Optimizing Ascon on RISC-V

Lars Jellema

Supervisors
Peter Schwabe
Christoph Dobraunig

July 10, 2019

Abstract

Ascon is a family of authenticated encryption schemes which appears in the final portfolio of the CAESAR competition and as a candidate in the NIST lightweight cryptography competition. It uses a cryptographic permutation as main building block that is designed to have a straightforward and efficient implementation.

RISC-V is a hardware instruction set architecture which is completely open-source and based RISC principles. It offers a small and simple instruction set with little space for optimization.

We optimize the Ascon's permutation for the RISC-V RV32IMAC architecture. We find improvements over the reference implementation in several parts and get very close to the theoretical optimal speed. At the same time, we keep register usage to a minimum to allow our techniques to be ported to other architectures easily.

1 Introduction

In recent years, many critical vulnerabilities have been found in various kinds of software and hardware. The software industry continues to grow and the value of hacking grows with it. Making software secure is hard and many companies consider it a waste of money. This has lead to botnets, among other things. Botnets usually consist of a large number of cheap internet-connected consumer devices that are easily hacked. Because the profit margins are smaller for cheap devices, there is often less money available for implementing good security.

One way to make good security cheaper is by providing security primitives with simple interfaces that nonetheless ensure all desirable properties. An example of this is *authenticated encryption* systems, which provide all of the most important security properties for symmetric encryption. By using authenticated encryption, a developer does not need to introduce authentication to an encryption system manually, leading to fewer possibilities to make mistakes.

Authenticated encryption has traditionally been implemented by combining privacy-only encryption with message authentication codes [1] [2]. This combination turned out to be hard to make without introducing new vulnerabilities. In addition, performance suffers because two passes need to be made over the plaintext, one for encryption and one for authentication. To resolve this, cryptographers have called for new primitives that integrate all desirable properties.

This call has taken the shape of several competitions, where a diverse set of teams each publish a family of new primitives, followed by public analysis of these new primitives by many other researchers. New primitives are then selected based on how well they withstood public analysis.

Ascon is one of these new primitives, aimed at lightweight applications. It was first submitted to the CAESAR competition [3] and has been selected as the first choice for lightweight applications in the final portfolio. It was also recently submitted to be considered for standardization in NIST Lightweight Cryptography [4] and has been selected as a round 1 candidate.

While secure software is important, the hardware it runs on must also be secure. The integrated circuit industry has tradionally been a closed ecosystem. Chip manufacturers keep their implementations secret in order to monopolize the market. This means that when a vulnerability is discovered in a chip, consumers are entirely dependent on the manufacturer to fix them. It also makes it hard for third parties to verify a chips behavior.

RISC-V is a hardware instruction set architecture that intends to solve this issue by being completely open-source. This means anyone is free to use, implement, extend and adapt it. As a result, RISC-V has been gaining popularity with researchers as well as companies. RISC-V is an architecture based on RISC principles, which makes it a good fit for low-power applications.

So far, there has been little work towards optimizing crypto algorithms for RISC-V in software. Most efforts have been focused on creating hardware extensions for RISC-V as this was not possible with closed-ecosystem architectures. Despite the fact that RISC-V extensions do not have licensing costs and have reduced development costs, chip manufacturing remains expensive, so fast software implementations remain important. We optimize the Ascon authenticated encryption system for the RISC-V instruction set architecture.

1.1 Related work

Although there is very little related work for software optimizations of crypto for RISC-V, there are a number of previous implementations of Ascon. Among others, it has been optimized on FPGAs [5] and the ARM Cortex-M3 [6] and optimization using ARM NEON was attempted [7]. Some optimized software implementations have been benchmarked on various platforms as well [8].

2 Preliminaries

2.1 Symmetric encryption

In communication, it is often desirable to keep one's messages hidden from third parties. This property is called *confidentiality.* In more precise terms, confidentiality means that a message is transformed in such a way that authorized parties can recover the original message from it, while unauthorized parties can not.

In symmetric encryption, the authorized parties are defined as the parties that know some secret information, called the *key.* The original message is called the *plaintext* and the transformed message is called the *ciphertext.* The transformation from plaintext to ciphertext is called *encryption*, while the transformation back is called *decryption.* When the same secret key is needed for both encryption and decryption, it is called *symmetric encryption.* Because the key is needed for decryption, unauthorized parties are prevented from recovering the plaintext.

2.1.1 Nonces

While unauthorized parties are unable to decrypt ciphertexts, similar plaintext may result in similar ciphertexts, and unauthorized parties may therefore detect when similar messages are sent. In order to prevent this, during every encryption, a different number is used to modify the resulting ciphertext. It is needed again during decryption in order to revert that modification. Because a different number is used every time, the resulting ciphertexts will also differ.

This number is called the *nonce*, short for number used once. If it is used more than once, unauthorized parties may be able to infer information about the difference or similarity of the messages it was used for. One option to ensure uniqueness in practice is to use random numbers that are sufficiently large to make the chance of duplicates negligable. When it is chosen randomly, it must be attached to the ciphertext unencrypted, as decryption will fail without it.

A counter can also be used to generate unique nonces, if it is sufficiently large that it will never overflow within its lifetime. This has the advantage that a good source of randomness is not needed, and the disadvantage that the current count must be kept track of and synchronized between uses. If all messages are sent synchronously and messages cannot be lost, this can be as simple as attaching the current counter to each message.

If there is no space to attach the nonce to the ciphertext, it is not possible to use a random nonce. In this case, information that is available during both the encryption and decryption should be used to construct a unique nonce. For example, when adding encryption support to existing filesystems, there may be no space for nonces in the existing data structures and these probably need to stay compatible with earlier versions. In this case, a nonce for a block can be constructed by combining the inode and offset within the file into a nonce. With this construction, uniqueness cannot be guaranteed, so security is reduced, but if encryption is otherwise not possible, it is still an improvement over unencrypted data or encryption without nonces.

