The Challenge of Security in IoT – A Study of Cryptographic Sponge Functions and an Implementation for RIOT

Adrian Herrmann
Freie Universität Berlin

February 19, 2018
(Revision of March 23, 2018)

Bachelor Thesis
Supervisor: Dr. Emmanuel Baccelli

Contents

The growing market of the Internet of Things presents many possibilities but also poses challenges that must be addressed. One of these is security and in particular secure communication. This thesis aims to give an introduction to this field by studying the cryptographic sponge functions, which are the basis of the modern cryptographic hashing algorithm Keccak/SHA-3, and by presenting an implementation for low-end IoT devices running the open-source IoT operating system RIOT. Additionally, the process of upstreaming this implementation to the software repository of RIOT will be detailed.

1. Introduction and Motivation

1.1. IoT – Opportunities and challenges

The *Internet of Things* - commonly referred to as IoT - is a rather broad term for a trend in the computer electronics market during recent years where an increasing number of physical devices and "things" embedded with microelectronics and sensors or actuators is connected to the Internet in order to enable them to collect and exchange data. Such devices would typically be equipped with very low-resource and low-power hardware - in some cases, an object that could be an IoT device today would not have had any hardware at all until recently. Classic examples of IoT devices that may often come up in contemporary mainstream conversations about this subject include objects such as a "smart fridge" that keeps track of its contents and may even be able to automatically order groceries that are about to run out, or a smart thermostat. However, the field of IoT is basically arbitrarily broad, as almost any domain has possible applications for connected hardware that runs with limited resources, with the benefit of low cost and a high system uptime. Consider e.g. small low-powered microelectronics boards or MCUs (*Microcontroller units*) with embedded or externally connected sensors that can be deployed on a large scale to gather distributed environmental sensor data. The possibilities are basically endless, and it is therefore not much of a wonder that the Internet of Things is predicted by several market research institutes to grow exponentially in the coming years on several metrics, be it revenue, investment or install base [1], which was predicted to be over 6.5 billion devices in 2016, according to online statistics and market research portal *Statista* [2].

The special hardware environment of an IoT device presents a number of opportunities but also challenges for the software that it runs, most of which stem out of the severe hardware limitations. IoT devices may run with as little RAM as in the single-digit kilobyte range, less than 100 KB of flash memory and a base clock in the low MHz range. The benefit of this low-end hardware is that it's also extremely low-powered, making it possible for such a device to run for months or even years on end with one single battery. This makes large-scale and low-maintenance deployments possible, which would make sense in e.g. the aforementioned scenario of distributed environmental sensor data collection and many others. The challenge of this hardware is writing software in such a way that it is efficient yet sufficiently fast. There is probably no type of software for an IoT device that would be exempt from this challenge but there are

two particularly interesting cases we want to focus on for this thesis: The first being the problem of providing low-end IoT hardware with software enabling secure communication that is also efficient and fast, and the second being the choice of the underlying software platform, i.e. the operating system. This will be discussed in section 3.1.

1.2. Security in IoT

Security is one of the key concerns of computer science but it's particularly important for the Internet of Things. The special hardware environments pose additional challenges. Typical low-end IoT hardware may lack a *Memory Management Unit* (MMU)[1], widening the attack vector presented by stack or buffer overflows. The limited performance and memory must be taken into consideration when designing secure provisioning and update deployment chains, a problem only furthered by software fragmentation caused by a large variety of hardware configuration. The numbers at which IoT devices may be deployed in the future and are already in use today could enable malicious parties to execute attacks at unprecedented scales. Cisco proposes a framework for securing the Internet of Things, for which they identify a number of security challenges, including but not limited to the lack of physical security (see e.g. the lack of MMU), physical protection and tamper detection and, of particular interest, crypto resilience, meaning that devices may outlive the algorithm lifetimes of the cryptographic suites they're equipped with [4].

