight properties comes from using the simple bitwise operations that majority of microcontrollers have, like XOR, AND, OR, NOT, and bitwise rotations. The algorithm is based in the sponge construct, which is a cryptographic primitive that can be used to build cryptographic hash functions, pseudorandom functions, and authenticated encryption schemes, like the SHA-3 (also known as "Keccak") [7] algorithm. The sponge construct consists in keeping a finite internal state that takes input streams (absorb) to update the state and output streams (squeeze) to produce the output from the internal state. The Ascon state is composed by 5 64-bit words, also named as Ascon words, resulting in a 320-bit internal state. This internal state is then manipulated using the Ascon permutation procedure. In the following subsections, it will be explained the Ascon permutation and the Ascon-128 encryption and decryption procedures, which is the main Ascon scheme that this article will focus on.

2.1 Permutation

The Ascon permutation is the main building block of the Ascon family of algorithms and consists in 3 stages: round constant addition, a substitution-layer (S-Box) and a linear diffusion layer. It's then used in the AEAD encryption and decryption procedures in the form of p^a and p^b, where p is the permutation and a and b are the number of rounds. The parameters a and b are different for each algorithm of the Ascon family, but the permutation is the same for all of them, as it's displayed in Table 1.

As the Ascon state has 5 64-bit words, the round constant addition consists in XORing the round constant with the second Ascon word. The round constant is a 64-bit value that is different for each round. (add round constants table?) The substitution-layer consists in applying an S-Box to the Ascon state. The S-Box is a 5×5 matrix of 64-bit words, where each word is a 5-bit S-Box. The S-Box is applied to each Ascon word, resulting in a new Ascon state as displayed in Figure 1a. The linear diffusion layer consists in the linear diffusion function $x_i \leftarrow \Sigma_i(x_i)$ applied to each Ascon word x_i, where each word has a specific function definition $\Sigma_i(x_i)$. The linear diffusion function definitions are shown in Figure 1b.



(a) Ascon permutation S-box.

$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
x_1 &\leftarrow \Sigma_1(x_1) = x_0 \oplus (x_1 \ggg 61) \oplus (x_0 \ggg 39)\\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg \ \ 1) \oplus (x_2 \ggg \ \ 6)\\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg \ \ 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

(b) Ascon linear diffusion layer

Figure 1: S-box and linear diffusion layers in the Ascon permutation.

2.2 Encryption

The AEAD encryption procedure in Ascon consists of 3 stages: initialization, associated data, plaintext processing and finalization. It's represented in Figure 2. The initialization stage consists in initializing the Ascon state absorbing the bit string consisting of the initialization vector, a key, and a nonce. After that, the Ascon p^a permutation is applied to the Ascon state and absorbs a padded with zeros 320-bitstring of the key.

The associated data stage consists in absorbing the associated data into the Ascon state. The associated data is absorbed in 320-bit blocks, where each block is XORed with the Ascon state and then the Ascon p^b permutation is applied to the Ascon state. The last block of the associated data is padded with zeros until it reaches the 320-bit size of the Ascon state. At the end of the associated data processing stage, the Ascon state absorbs a 320-bitstring of the value 1.

The plaintext processing stage is where the ciphertext is generated. The plaintext is absorbed in the data blocks of size r, specified by the scheme parameters, where each block is XORed with the Ascon state, the state squeezes a ciphertext of same size, and, then, the Ascon p^b permutation is applied. The last block of the plaintext $\tilde{P}_t$ is also XORed with the Ascon state, but the squeezed ciphertext is $\tilde{C}_t \leftarrow \lfloor S_r \rfloor_{|P| \bmod r}$.

The finalization stage consists of XORing the state with the key padded with zeros in the beginning with r zeros and then the remaining zeros to complete a 320-bit bit string, then applying the Ascon p^a permutation to the Ascon state. At the end, the tag T is generated by $T \leftarrow \lceil S \rceil^{128} \oplus \lceil S \rceil^{128}$, and the algorithm returns the ciphertext C and the tag T.