2.1.2 Authentication

Although unauthorized parties cannot recover the plaintext from the ciphertext, they may be able to modify the ciphertext, resulting in a modified plaintext after decryption. It is not always possible to prevent such modifications, but it is possible to detect them. The property that it is detectable whether a message has been changed, is called *integrity*.

Unauthorized parties may also attempt to construct messages from scratch. Because the key is needed for encryption, they will be unable to encrypt a specific plaintext, however, they will be able to send specific ciphertexts, even if they do not know what plaintexts corresponds to them. In order to prevent this, it is desirable for the receiving party to be able to verify that a message comes from an authorized party. Together with integrity, this property is called *authenticity*. Encryption that provides both confidentiality and authenticity to the plaintext is called *authenticated encryption* or *AE*.

Authenticated encryption is usually implemented by generating an *authentication tag*. Just like the ciphertext, this tag is based on the key, the nonce and the plaintext. This tag is needed again during decryption. The decryption algorithm checks if the tag is correct and only returns a plaintext if it is. If the ciphertext is modified, it will decrypt to a different plaintext from the one used to generate the tag and cause the decryption to fail. Because the tag also depends on the key, valid tags cannot be created without it.

2.1.3 Associated data

Even when confidentiality and authenticity are assured, unauthorized parties may still repeat a message they have seen before in a different context. To prevent this, some data about the context in which a message is allowed to appear can be associated with it. This is one of the uses of *associated data*, which is defined as information that requires authentication but not confidentiality. An authenticated encryption scheme that supports this is called *authenticated encryption with associated data* or *AEAD*.

Figure 1: The following table specifies the symbols and notation used in this document.

$\texttt{1337}_h$	hexadecimal number
$\perp$	verification failure
$x \parallel y$	concatenation of bitstrings x and y
$x \oplus y$	bitwise addition of bitstrings x and y
$x \ggg n$	64-bit word x rotated right by n bits
$\mathbf{2}^l$	set of messages of l bits
$\mathcal{P}(S)$	Set of all subsets of set S.

In some encryption schemes, associated data is used during encryption to modify the resulting ciphertext and is needed during decryption in order to revert that modification. In these cases, because the correct associated data is needed during decryption, it prevents messages from being decrypted using the wrong associated data. In other schemes, associated data only affects the authentication tag and it is the responsibility of the implementation to reject inputs with incorrect authentication tags.

The associated data can be sent or stored together with the ciphertext, but in some cases it can also be inferred from other data during encryption and decryption. In those cases, sending or storing it is not necessary. If the associated data is inferred from the context, it can prevent encrypted data from being decrypted in the wrong context. For example, this prevents two encrypted records containing positive and negative information respectively from being swapped between contexts specifying different persons.

Note that some encryption schemes do not need the associated data for decryption. Correct implementations of these will still check the associated data and return failure if it does not match. The behavior is the same, but such schemes allow incorrect implementations to go undetected more easily.

2.1.4 Formal definition

To formalize AEAD schemes, we begin by defining their interface: An AEAD scheme is defined by a tuple of functions $(\mathcal{E}, \mathcal{D})$. $\mathcal{E}$ is a function that takes a key, a nonce, some associated data and a message and produces a ciphertext and an authentication tag.

$$\mathcal{E} : K \times N \times A \times M \to C \times T \tag{1}$$

$\mathcal{D}$ is a function that takes a key, a nonce, some associated data, a ciphertext and an authentication tag and produces either a failure or a message.

$$\mathcal{D} : K \times N \times A \times C \times T \to M \cup \{\perp\} \tag{2}$$

Let k be a key, n be a nonce, a be any associated data and m be a message. The result of decryption after encryption with the same nonce, key and associated data is the original message.

$$\forall k, n, a, m. \quad \mathcal{D}(k, n, a, \mathcal{E}(k, n, a, m)) = m \tag{3}$$

In particular, this means decryption never returns $\perp$ when supplied with valid encryption results. This property ensures the message can be recovered from the ciphertext and related data. If this property were not required, an AEAD scheme could be constructed simply by always returning the empty ciphertext for encryption and failure for decryption.

The remaining properties of AEAD systems, authenticity and confidentiality, must be defined with respect to a *security parameter*. The security parameter specifies how much effort is required to break the security properties of the system. The security parameter is expressed in bits and is equivelant to the 2-log of the number of operations needed to break the security claim.

A security property holds when no algorithm exists that can break it in fewer operation than specified by its security parameter. We say that no *efficient* algorithm exists. These algorithms implicitly have access to all public knowledge.

We can now define the property of authenticity. First, we define V_k to be the set of valid decryption inputs for key k.

$$\forall k \in K. \quad V_k = \{(n, a, c, t) \mid \mathcal{D}(k, n, a, c, t) \neq \perp\} \tag{4}$$

Authenticity is violated when an efficient algorithm exists that constructs valid and new decryption inputs without the correct key. An encryption system provides authenticity if no such algorithm exists.

$$\nexists \mathcal{A}. \quad \forall k \in K, s \in \mathcal{P}(V_k). \quad \mathcal{A}(s) \notin s \wedge \mathcal{A}(s) \in V_k \tag{5}$$

To define confidentiality, we must define what it means to know something about a message. In order to keep this definition simple, we will assume the length of a message is always known, as is often the case with AEAD schemes.

We will define knowledge about a message m of length l as a probability distribution over all messages of that length, $\mathbf{2}^l$. We define u_l to be the uniform probability distribution, which has no information about any message.

$$\forall l \in \mathbb{N}, m \in \mathbf{2}^l. \quad u_l(m) = \frac{1}{2^l} \tag{6}$$

When the probability that a distribution d picks m is greater then the probability u picks m by a factor of 2^b, we say d knows b bits about m. $I(d, m)$ represents the information in bits that d has about m.

$$\forall l \in \mathbb{N}, m \in \mathbf{2}^l, d \in D(\mathbf{2}^l). \quad I(d, m) = \log_2 \frac{d(m)}{u_l(m)} \tag{7}$$

An encryption scheme provides confidentiality if no efficient algorithm exists that determines knowledge about a message given its decryption inputs and all prior public knowledge. This public knowledge can include many other valid decryption inputs, as well as decryption inputs with their corresponding plaintexts, but with different nonces.

$$\nexists \mathcal{A}. \quad \forall k, n, a, m. \quad I(\mathcal{A}(k, n, a, \mathcal{E}(k, n, a, m)), m) > 0 \tag{8}$$

It should be noted that the formal definitions given above are not completely accurate. For example, it is hard to formalize the tradeoffs between computational power and odds of succes for an attack. It is also hard to formalize all the public data that attacking algorithms can be based on. There are entire books dedicated to formalizing these properties so the formalizations above should be taken with a grain of salt.