While security is a very broad subject with many different facets, at the core of many of them lies the problem of secure communication. Today's widespread encryption and communication schemes rely on e.g. AES (Advanced Encryption Standard) for symmetric encryption, RSA for asymmetric encryption, the Diffie-Hellman protocol for secure key exchange and SHA-2 for cryptographic hashing and integrity verification of communications. While these algorithms have proven to be robust over years of analysis and widespread real-word usage, a software environment on a low-end IoT device could be faced with the problem of these cryptographic suites possibly requiring too many resources that the low-end hardware does not provide, or requiring too many calculations for the device to perform at sufficient and satisfying speed. Consequently, alternatives should be sought that not only need to be as secure as the algorithms already in use but also perform well within the restrictions of a low-end hardware environment. Additionally, keeping in mind the issue of crypto resilience identified by Cisco, cryptographic suites for IoT devices should be as future-proof as possible at the time of deployment, considering e.g. a scenario where a working and sufficiently large quantum computer could be developed in functional hardware, breaking today's popular asymmetric public-key algorithms (a problem that the field of post-quantum cryptography works to solve).

In 2007, the *United States National Institute of Standards and Technology* (NIST) issued a call for a new hashing algorithm to be called SHA-3 (*Secure Hashing Algorithm*), when it was thought that the security of the then-newest hashing standard SHA-2 may be threatened after feasible attacks were developed for the MD5 and SHA-1 hashing

[1] [3], page 4

algorithms (the first known instance of a SHA-1 collision was eventually provided by researchers at Google in February 2017[2]). Five years later, an algorithm called *Keccak*, developed by Guido Bertoni, Joan Daemen and Gilles Van Assche of STMicroelectronics and Michaël Peeters of NXP Semiconductors, was chosen among 64 applications [5]. To quote NIST on their announcement, "the NIST team praised the Keccak algorithm for its many admirable qualities, including its elegant design and its ability to run well on many different computing devices. The clarity of Keccak's construction lends itself to easy analysis [...], and Keccak has higher performance in hardware implementations than SHA-2 or any of the other finalists." We will attempt to measure the stated performance advantage over SHA-2. The Keccak algorithm is based on the concept of the *cryptographic sponge construction*, which will be explored in section 2.1.

1.3. Contributions of this thesis

The contributions of this thesis are as follows:

1. It aims to provide an understanding of the Keccak algorithm on a theoretical level by explaining the theory behind the sponge construction.
2. The Keccak algorithm is examined on a practical level by providing a comparison of performance and memory usage of Keccak-based hashing compared to SHA-2 with a focus on low-resource hardware such as IoT devices.
3. The thesis presents an implementation bundling the original Keccak implementation and the open-source IoT operating system *RIOT*.
4. The entire IT industry is highly dependent on collaboration and interoperability of different code bases by different authors. This is especially true of open-source software projects where contributions need to be merged together while maintaining functionality, compatibility and existing coding and styling conventions. An example of this not always straightforward process is given in this thesis as steps and challenges of upstreaming a package for a Keccak suite to the code base of RIOT will be presented in detail.

2. The Keccak Algorithm

2.1. The cryptographic sponge construction

2.1.1. Overview

The cryptographic sponge construction, introduced by the researchers behind Keccak, is a cryptographic mode of operation that generalizes both hash functions and stream ciphers, insofar that it accepts an input and returns an output of variable length, as opposed to the fixed-length output of hash functions and the fixed-length input of stream

[2]The first collision for full SHA-1 (https://shattered.io/static/shattered.pdf)

ciphers. This allows sponge constructions to be used as the basis for cryptographic primitives. Originally, the sponge construction was designed as a reference to make security claims, which will be explained in more detail in section 2.1.2. The basis of the sponge construction is a single permutation or transformation that is applied iteratively on a state of fixed width. Sponge constructions can be classified by the specific permutation or transformation in use and a number of parameters such as the state width. The permutations used by the Keccak algorithm will be detailed in section 2.2.1.

Definition 2.1 (Sponge construction)**.** *Let the sponge construction be defined by a function*

$$\begin{array}{rccc} sponge: & \mathbb{Z}_2^* & \longrightarrow & \mathbb{Z}_2^\infty \\ & M & \longmapsto & Z \end{array}$$

with input M of variable length and output Z and let $f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$ be a permutation or transformation where $b = r + c$ and b is the aforementioned width, r is called the bit rate *and c is called the* capacity*.*

Note that the co-domain of *sponge* is not $\mathbb{Z}_2^*$ but $\mathbb{Z}_2^\infty$, which means that the output Z will be, in principle, of infinite length. It follows that mathematically, any desired output length can be achieved by a truncation, and by a termination of the sponge construction's execution in implementation practice.