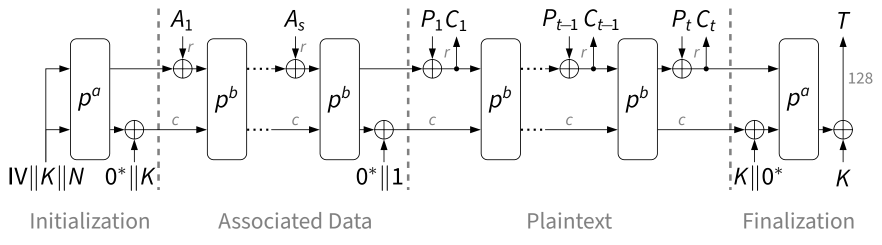


Figure 2: Ascon-128 encryption scheme.

2.3 Decryption

The Ascon-128 decryption has almost the same structure as seen in the encryption. The initialization and the associated data stages are the same as in the encryption. The ciphertext processing stage consists in squeezing a block of plaintext, absorbing a ciphertext block and then permutate the Ascon state with p^b. The last block of the ciphertext $\tilde{C}_t$ is also absorbed in the Ascon state, but the squeezed plaintext is $\tilde{P}_t \leftarrow \lfloor S_r \rfloor_{|C| \bmod r}$. After this, the first Ascon word is XORred with a 64-bit word of the last deciphered plaintext concatenated with the value 1 and padded with zeros until it reaches the 64-bit size of the Ascon word, in mathematical terms, these both operations can be seen in Equation (1) and Equation (2).

The finalization stage consists of XORing the Ascon state with the key padded with r zeros in the beginning and the remaining zeros to complete a 320-bit bit string, then applying the Ascon p^a permutation to the Ascon state. At the end, the tag T is generated by $T \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$, and the algorithm returns the plaintext P if the verification of the tag is correct, otherwise it returns the $\perp$ error.

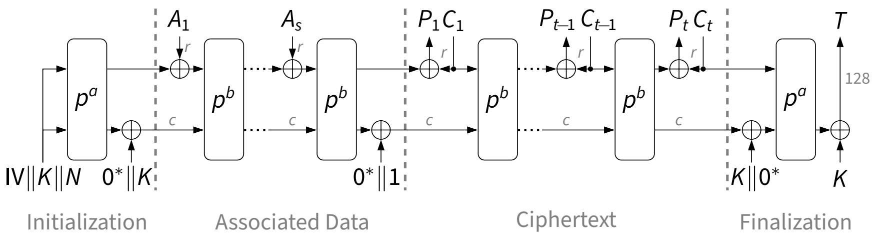


Figure 3: Ascon-128 decryption scheme.

3 Implementation and Optmizations

The device used for this research is the MangoPi MQ-Pro, an SBC powered with an Allwinner D1 chip and 1 GB DDR3 of RAM, with Wi-Fi, Bluetooth and HDMI video output. The Allwinner D1 chip contains a T-Head Xuantie C906 core, a RISC-V 64-bit 1GHz CPU supporting RV64GC ISA. The RV64GC set of extensions is equivalent to the RV64IMAFDCZicsr_Zifencei set of extensions, where I stand for the Integer instruction set, M for multiplication, A for atomic, F for Single-Precision Floating-Point, D for Double-Precision Floating-Point, C for compressed, Zicsr for Control and Status Register (CSR), and, Zifencei for Instruction-Fetch Fence. The board runs Ubuntu Server 23.04, running the 6.2.0-36-generic version of the Linux kernel. For compiling the implementation in C, it was used the RISC-V GNU Compiler Collection (GCC) version 12.2.0 [8] through cross-compilation with Newlib, using a MacBook Pro with an Apple M1 chip. The implementation techniques that are going to be described next were used in the `ascon-v` implementation to attempt to overcome the performance of `ref` and `opt64` implementations.