2.2 Ascon internals

Ascon [9] is an authenticated encryption cipher designed for use in resource-constrained environments, like embedded devices. It has an internal state of just 320 bits, which can be kept in registers on most architectures. This ensures moving data between registers and memory is kept to a minimum, which is important, as embedded devices usually do not have the same amount of cache available as larger systems.

2.2.1 Mode of operation

Ascon aims to provide 128 bits of security. To that end, its key, nonce, and authentication tag are 128 bits in size each. The plaintext, ciphertext and associated data can all be of any length and are processed in *blocks*. There are multiple variants of Ascon, we implement two of them: Ascon-128, which processes 64-bit blocks and Ascon-128a, which processes 128-bit blocks.

Ascon uses a 320-bit state and a permutation that mixes the state in a way that is hard to reverse. This permutation consists of a transformation that is applied in multiple rounds, each with a different round constant. Both variants of Ascon use permutation p^a, which consists of 12 rounds, during initialization and finalization. After processing each block, Ascon uses permutation p^b, which consists of 6 rounds in Ascon-128 and 8 rounds in Ascon-128a. Figure 2 gives an overview of encryption and decryption.

Encryption starts with the initialization of the state. The state is initialized to the concatenation of an initialization vector, the key and the nonce. The initialization vector encodes the parameters of the variant of Ascon that it is used for. This initial state is then passed through the first permutation, p^b. Finally, the key is bitwise added to the least significant bits of the state again.

After initialization, the associated data is mixed into the state. If there is no associated data, the state is left unmodified. Otherwise, the associated data is padded and split into blocks. Each of the blocks is bitwise added to the most significant bits of the state, followed by applying the second permutation, p^b. At the end, the least significant bit is inverted. This is done to separate the associated data from the plaintext, which comes next.

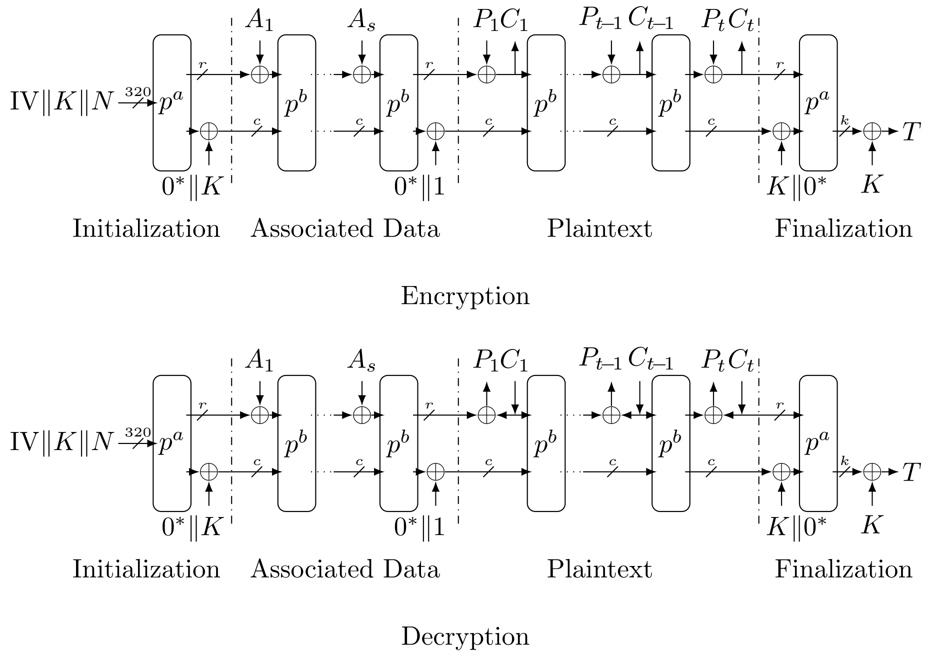

Figure 2: Ascon's mode of operation

The plaintext is padded and split into blocks in the same way as the associated data. Each block of plaintext is bitwise added to the most significant bits of the state. The result of this bitwise addition also forms the corresponding block of the ciphertext. After adding each block, except for the last, permutation p^b is applied again.

Finalization begins with bitwise adding the key to the most significant bits of the state that were not used to add the plaintext. After that, the permutation p^a is applied again and finally, the key is bitwise added to the least significant bits of the state again. The result of this last operation is used as the authentication tag.

Decryption is identical to encryption, except the ciphertext is processed in a slightly different way from the plaintext: Each block of the ciphertext is again bitwise added to the most significant bits of the state to form the plaintext, but instead of using this result in the state, the most significant bits of the state are replaced with the ciphertext before the bitwise addition.

2.2.2 Permutation

Ascon's main component is a permutation consisting of three phases, which are applied in several rounds. Ascon-128 uses 6 rounds to process blocks of 64 bits at a time. Ascon-128a uses 8 rounds to process blocks of 128 bits at a time. Both use keys of 128 bits and 12 rounds to initialize and finalize the state. A round consists of the following three phases: The addition of the round constant, the substitution layer and the linear diffusion layer. Each of these phases modify the internal state in a different way. The internal state consists of 320 bits, logically split into five 64-bit state words called x_0, x_1, x_2, x_3 and x_4.

The round constant is a single byte that changes from round to round and is added bitwise to the least significant 8 bits of x_2. It ensures rounds are not all identical. The final round constant is always $\texttt{4b}_h$. The round constant changes linearly, decreasing by $\texttt{f}_h$ every round. This means the round constant can be computed easily based on the number of rounds that are left.