If $|M|$ is not a multiple of r, a *padding rule* is required:

Definition 2.2 (Padding rule)**.** *Let M be a message of arbitrary size and x any integer. Then $M_{pad} = M \circ pad[x](|M|)$ denotes the padding of a message M under a padding rule pad to a sequence M_{pad} of x-bit blocks. pad describes the pattern under which bits are to be appended to M such that $|M_{pad}|$ is a multiple of x.*

Bertoni, Daemen and Van Assche define the following criterion for a padding rule to be *sponge-compliant*[3]:

Definition 2.3 (Sponge-compliant padding rule)**.** *A padding rule pad is* sponge-compliant *if it never results in the empty string and if*

$$\forall n, r \geq 0, \forall M, M' \in \mathbb{Z}_2^* : M \neq M' \implies M \circ pad[r](|M|) \neq M' \circ pad[r](|M'|) \circ 0^{nr}.$$

This means that a padding rule is sponge-compliant if no two different bit strings exist that would be padded to the same result string under said padding rule, even if any number of r-sized blocks of 0's were appended to the end of one of the result strings. The simplest sponge-compliant padding rule, denoted by $pad10^*$, appends a 1 followed by the minimum number of 0's such that $|M \circ pad[r](|M|)|$ is a multiple of r. That means that under this padding rule, a minimum of one bit would always be appended regardless of whether $|M|$ is already a multiple of r (in which case the string 10^{r-1} would be appended). Otherwise there would be a string A with a length divisible by r for which a shorter string B would exist such that $A = B \circ pad[r](|B|)$.

[3] [6], page 11

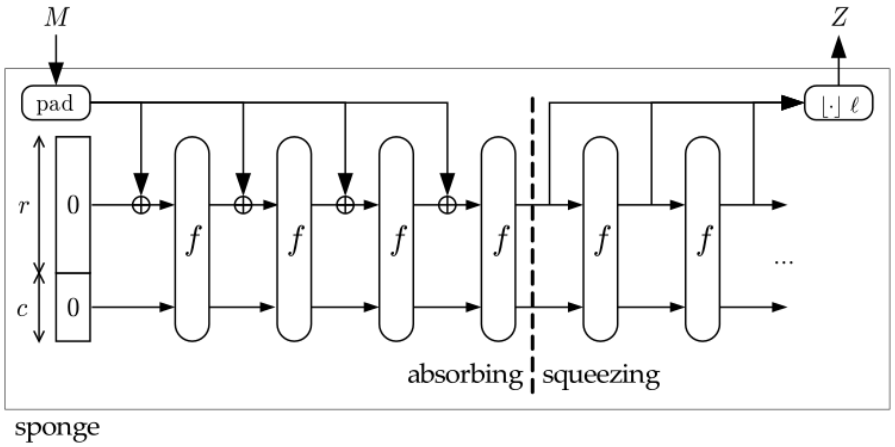

Figure 1: The sponge construction ([6], page 13)

As shown in figure 1, the sponge construction has a state width of $b = r + c$ bits and consists of two phases. During the *absorbing phase*, r bits from the padded input message are XORed to the first r bits of the current state before the permutation $f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$ is applied to the entire now-current state. This way, a new state is obtained with each iteration, where the first state before all iterations is a bit string consisting entirely of 0's. This is done as long as r-bit blocks of the input are left to be absorbed. When all have been absorbed, the *squeezing phase* begins, where the first r bits of each then-current state are appended to the output string and a new state is obtained through application of the permutation f. This is repeated as often as is necessary to obtain the desired output length.

Worth a mention but beyond the scope of this thesis is the very similar *duplex construction* where there are no separate absorbing and squeezing phases but instead the absorption of an r-bit input message block is immediately followed by appending the first r bits of the current state to the output string. The duplex construction is of equivalent security to the sponge construction.