Since it's a 64-bit architecture, it's possible to store each of the five 64-bit Ascon words in one register at a time. The use of the library `<stdint>` becomes very useful since it's possible to define exactly the bit string size representation, not being dependent on what the C types `long long` and `unsigned long long` translates to. The Ascon permutation S-box translates to what it can be seen in the Listing 1, as well the round constant addition and linear diffusion stages.

Ascon words, used to maintain the state in the sponge construct, are big endian. The reference implementation merges data to these words by loading and storing bytes using big-endian, requiring operations to fill the right-side with zeros. However, RISC-V, as like most other ISAs, is little en-

```
// Ascon state with 5 64-bit words.
typedef struct {
    uint64_t x[5];
} ascon_state_t;

// Bitwise rotation to the right.
static inline uint64_t ROR(uint64_t x, int n) {
    return x >> n | x << (-n & 63);
}

// Ascon permutation round function.
static inline void ROUND(ascon_state_t *s, const uint8_t C) {
    ascon_state_t t;
    /* round constant layer */
    s->x[2] ^= C;
    /* substitution layer */
    s->x[0] ^= s->x[4];
    s->x[4] ^= s->x[3];
    s->x[2] ^= s->x[1];
    t.x[0] = s->x[0] ^ (~s->x[1] & s->x[2]);
    t.x[1] = s->x[1] ^ (~s->x[2] & s->x[3]);
    t.x[2] = s->x[2] ^ (~s->x[3] & s->x[4]);
    t.x[3] = s->x[3] ^ (~s->x[4] & s->x[0]);
    t.x[4] = s->x[4] ^ (~s->x[0] & s->x[1]);
    t.x[1] ^= t.x[0];
    t.x[0] ^= t.x[4];
    t.x[3] ^= t.x[2];
    t.x[2] = ~t.x[2];
    /* linear diffusion layer */
    s->x[0] = t.x[0] ^ ROR(t.x[0], 19) ^ ROR(t.x[0], 28);
    s->x[1] = t.x[1] ^ ROR(t.x[1], 61) ^ ROR(t.x[1], 39);
    s->x[2] = t.x[2] ^ ROR(t.x[2], 1) ^ ROR(t.x[2], 6);
    s->x[3] = t.x[3] ^ ROR(t.x[3], 10) ^ ROR(t.x[3], 17);
    s->x[4] = t.x[4] ^ ROR(t.x[4], 7) ^ ROR(t.x[4], 41);
}
```

Listing 1: Ascon permutation used in `ref` implementation.

dian, making bitwise operations slower than it could be if the architecture had the same endianness as the algorithm. It's possible to implement an optimization considering this issue by handling the data as little-endian in the implementation and reversing the endianness when merging data to the Ascon words [4]. This turns out to be way more effective than operating in data in big endianness since loading bytes and other bitwise operations does not need to fill the right-side of the bit string with zeros, as it is in big-endian. That way, the cost of reversing the endianness of a little-endian 64-bit bit string is lower than the cost of loading data in big-endian.

Another minor improvement is done in the finalization of the ciphertext procedure that happens in the decryption procedure. After retrieving the last section of the decryption of the ciphertext into plaintext, the algorithm defined that the Ascon state needs to XORed with the bitwise string consisting of the last piece of plaintext, the value of 1, and padded bitwise 0 bit string of zeros until it reaches the 320-bit size of the Ascon state. Equation (1) displays the operation, where S_r is the Ascon state, $\tilde{P_t}$ is the last piece of plaintext, $\tilde{C_t}$ is the last piece of ciphertext, and 0^* is the bit string of zeros until it reaches the 320-bit size of the Ascon state:

$$\tilde{P_t} \leftarrow \lfloor S_r \rfloor_{|\tilde{C_t}|} \oplus \tilde{C_t} \tag{1}$$