The substitution layer applies a 5-bit lookup function (called an S-box) in parallel to the five state words. It provides non-linear mixing between the five state words. It is usually implemented as a sequence of bitwise operations which compute the substitution on five machine words in parallel.

The linear diffusion layer provides linear mixing of the bits within each state word. It consists of a bitwise addition of the original state word with the same state word rotated by two different amounts. Each of the five state words uses different rotation amounts.

2.3 RISC-V

RISC-V [10] is a hardware instruction set architecture which aims to be completely open: Anyone is free to create custom variations and implementations. It is also designed to be modular: A basic RISC-V processor starts with just the base integer instruction set, using either 32-bit or 64-bit machine words. Several extensions are available which offer different common processor features, such as support for floating point operations. Manufacturers can choose to implement any combination of these extensions and will usually pick extensions based on the intended application.

In this thesis, we will use the HiFive1 development board created by SiFive. It features a `RV32IMAC` processor. The `RV32I` stands for the RISC-V 32-bit base integer instruction set. The letters that follow stand for the extensions that are implemented in this processor: `M` stands for the multiplication extensions which offers integer multiplication and division instructions, `A` stands for the atomic memory access extension and `C` stands for the compressed instruction set extension which can be used to reduce code size.

At this point it is important to note that our target processor uses 32-bit *machine words*, while Ascon is defined in terms of five 64-bit *state words*. These state words do not fit in a single register, so each state word will be stored using two registers. We describe our implementation mostly in terms of machine words.

For Ascon, we won't need multiplication instructions or atomic memory access. The compressed instructions are used transparently by the assembler: Each compressed instruction corresponds to a single full size instruction and whenever a compressed instruction is available, it will be used instead of a full size instruction. We would like to use the bit manipulation extension, but it is not available on this board and may not be available on other embedded RISC-V systems, so we optimize Ascon for RISC-V processors without it.

RISC stands for Reduced Instruction Set Computer, as opposed to CISC, which stands for Complex Instruction Set Computer. RISC architectures aim to have a small set of simple and general instructions, while CISC architectures aim to provide complex instructions which can do multiple things at once. An example of this is that the CISC x86 architecture allows memory access as part of almost every instruction, while RISC architectures usually have separate memory access instructions.

RISC-V takes RISC to an extreme, it is much more reduced than ARM, which stands for Advanced RISC Machine. For example, RISC-V does not save carries for arithmetic operations and it only has about 40 instruction in the base integer instruction set, all of which do only one thing. Figure 3 shows the eight instructions needed for the inner loop of Ascon.

3 Optimization

In this section, we will optimize Ascon in several parts. We begin by looking at the state representation and we continue by considering the possibilities for each phase of the Ascon permutation. We find efficient implementations for them and consider whether further improvement is possible. In the end, we look at the entire round and the entire permutation, which loops over several rounds.

Figure 3: The RISC-V instructions used in the inner loop of Ascon.

`and r, a, b`	Store the bitwise and of registers `a` and `b` in register `r`.
`or r, a, b`	Store the bitwise or of registers `a` and `b` in register `r`.
`xor r, a, b`	Store the bitwise addition of registers `a` and `b` in register `r`. This operation is also known as bitwise exclusive or.
`not r, a`	Store the bitwise inversion of register `a` in register `r`.
`addi r, a, b`	Store the sum of the value of register `a` and constant `b` in register `r`.
`srli r, a, b`	Store the value of register `a` shifted right by constant amount `b` in register `r`.
`ssli r, a, b`	Store the value of register `a` shifted left by constant amount `b` in register `r`.
`bne a, b, l`	Jump to the label `l` if the value of register `a` is not the same as the value of register `b`.

3.1 Endianness

The first optimization we can make is due to endianness. Ascon uses big endian state words, while most processors, including the HiFive1, support little endian memory access only. This means the endianness of the plaintext and the associated data needs to be reversed before it can be merged into the state.

The reference implementation switches endianness while loading and storing the Ascon state to memory every time new data needs to be merged into it. It does this by storing individual bytes, which means 40 load and store operations are needed, while just 10 would be sufficient if the endianness had matched.

To improve on this number, it's possible to keep the endianness of the state the same at all times and instead switch the endianness of the data that needs to be merged into it. For Ascon128, this means only 64 bits of data needs their endianness reversed instead of 320 bits, and for Ascon128a, only 128 bits of data needs their endianness reversed. In addition, the data that needs to be merged into the state only needs to be loaded, it doesn't need to be stored in the same endianness again.

3.2 Substitution layer

The substitution layer in Ascon is a 5-bit S-box which is applied in parallel to 64 sets of 5 bits. This allows an efficient implementation using *bit slicing*. Bit slicing is a technique for computing many lookup tables in parallel and in constant time by computing the lookup table using binary operations. Each of these binary operations take two bits as input and produce one bit as output. All common architectures have bitwise operation instructions that run such binary operations in parallel on entire machine words.

Ascon was optimized for bit slicing: The 5 bits of the S-box are located in 5 different state words in the same bit position. The Ascon paper describes an instruction sequence that uses bit slicing to compute the S-box efficiently. The

```
x0 ^=  x4;  x4 ^=  x3;  x2 ^=  x1;
t0  = ~x0;  t1  = ~x1;  t2  = ~x2;  t3  = ~x3;  t4  = ~x4;
t0 &=  x1;  t1 &=  x2;  t2 &=  x3;  t3 &=  x4;  t4 &=  x0;
x0 ^=  t1;  x1 ^=  t2;  x2 ^=  t3;  x3 ^=  t4;  x4 ^=  t0;
x1 ^=  x0;  x0 ^=  x4;  x3 ^=  x2;  x2  = ~x2;
```

Figure 4: This instruction sequence is the parallel implementation the S-box of Ascon was designed for. It consists of 22 operations.

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

Figure 5: These formulas compute the Ascon S-box in 17 operations (once duplicate operations are taken out), while the reference implementation uses 22 operations. o_n indicates output bit n and i_n indicates input bit n.

instruction sequence translates to 22 single-cycle instructions. Because these can only be applied to 32 bits at a time on 32-bit platforms and the five state words are 64 bits in size each, they need to be run twice. Therefore, a straightforward implementation takes 44 cycles. The instruction sequence is shown in Figure 4.