2.1.2. Security claims

The arbitrary length of a hash function's input and the fixed length of its output (also called *digest*) makes the function's co-domain necessarily smaller than the input domain, inevitably leading to collisions where two input strings exist with the same digest. The resistance of a cryptographic hash function against an attack is expressed relative to the digest length as the worst-case number of tries required for a brute-force attacker to succeed. Cryptographic hash functions should offer resistance against the following attacks:

1. *Pre-image attacks* where an attacker attempts to find a message m for a given hash h such that h is the digest of m.

2. *Second pre-image attacks* where given an input m_1 an attacker attempts to find a second message m_2 (not equal to m_1) such that both messages have the same digest.

3. *Collision attacks* where an attacker attempts to find two messages that have the same hash.

The *pre-image attack* resistance should be 2^n for a digest length of n because assuming an uniform distribution of the output set with a cardinality of 2^n, the probability to guess a right message for any given hash should be $\frac{1}{2^n}$. Likewise, the resistance against a *second pre-image attack* should also be 2^n for a digest length of n because the probability to guess the right pre-image for any digest (by guessing a message and comparing its digest with the given one) should be $\frac{1}{2^n}$. The resistance against a *collision attack* (where an attacker calculates many variations of an authentic and a fake message until a message pair with identical hashes is found) should be $2^{\frac{n}{2}}$ because that is how many tries (i.e. new message pairs) should be needed for a total number of 2^n possible digests of length n as a result of the birthday problem.

Following from this, we can define that a hash function has a *security level* of k bits if its smallest attack resistance is 2^k.

When designing a cryptographic function that makes use of cryptographic primitives, security claims need to be made, which is usually done with regard to a model. The advantage of this is that it makes security analysis easier and more compact than if all the resistance properties (and the corresponding levels of resistance) of the cryptographic function were to be listed and analyzed separately. A popular model is the *random oracle model*. A random oracle is an *oracle machine* that produces an output randomly chosen (but fixed for each unique input) from its uniformly distributed output domain. A random oracle satisfies the attack resistances described above so an ideal hash function should behave as close to a random oracle as possible and when making security claims of a cryptographic function, hash functions are often assumed to behave like a random oracle with respect to classical attacks. A hash function is considered broken when an attack is found with a complexity lower than that required against a random oracle. However, the Keccak team argues that this assumption is no longer reasonable [7]. Most hash functions are iterated functions, which have the advantage of being able to hash a message blockwise as they come in without the need for a temporary storage. Iterated functions use finite memory for their state, which summarizes the input blocks received so far at any point in time. Due to the finite memory of the state, collisions cannot only happen in the output but also in the state after any number of iterations (called inner collisions), which means that two messages M_1 and M_2 can exist such that $M_1 \circ N$ and $M_2 \circ N$ have the same hash for any suffix N. This is the point where the random oracle models fails, as while it does account for collisions in the output, it does not account for collisions in the state since this concept does not exist for random oracles. Consequently, any attacks on the state of an iterated function that are possible because of its finite size are left unconsidered by the random oracle model, so an iterated hash function can never

achieve the attack resistance of a random oracle and must thus necessarily be considered broken by the random oracle model.

As an alternative to the random oracle model, the Keccak team proposes a new model based on *random sponges*. A random sponge is a sponge function with a random transformation or permutation.[4] The Keccak team has proven that a random sponge with a sponge-compliant padding rule returns output bits that are uniformly and independently distributed as long as there are no inner collisions.[5] This means that random sponges only differ from random oracles by the possibility of inner collisions. This leads to a difference in attack resistance that depends on the nature of the attack and the parameters of the random sponge like bit rate and capacity, of which the capacity has the greatest impact. Said difference (called the *random oracle differentiating advantage*) can be upper bounded by the worst case. This upper bound depends only on the capacity of the sponge function [7]. To formalize the upper bound, the Keccak team formulates the *flat sponge claim* that goes as follows:

Definition 2.4 (Flat sponge claim[6])**.** *For a capacity c the success probability of an attack on a random sponge should be at most*

$$p + 1 - exp(N^2 2^{-(c+1)})$$

where p is the success probability of the same attack on a random oracle and N is the number of calls to the sponge function or its inverse for which the workload is equivalent to that of the attack.