$$S_r \leftarrow S_r \oplus (\tilde{P_t}||1||0^*) \tag{2}$$

The `ref` and `opt64` implementations do this operation by cleaning the first (in big-endian representation) $|\tilde{C_t}|$ bytes of S_r, then ORing with $\tilde{C_t}$ to load these bytes into S_r. Since $S_r \oplus \tilde{P_t} = S_r \oplus (S_r \oplus \tilde{C_t}) = 0 \oplus \tilde{C_t} = \tilde{C_t}$, $S_r \oplus \tilde{P_t}$ is equivalent to $S_r \mid = \tilde{C_t}$, after the first $|\tilde{C_t}|$ bytes of S_r are cleared. In `asconv`, in the purpose of trying to improve the performance, the operations done in Equation (1) and Equation (2) are done with bitwise shift instructions, as it's displayed in Equation (3) and Equation (4):

$$d \leftarrow (S_r \oplus \tilde{C_t}) \gg (r - |\tilde{C_t}|) \ll (r - |\tilde{C_t}|) \mid (1|0^*) \tag{3}$$

$$S_r \leftarrow S_r \oplus d \tag{4}$$

It was used optimization techniques to improve the performance of the implementation by using compile-time optimizations. Because many of the operations used in the Ascon are used very often, it's faster to implement them as `inline` functions, so the compiler can optimize the code by inlining the function calls. The `ROUND` function, used in the Ascon permutation, is a good example of this, as it's used in every round of the permutation. The permutations p^6 and p^{12}, used in Ascon128, instead of retrieving the value of their round constants in runtime, it's better to implement as inline functions to eliminate the time of getting the specific round constant value for that round. Compilation flags were also used to improve the performance, using `-O2` optimization, `-march=rv64gc` to enable the RV64GC ISA, and `-mtune=thead-c906` to enable the compiler to optimize the code for the C906 CPU core inside the Allwinner D1 chip.

4 Results

All the implementations (`ref`, `opt64` and `asconv`) were then benchmarked to compare the performance of each code. To benchmark, it was measured the elapsed time, the clock cycle, of operations encryption and decryption for each implementation, with different message sizes. implementations Considering t the elapsed time to run encryption/decryption of a plaintext/ciphertext, the resolution R of the timer used to measure the time of the C906 core to be 45 nanoseconds [9], F the CPU frequency, the number of clock cycles used in encryption/decryption can be calculated Equation (5):

$$C = t \times R \times F \times \frac{10^9}{60} \tag{5}$$

The CPU time was measure using the RISC-V instruction `rdcycle`, which is read from a CSR register. Then, this value is used as t in the Equation (5) to calculate the cycle count. In Table 2 and Table 3, it's possible to see the benchmark of the `asconv` implementation of Ascon-128 of the

Implementation	**Message size (B)**	**Cycles**	**Cycles/B**	**Time (s)**
`asconv`	8	66	9	0.000003960
`asconv`	32	96	3	0.000005805
`asconv`	64	123	1	0.000007380

Table 2: Benchmark of `asconv` implementation of Ascon-128 of the encryption operation with different message sizes.

Implementation	**Message size (B)**	**Cycles**	**Cycles/B**	**Time (s)**
`asconv`	8	60	8	0.000003960
`asconv`	32	83	2	0.000005805
`asconv`	64	120	1	0.000007380

Table 3: Benchmark of `asconv` implementation of Ascon-128 of the decryption operation with different message sizes.

encryption and the decryption operations with progressively increasing message sizes. The cycle count per byte decays along with the message size, as it's expected.

The Table 4 shows the benchmark of the `asconv` implementation of Ascon-128 when compared with the reference implementation (`ref`) and the optimized implementation for a generic 64-bit architecture (`opt64`). The benchmark was done considering a message of 4 MB. The `asconv` implementation is 11% faster than the `ref` implementation when encrypting and 12% faster when decrypting. Still, the `opt64` is the fastest implementation, being 16% faster than the `asconv` implementation when encrypting and 18% faster when decrypting, placing `asconv` in the middle of the `ref` and `opt64` implementations.