The substitution layer can be optimized by computing the same S-box with a different formula. I derived shorter binary formulas by first writing down the the bit sequences that occur for each of the 5 output bits for all 32 inputs. This allowed me to recognize patterns in the bit sequences. I first eliminated input bits that did not affect the output, or only affected the output through a single exclusive or operation. Then, I looked at the remaining bit patterns and found short and overlapping binary formulas for them.

Figure 5 shows the result of this analysis. These formulas are based on three bitwise additions which are used three times each. Three operations are used to compute these bitwise additions and 14 more operations are then used to compute all formulas. This results in a total of 17 operations, five fewer than the reference implementation.

From these, an instruction sequence can be produced like the one in Figure 6. It computes all five of the above formulas with one caveat: The results end up in different registers. We will compensate for this in the linear diffusion layer.

```
.macro sbox s0, s1, s2, s3, s4, r0
        xor t0, \s1, \s2
        xor t1, \s0, \s4
        xor t2, \s3, \s4
        not \s4, \s4
        or \s4, \s4, \s3
        xor \s4, \s4, t0
        xor \s3, \s3, \s1
        or \s3, \s3, t0
        xor \s3, \s3, t1
        xor \s2, \s2, t1
        or \s2, \s2, \s1
        xor \s2, \s2, t2
        not t1, t1
        and \s1, \s1, t1
        xor \s1, \s1, t2
        or \s0, \s0, t2
        xor \r0, \s0, t0
.endm
```

Figure 6: This macro computes boolean formulas of the substitution layer in parallel on the registers `s0` through `s4`, using `t0`, `t1` and `t2` as temporary registers. This macro must be run twice with different state registers, because each 64-bit state word is split in two 32-bit machine words. Note that after this macro, the state words end up in different registers. In order to make space to move the state words back into position, the final result is stored in register `r0` instead.

3.2.1 Superoptimization

As these formulas were constructed by hand, it may be possible to do better. There are many possible formulas, so it is infeasable to find the best option by hand. One option is to use something like the GNU Superoptimizer [11] which tries all possible instruction sequence of a certain length in order to see if any of them computes a specific formula. Unfortunately, according to its README, the longest instruction sequence it was able to find for anything was seven instructions long. This is not enough, since it is expected that at least five bitwise addition operations and at least five bitwise and operations need to be computed, leading to a minimum of 10 instructions.

Stoffelen [12] attempted to optimize binary formulas for S-boxes of various cryptographic primitives using a SAT solver. He found ways to compile an S-box to a satisfiability problem determining whether it can be computed in a given number of instructions. I made use of his project to prove that it is not possible to compute the Ascon S-box in 10 instructions. Unfortunately, the project did not finish within reasonable time for instruction counts larger than 10, so I was unable to verify whether 17 instructions is the best number possible.

3.3 Linear diffusion layer

The linear diffusion layer in Ascon mixes the bits with each of the five state words. Each of them is rotated by two different amounts and added with the results. For each of the five state words, two different rotation amounts are used.

The state words are 64 bits in size, so 64-bit rotations are used. On most 64-bit architectures, this can be done with just one instruction, but 32-bit architectures generally do not have 64-bit rotate instructions.

The most straightforward method of simulating a 64-bit rotate instruction is using shifts. By combining a left shift and a right shift, all bits of a single machine word can be placed at the correct offset. Because each shift loses some of the bits when they are shifted out, it is not possible to simulate a rotate instruction with one shift. Because a state word is two machine words in size, four shifts are needed in total. After this, there are four intermediate results that need to be combined to two resulting machine words. There are several options to combine these intermediate results:

Since each bit will be zero in at least one of the two operands, the combining instructions can be bitwise or, bitwise exclusive or, or integer addition instructions. Since the results will be combined using bitwise exclusive or operations, we will use these for combining intermediate shift results as well. Because bitwise exclusive or operations are associative, it does not matter in what order all intermediate results are combined, which gives us more freedom during implementation. Bitwise exclusive or can also be seen as bitwise addition, or addition without any carries between the bits. I will use *bitwise addition* to refer to this operation from here on.

Because two shifts are needed to simulate one rotate instruction, it would be preferable to use the latter. Unfortunately, rotating both halves of a state word independently does not have the same effect as rotating the word as a whole.

This is usually resolved using bit interleaving: By storing even-numbered bits in one word and odd-numbered bits in another, a 64-bit rotation can be simulated using two 32-bit rotations. For even rotation amounts, both words are rotated by half the amount. For odd rotation amounts, the words are rotated by half the amount, rounded up and down respectively, and then swapped. The disadvantage is that some extra cycles are needed to convert between bit-interleaved representation and normal representation.

Unfortunately, the microcontroller we are targeting does not have rotate instructions at all. The SiFive HiFive1 does not include the bit-manipulation extension, which is where RISC-V introduces its rotate instructions. This means that we are stuck using the straightforward method using four shifts and two bitwise additions. The result also needs to be bitwise added back into the original, which takes another two instructions, one for each of the machine words. This means that handling one rotation takes eight instructions in total.

There are five state words which each have two rotations applied to them. This means that a total of 80 instructions are needed for the linear diffusion layer. On 32-bit RISC-V, without the bit manipulation extension, it is not possible to do better. This is easy to see:

The only method of moving bits to different offsets is the shift instruction, and because some bits are lost when they are shifted out, there are always two needed to handle 32 bits, or one machine word. Therefore, to move all bits into two separate new positions for five state words, or ten machine word, a total of 40 shifts are needed. Each of these shifts produces an intermediate result and all those intermediate results need to be merged back into the unrotated state words.

Each merging instruction takes two operands and produces one result, reducing the number of intermediate results by one, so a total of 40 merging instructions is needed to merge all intermediate results. On the HiFive1, there are no instructions that can merge more than two machine words at a time. This means that it is not possible to do the linear diffusion layer in less than 80 instructions. Figure 7 shows how the rotations for a single state word are implemented in RISC-V assembly.