This flat claim that only takes the capacity as a parameter makes security claims with regard to the random sponge model significantly easier than were all parameters taken into account.

The random oracle differentiating advantage is upper bounded by $N^2 2^{-(c+1)}$ for a capacity c and a fixed number of N calls to the sponge construction's permutation/ transformation f or its inverse.[7] Consider an attack on a random oracle for which the attack resistance is 2^k and the success probability is $q2^{-k}$ for a number q of messages tried. Then the success probability of this attack on a random sponge is upper-bounded by

$$N^2 2^{-(c+1)} + q2^{-k}. \tag{1}$$

For $c > 2k$, it follows that

$$N^2 2^{-(c+1)} < N^2 2^{-(2k+1)} \tag{2}$$

and for growing k and fixed q and N it asymptotically holds that

$$N^2 2^{-(2k+1)} < q2^{-k}. \tag{3}$$

[4] [6], page 13
[5] [6], Theorem 5, page 54
[6] [6], page 73
[7] [6], page 68

From the equations (2) and (3) above it follows for growing k and fixed q and N that

$$N^2 2^{-(c+1)} < q2^{-k}. \tag{4}$$

Therefore, the attack success probability against a random sponge upper-bounded by equation (1) is close to $q2^{-k}$, which is the success probability for the attack on a random oracle. Consequently, a random sponge offers a similar attack resistance to a random oracle if $c > 2k$. Recall that we considered an attack against which a random oracle had a resistance of 2^k. It follows that Keccak has an attack resistance of $2^{\frac{c}{2}}$ for a capacity c, i.e. a security level of $\frac{c}{2}$ bits.

2.2. The Keccak sponge function family

2.2.1. The Keccak permutations

The researchers behind Keccak not only define the generalized sponge construction but also specific sponge permutations Keccak-f for the SHA-3 standard. They define seven permutations of widths $b \in \{25, 50, 100, 200, 400, 800, 1600\}$.

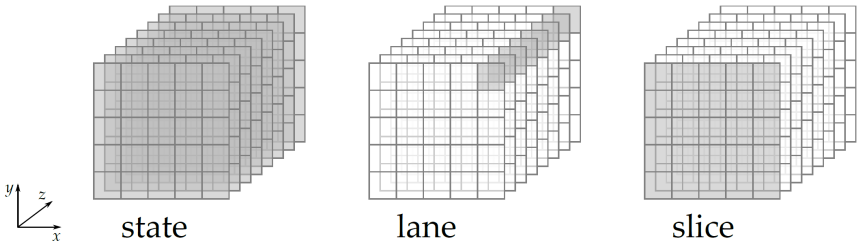

Figure 2: The Keccak permutation state ([8], slide 15)

Each permutation has a state of $5 \times 5 \times 2^l$ bits (with $l \in \{0, ..., 6\}$) which we describe by a matrix A of that size. It consists of 5×5 lanes with 2^l bits or 2^l slices of (5×5) bits each, as shown in figure 2. There are $12 + 2l$ rounds to the permutation, each of which computes a function $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$ where the different components fulfill different purposes:

1. θ (see algorithm 1) is a linear mapping that is translation-invariant in all directions and diffuses the state. Each bit $A[x, y, z]$ is diffused (i.e. a new $A'[x, y, z]$ is calculated) by adding the bitwise sum of the parities of the columns $A[x - 1, \cdot, z]$ and $A[x + 1, \cdot, z - 1]$ to it, as depicted in figure 3. θ is the first component of the round function in order to mix between the inner and outer parts of the state (i.e. the first r bits and the last c bits) early on.

2. ρ (see algorithm 2) disperses between slices through a cyclic shift of lanes with offsets $\frac{i(i+1)}{2}$ mod 2^l where i is the index in the lane.

   The xy-origin is located at the center of the slices.