Implementation	**Encrypt cycles**	**Decrypt cycles**	**Encrypt speed**	**Decrypt speed**
`ref`	4533	4603	1.00	1.00
`opt64`	3912	3917	1.16	1.18
`asconv`	4078	4093	1.11	1.12

Table 4: Comparison and relative speeds of different implementations of Ascon-128 when compared with the reference implementation (`ref`). The benchmark was done considering a message of 4 MB.

5 Conclusions

As seen in Section 4, the RV64GC instructions do not allow great optimizations from the architecture itself, since it doesn't have any special instructions to accelerate operations of the Ascon-128.

The Ascon permutation proposed in the specification, shown in Listing 1, also allows the use of parallelism that could accelerate the performance. Unfortunately, the Allwinner D1 chip does not support vectorial instructions and the XuanTie C906 microarchitecture is single-issue only, so the analysis of the use of parallelism for increasing the algorithm performance will be left for future work.

However, RISC-V does have instructions extensions that were recently ratified, that could improve the performance of Ascon, allowing great optimizations using hardware accelerated instructions. Such cryptographic specialized instruction extensions are divided in scalar [10] and vectorial

[11]. The Scalar Cryptography set of extensions (Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh, Zkn, Zks, Zkt, Zk, Zkr) provide instructions that could accelerate operations of the Ascon permutation. The Zbkb extension provides bit manipulation instructions for cryptographic operations. Such operations are bit rotations (`rori`), and, bitwise logical AND operation between a value a and the bitwise inversion of a value b (`andn`), that could accelerate the Ascon permutation as seen in Listing 1. This same extension also provides a byte-reverse register instruction (`rev8`) that could be used to reverse the endianness of the Ascon words, making the work with little-endian data loading first and then reversing to big endianness way faster. The Zkn extension introduces an entropy source within a dedicated CSR register. Utilizing this feature, the algorithm gains the ability to generate random numbers for both nonce and key, thereby significantly enhancing the security protocols within the algorithm. There's also the Vectorial Cryptography set of extensions, which consists in the vectorial versions of the Scalar Cryptography instructions, allowing even more the use of parallelism.

References

[1] H. Fujii and D. F. Aranha. "Curve25519 for the Cortex-M4 and beyond". In: June 2017. URL: `http://www.cs.haifa.ac.il/~orrd/LC17/paper39.pdf`.

[2] Meltem Sönmez Turan et al. "Status Report on the Final Round of the NIST Lightweight Cryptography Standardization Process". In: (2023).

[3] Christoph Dobraunig et al. *Ascon v1.2*. Submission to Round 1 of the NIST Lightweight Cryptography project. 2019. URL: `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf`.

[4] Lars Jellema. "Optimizing ascon on RISC-V". In: *Bachelor Thesis, Radboud University* (2019).

[5] Ascon Team. *ascon-c: Reference and optimized implementations of Ascon*. `https://github.com/ascon/ascon-c`. 2023.

[6] Paulo Pacitti. *ascon-v: Ascon lightweight cryptographic algorithm implementation for improved performance on RISC-V*. `https://github.com/paulopacitti/ascon-v`. 2023.

[7] Guido Bertoni et al. "Keccak". In: *Cryptology ePrint Archive* (2015).

[8] RISC-V Collaboration. *riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC*. `https://github.com/riscv-collab/riscv-gnu-toolchain`. 2023.

[9] Lukas Gerlach et al. "A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2321–2338. DOI: `10.1109/SP46215.2023.10179399`.

[10] RISC-V Foundation. *RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions (v1.0.1)*. `https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar`. 2023.

[11] RISC-V Foundation. *RISC-V Cryptography Extensions Volume II: Vector Instructions (v1.0.0)*. `https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0`. 2023.