3.4 Addition of round constant

The addition of the round constant is a very short phase: A single byte gets bitwise added to one of the state words. This takes one instruction. The correct value to be added still needs to be computed though. We note that the round constant for any given round is always $\mathtt{f}_h$ lower than in the previous round. This means we can compute it based on the previous round with one subtraction.

```
.macro xorror dl, dh, sl, sh, sl0, sh0, r0, sl1, sh1, r1
        slli t0, \sl0, (32 - \r0)
        srli t2, \sh0, \r0
        xor t0, t0, t2
        slli t2, \sl1, (32 - \r1)
        xor t0, t0, t2
        srli t2, \sh1, \r1
        xor t0, t0, t2
        slli t1, \sh0, (32 - \r0)
        srli t2, \sl0, \r0
        xor t1, t1, t2
        slli t2, \sh1, (32 - \r1)
        xor t1, t1, t2
        srli t2, \sl1, \r1
        xor t1, t1, t2
        xor \dl, \sl, t1
        xor \dh, \sh, t0
.endm
```

Figure 7: The `xorror` macro applies the rotates and bitwise additions of the linear diffusion layer to a single state word. Because of the limitations of assembly macros, it takes many arguments: `dl` and `dh` are the registers for storing the low and high part of the result. `sl`, `sh`, `sl0`, `sh0`, `sl1` and `sh1` are the source registers for the state word, without rotation, with rotation `r0` and with rotation `r1` respectively. Because 32-bit shift instructions can only shift by 31 at most, representing rotate amounts above 32 is done by subtracting 32 and swapping the respective source registers. This is why the source registers must be supplied three times.

3.5 Full round

The round constant for the initial round is based only on the number of rounds that will be run in that invocation of the permutation. Therefore, it can be a parameter to the permutation which is constant in every location where it is called. Because the constant is just one byte in size, the initial constant can be created with just one move immediate instruction.

The initial constant has been chosen in such a way that the constant in the final round is always $\texttt{4b}_h$. This means the current round constant can be used to check if we've reached the final round. This saves us the instructions necessary to keep track of the round number.

As mentioned before, the substitution layer shuffles the state words into different registers and this needs to be resolved during the linear diffusion layer. The first step is to store one state word in temporary registers. That frees up the two registers corresponding to another state word. We compute the linear diffusion layer for the state word that should end up in those registers, thereby freeing another pair of registers, the ones this new state word was stored in.

We repeat this five times, and at the end, we use the state word that was stored in the temporary registers. Figure 8 shows the inner loop for the entire round. The `xorror` macro's first two arguments are the registers where the result will be stored and the next two arguments are the registers that are freed up.

4 Benchmarks

In this section, we will look at the expected and measured performance of the permutation loop in detail. In principle, because this is the innermost loop, the optimizations done here should have the most effect. We focus on the inner loop and leave the outer loop for future work as it is relatively easy to optimize, and therefore not as interesting.

4.1 Register usage

Before looking at our implementations speed, it is interesting to look at the number of registers our implementation uses. Although variations exist, most RISC-V microarchitectures offer 31 general-purpose registers. One of these is always needed to point to the stack, leaving us with 30 registers for our implementation. We will still try to keep register usage to a minimum however, so that the techniques used in this implementation may be easily ported to other microarchitectures.

The 320-bit Ascon state needs ten 32-bit registers at all times in order to avoid loading and storing it from memory. Our implementation of the substitution layer uses three temporary registers during its computation. Because these are temporary registers, they are available again after the substitution layer finishes.

Our linear diffusion layer uses the same three temporary registers for its own computation, however, because the substitution layer moved the state words

```
round:
        # Addition of round constant
        xor a2, a2, t5

        # Substitution layer
        sbox a0, a1, a2, a3, a4, t3
        sbox s0, s1, s2, s3, s4, t4

        # Linear diffusion layer
        xorror a0, s0, a2, s2, a2, s2, 19, a2, s2, 28
        xorror a2, s2, a4, s4, a4, s4, 1,  a4, s4, 6
        xorror a4, s4, a1, s1, a1, s1, 7,  s1, a1, 9
        xorror a1, s1, a3, s3, s3, a3, 29, s3, a3, 7
        xorror a3, s3, t3, t4, t3, t4, 10, t3, t4, 17

        # Compute next round constant
        addi t5, t5, -15

        # Loop back if final round has not been reached
        bne a5, t5, round
```

Figure 8: The main permutation loop that loops over all rounds in a permutation, defined in terms of the `sbox` and `xorror` macros. `t5` contains the round constant, `a0` through `a4` contain the least significant halves of the five state words, `s0` through `s4` contain the most significant halves of the five state words. `t0` through `t4` are used as temporary registers and `a5` contains the round constant where the loop should end, $\texttt{3c}_h$.

Figure 9: Register allocation during the permutation

Purpose	Registers used
Ascon state	10
Temporary	3
Shuffle into place	2
Current round constant	1
Final round constant	1
Total	17

around, two more registers are needed to move them back into place again.

The current round constant uses another register and the final round constant uses one more. Because the final round constant is small and constant, it can be traded for a single-cycle instruction to generate it, but on RISC-V, there is no need.

In total, we use 17 registers out of 30, which leaves 13 registers to implement the outer loop. This should allow the implementation of the outer loop to avoid loading or storing anything but the inputs and outputs of the algorithm. Figure 9 gives an overview of these allocations.

4.2 Expected performance

The substitution layer was optimized by using different binary formulas to compute the S-box. The original formulas used 22 single-cycle instructions while the improved formulas use 17 single-cycle instructions. We have made an attempt to prove this is the smallest number possible, but the method we used was not sufficiently powerful.

Because we can only operate on machine words and the Ascon state words are twice the size of a machine word, we expect the substitution layer to take 34 cycles.

As described in the section on optimization, the linear diffusion layer must take at least 80 cycles. We give an implementation that reaches this minimum, while also moving the state words back into their original registers, to reverse the shuffling from the substitution layer. This is possible thanks to RISC-V's abundant number of registers.