Algorithm 1 θ

for x = 0 to 4 **do**
  C[x] = A[x,0,·]
  for y = 1 to 4 **do**
    C[x] = C[x] ⊕ A[x,y,·]
  end for
end for
for x = 0 to 4 **do**
  D[x] = C[x-1] ⊕ ROT(C[x+1], 1)
  for y = 0 to 4 **do**
    A'[x,y,·] = A[x,y,·] ⊕ D[x]
  end for
end for

ROT(C,d) denotes a rotation of the values in a lane C by d indices.

Algorithm 2 ρ

A'[0,0,·] = A[0,0,·]
$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
for t = 0 to 23 **do**
  A'[x,y,·] = ROT(A[x,y,·], (t+1)(t+2)/2)
  $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
end for

3. π (see algorithm 3) disperses lanes (therefore it is translation-invariant in the z-axis) through transposition along six axises, as shown in figure 4.

   The xy-origin is again at the center of the slices.

4. χ (see algorithm 4) is a nonlinear mapping applied in parallel and independently to 5-bit rows of a slice that flips a bit if its two neighbors (where the left neighbor of the first bit from the left is the first bit from the right and vice versa) exhibit a 01 pattern, e.g. the 5-bit row 11010 would be mapped to 01010 and 11011 would be mapped to 11001. It is the only nonlinear mapping in R.

5. ι intends to break symmetries by XORing a round-dependent constant vector to the lane in the xy-origin. Otherwise, the function R would be invariant to translation in the z-direction and equal for each round. The round constant vector is obtained through a *linear-feedback shift register* (LSFR). More precisely, the round constant RC_i for the i-th round is given by

$$\begin{aligned} RC_i[0][0][2^j - 1] &= rc[j + 7i] \quad \text{for } 0 \leq j \leq l, \\ RC_i[x][y][z] &= 0 \qquad\qquad\qquad \text{otherwise,} \end{aligned} \tag{5}$$

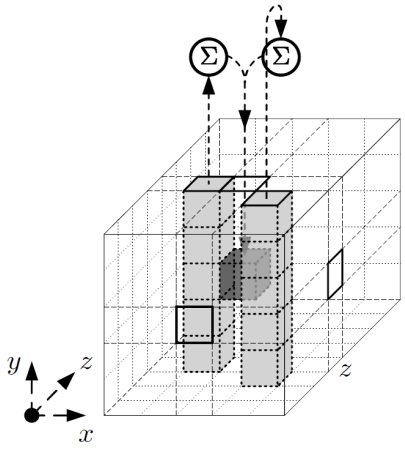

Figure 3: The Keccak mapping θ ([9], page 18)

Algorithm 3 π

for x = 0 to 4 **do**
  for y = 0 to 4 **do**
    $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
    A'[x',y',·] = A[x,y,·]
  end for
end for

where $rc[t] \in GF(2)$ is the output of the LSFR:

$$rc[t] = (x^t \; mod \; x^8 + x^6 + x^5 + x^4 + 1) \; mod \; x \quad \text{in } GF(2)[x]. \tag{6}$$

The components have not only been designed for properties like diffusion and dispersion but also considering possible hardware implementations. χ and θ require only three and two gates per bit respectively (one XOR, one AND and one NOT for χ, two XORs for θ). ρ and π don't require any gates but can be realized with wiring instead. ι can be realized in hardware with a few XORs and circuitry for the LSFR.

2.2.2. The multi-rate padding rule

The Keccak sponge function family uses a specific padding rule (see definition 2.2) called *multi-rate padding*[8]:

Definition 2.5. *The* multi-rate padding *rule, denoted as pad*10^*1*, appends a 1, followed by the minimum number of 0's and a single 1 for the padded message to be a multiple of the rate in length.*

[8] [9], page 7

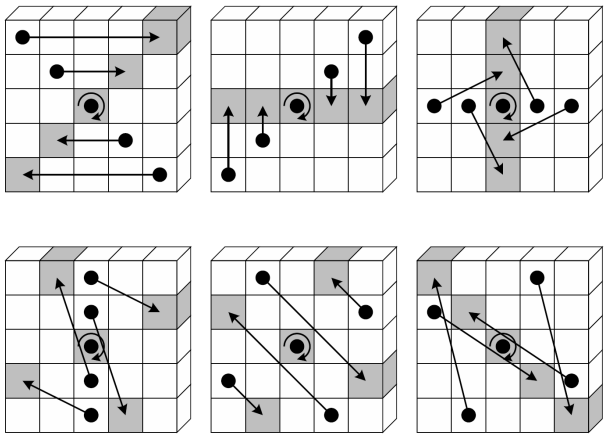

Figure 4: The Keccak dispersion π ([9], page 20)