Finally, the addition of the round constant takes a single cycle, and its computation from the previous round constant takes one more. This round constant is also used to terminate the loop, by comparing it to the final round constant. As long as this constant is loaded in a register, comparing to it and jumping back should take on cycle. Of course, due to branch misprediction, this may take more cycles in the first few rounds and at the end of the last round. On all other rounds, the addition of the round constant and the loop together should take 3 cycles. Summing these, one round of the permutation is expected to take 117 cycles.

4.3 Method

We take several measures in order to get highly reliable benchmarks. First of all, we run the processor at a low frequency, approximately 18 MHz, much lower than this processor's maximum frequency of around 300 MHz. This assures other components, like memory and instruction cache can keep up. Second, we run each benchmark 32 times. We report the median of these results.

Next, in order to test only the speed of rounds after the branch predictor and other factors have stabilized, we report the difference between the timings of running the target number of rounds and twice that number of rounds in a single permutation.

Finally, we choose the number of permutations to be the product of several small factors that could be the periods of various kinds of fluctuations. We use 120 rounds, which is well above realistic usage of Ascon, but the computation remains the same, so the speed should not be affected. After taking all these measures, we find the results are completely consistent between runs.

4.4 Results

Figure 10 shows the measured cycles for a single round of the Ascon permutation. While a single round was expected to take 117 cycles, in practice it takes 118 cycles. It is unclear what causes this. The most likely explanation is that the compare-and-jump-if-not-equal instruction used at the end of the loop takes two cycles rather than one. Unfortunately, the documentation from SiFive does not specify clear cycle counts for each instruction.

Figure 11 shows the resulting increase in speed for encryption and decryption across the different implementations. To encrypt 4096 bytes using Ascon128, it is first padded to 4104 bytes and then split into 513 blocks. To separate these blocks, 6 rounds of the permutation are run between them, 512 times in total. Initialization and finalization cost another 12 rounds each, for a total of 3096 rounds. These rounds cost 365328 cycles, before branch mispredictions. This means encryption spends 66% of its time on the Ascon permutation and decryption spends 76% of its time on the permutation. This difference is due to the time spent copying and padding the plaintext during encryption, which is not needed during decryption.

5 Discussion

We have optimized the Ascon internal loop for the RISC-V RV32IMAC architecture. While doing this, we have noted that both Ascon and RISC-V are very simple and have little room for optimization. Nonetheless, we have found improvements in the boolean formulas used to compute the substitution layer and an oppertunity to combine the round constant with the loop counter. We have

Figure 10: Number of cycles each phase of an Ascon permutation round takes after the branch predictor has stabilized.

Phase	Expected	Measured
Substitution layer	34	34
Linear diffusion layer	80	80
Addition of constant and loop	3	4
Total	117	118

Figure 11: Number of cycles for encrypting and decrypting 4096 bytes for different implementations.

Implementation	Encryption	Decryption	Relative speed
Reference implementation	701340	627863	100%
Big endian state	612095	538205	116%
Inner loop in assembly	552076	478262	129%

also switched the representation of the state to big endian in order to reduce the number of endian conversions necessary. Our improvements have provided an increase in speed of 29% relative to the reference implementation.

There is still room for more improvement however. We have omitted to optimize the Ascon outer loop. Because RISC-V offers plenty of registers, writing the outer loop in assembly makes it possible to avoid loading and storing the state to memory altogether. It is also possible that the boolean formulas of the substitution layer are not yet optimal. Finding and proving the optimal formulas will require some careful analysis however, as the search-space is quite large.

Finally, it is important to note that our optimizations are quite general and should be portable to other architectures. Our new boolean formulas do not only improve speed from 22 to 17 cycles, but also register usage from 5 to 3 temporary registers. This makes it plausible to implement the substitution layer without needing the stack on 32-bit ARM architectures which often only have 14 registers available for implementation. It is also of interest to improve our implementation using the RISC-V bit manipulation extension, once it is frozen and more widely available.

References

[1] H. Krawczyk, "The order of encryption and authentication for protecting communications (or: How secure is ssl?)," in *Advances in Cryptology —*

CRYPTO 2001 (J. Kilian, ed.), (Berlin, Heidelberg), pp. 310–331, Springer Berlin Heidelberg, 2001.

[2] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 531–545, Springer, 2000.

[3] "Caesar: Competition for authenticated encryption: Security, applicability, and robustness." `https://competitions.cr.yp.to/caesar.html`.

[4] "Nist lightweight cryptography." `https://csrc.nist.gov/projects/lightweight-cryptography`.

[5] W. Diehl, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Face-off between the caesar lightweight finalists: Acorn vs. ascon." Cryptology ePrint Archive, Report 2019/184, 2019. `https://eprint.iacr.org/2019/184`.

[6] A. Adomnicai, J. J. Fournier, and L. Masson, "Masking the lightweight authenticated ciphers acorn and ascon in software." Cryptology ePrint Archive, Report 2018/708, 2018. `https://eprint.iacr.org/2018/708`.

[7] N. Bangma, "Ascon: An attempt in NEON on the Cortex-A8." Bachelor's thesis at Radboud University, 2018.

[8] R. Ankele and R. Ankele, "Software benchmarking of the 2nd round caesar candidates." Cryptology ePrint Archive, Report 2016/740, 2016. `https://eprint.iacr.org/2016/740`.

[9] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2." Submission to the CAESAR competition: `https://competitions.cr.yp.to/round3/asconv12.pdf`, 2016.

[10] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[11] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, (Los Alamitos, CA, USA), pp. 122–126, IEEE Computer Society Press, 1987.

[12] K. Stoffelen, "Optimizing S-Box implementations for several criteria using SAT solvers," in *Fast Software Encryption* (T. Peyrin, ed.), (Berlin, Heidelberg), pp. 140–160, Springer Berlin Heidelberg, 2016.

[13] E. Dolstra, M. de Jonge, and E. Visser, “Nix: A safe and policy-free system for software deployment,” in *Proceedings of the 18th USENIX Conference on System Administration*, LISA ’04, (Berkeley, CA, USA), pp. 79–92, USENIX Association, 2004.

Appendices

A Reproducability

Nix [13] is a functional package manager designed to make its packages highly reproducable. As a package manager, it cannot eliminate entropy from concurrency and some other sources, so multiple builds of the same package are not necessarily byte-for-byte identical. Instead, it tries to eliminate entropy by specifying almost all direct and indirect dependencies with high precision.

Packages are specified by *derivations*, which specify a shell script that builds the package, any relevant environment variables, and a list of sources and package dependencies the build needs. These derivations are cryptographically hashed, and the resulting hash uniquely identifies the package. Whenever the version or configuration of a package or one of its dependencies changes, the resulting hash will be different, which means it will be a different package.

In order to make our results reproducable, we use Nix to specify precisely what compiler was used to compile the binaries for the board. We do this by creating a pseudo-package, which specifies dependencies, but no build instructions. The `nix-shell` command was designed to use this: It takes a package and starts a bash session with all dependencies injected into the `PATH`, in the same way it would when building the package. The code used for this thesis is available online.[1]

B Code listing

B.1 `permutation.s`

```
# This macro computes boolean formulas of the substitution layer in
# parallel on the registers s0 through s4, using t0, t1 and t2 as
# temporary registers. This macro must be run twice with different
# state registers, because each 64-bit state word is split in two
# 32-bit machine words. Note that after this macro, the state words
# end up in different registers. In order to make space to move the
# state words back into position, the final result is stored in
# register r0 instead.
.macro sbox s0, s1, s2, s3, s4, r0
```

[1] https://github.com/Lucus16/ascon-riscv

```
        xor t0, \s1, \s2
        xor t1, \s0, \s4
        xor t2, \s3, \s4
        not \s4, \s4
        or \s4, \s4, \s3
        xor \s4, \s4, t0
        xor \s3, \s3, \s1
        or \s3, \s3, t0
        xor \s3, \s3, t1
        xor \s2, \s2, t1
        or \s2, \s2, \s1
        xor \s2, \s2, t2
        not t1, t1
        and \s1, \s1, t1
        xor \s1, \s1, t2
        or \s0, \s0, t2
        xor \r0, \s0, t0
.endm

# The xorror macro applies the rotates and bitwise additions of the
# linear diffusion layer to a single state word. Because of the
# limitations of assembly macros, it takes many arguments: dl and dh
# are the registers for storing the low and high part of the result.
# sl, sh, sl0, sh0, sl1 and sh1 are the source registers for the state
# word, without rotation, with rotation r0 and with rotation r1
# respectively. Because 32-bit shift instructions can only shift by 31
# at most, representing rotate amounts above 32 is done by subtracting
# 32 and swapping the respective source registers. This is why the
# source registers must be supplied three times.
.macro xorror dl, dh, sl, sh, sl0, sh0, r0, sl1, sh1, r1
        slli t0, \sl0, (32 - \r0)
        srli t2, \sh0, \r0
        xor t0, t0, t2
        slli t2, \sl1, (32 - \r1)
        xor t0, t0, t2
        srli t2, \sh1, \r1
        xor t0, t0, t2
        slli t1, \sh0, (32 - \r0)
        srli t2, \sl0, \r0
        xor t1, t1, t2
        slli t2, \sh1, (32 - \r1)
        xor t1, t1, t2
        srli t2, \sl1, \r1
        xor t1, t1, t2
        xor \dl, \sl, t1
        xor \dh, \sh, t0
```

```
.endm

.text

.globl permutation
.align 2
permutation:
        # a0 state pointer
        # a1 round constant

        addi sp, sp, -20
        sw s0, (sp)
        sw s1, 4(sp)
        sw s2, 8(sp)
        sw s3, 12(sp)
        sw s4, 16(sp)

        mv t6, a0
        li a5, 0x3c
        li t5, 0xf
        mul t5, t5, a1
        add t5, t5, a5

        # t6 state pointer
        # t5 round constant
        # a5 final round constant

        lw a4, (t6)
        lw s4, 4(t6)
        lw a3, 8(t6)
        lw s3, 12(t6)
        lw a2, 16(t6)
        lw s2, 20(t6)
        lw a1, 24(t6)
        lw s1, 28(t6)
        lw a0, 32(t6)
        lw s0, 36(t6)

        # a0-a4 lower halves of state
        # s0-s4 upper halves of state

# The main permutation loop that loops over all rounds in a
# permutation, defined in terms of the sbox and xorror macros. t5
# contains the round constant, a0 through a4 contain the least
# significant halves of the five state words, s0 through s4 contain
# the most significant halves of the five state words. t0 through t4
# are used as temporary registers and a5 contains the round constant
# where the loop should end, 0x3c.
round:
        # Addition of round constant
        xor a2, a2, t5

        # Substitution layer
        sbox a0, a1, a2, a3, a4, t3
        sbox s0, s1, s2, s3, s4, t4

        # Linear diffusion layer
        xorror a0, s0, a2, s2, a2, s2, 19, a2, s2, 28
        xorror a2, s2, a4, s4, a4, s4, 1,  a4, s4, 6
        xorror a4, s4, a1, s1, a1, s1, 7,  s1, a1, 9
        xorror a1, s1, a3, s3, s3, a3, 29, s3, a3, 7
        xorror a3, s3, t3, t4, t3, t4, 10, t3, t4, 17

        # Compute next round constant
        addi t5, t5, -15

        # Loop back if final round has not been reached
        bne a5, t5, round

        sw a4, (t6)
        sw s4, 4(t6)
        sw a3, 8(t6)
        sw s3, 12(t6)
        sw a2, 16(t6)
        sw s2, 20(t6)
        sw a1, 24(t6)
        sw s1, 28(t6)
        sw a0, 32(t6)
        sw s0, 36(t6)

        lw s0, (sp)
        lw s1, 4(sp)
        lw s2, 8(sp)
        lw s3, 12(sp)
        lw s4, 16(sp)
        addi sp, sp, 20

        ret
```