Algorithm 4 χ

```
for y = 0 to 4 do
    for x = 0 to 4 do
        A'[x,y,·] = A[x,y,·] ⊕ ((NOT A[x+1,y,·]) AND A[x+2,y,·])
    end for
end for
```

Under this padding rule, a minimum of two bits is appended, even if the message is already a multiple of the rate r in length. At most, $r + 1$ bits are appended, which is the case when the length of the message is $\equiv -1 \, mod \, r$.

2.2.3. The SHA-3 standard

The SHA-3 standard consists of the Keccak permutations, the cryptographic sponge construction and different parameter configurations that include the multi-rade padding rule and values for bit rate and capacity. Table 1 details the configurations that are part of the SHA-3 standard. It also includes the respective security levels against collision and pre-image attacks. All parts of the standard are based on Keccak-1600 with a permutation width of 1600 bits.

For variable digest lengths, SHAKE128 and SHAKE256 are not actually cryptographic hash functions but *extendable output functions* (XOFs), a concept that was in fact introduced with the SHA-3 standard. A possible use of XOFs are stream ciphers but they can also be used as hash functions if a fixed hash length is chosen.[9] The different padding rules are to guarantee different hashes for the SHA-3 and SHAKE* variants and consist of the multi-rate padding rule prepended by 01 in the case of SHA3-* and 1111 in the

[9] [10], page 24

Name	Hash length	Rate	Capacity	Padding rule	Collision	Pre-image
SHA3-224	224	1152	448	0110*1	112	224
SHA3-256	256	1088	512	0110*1	128	256
SHA3-384	384	832	768	0110*1	192	384
SHA3-512	512	576	1024	0110*1	256	512
SHAKE128	variable	1344	256	111110*1	$min(\frac{n}{2}, 128)$	$min(n, 128)$
SHAKE256	variable	1088	512	111110*1	$min(\frac{n}{2}, 256)$	$min(n, 256)$

Table 1: The SHA-3 standardization [10]

case of SHAKE*. These are called *domain separation suffixes*.

2.3. MACs and other functionalities beyond hashing

The sponge construction's variable input and output lengths enable it for cryptographic uses beyond hashing, e.g. for the calculation of *message authentication codes* (MACs). A MAC, which is calculated from the original message and a secret key the sender and receiver agreed on previously, is used to verify a message's authenticity and integrity by checking if the MAC calculated by the receiver is the same as the one the sender attached to the message. In principle, a MAC is similar to a hash digest but differs through the inclusion of the secret key in its calculation. Using the sponge construction, a MAC can be calculated by absorbing the secret key as the first part of the input as shown in figure 5, i.e. the input of the sponge becomes $K \circ M$ where K is the key and M is the message.

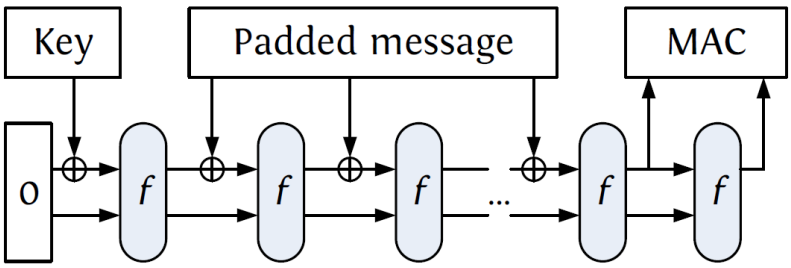

Figure 5: MAC calculation with the sponge construction ([8], page 41)

This is a more straightforward construction for MAC calculation than it is required for hashing functions based on the Merkle-Damgård construction such as SHA-256, as those are vulnerable to *length-extension attacks*. Since the digest of a Merkle-Damgård

hashing function is simply the final state, it would be possible to forge a MAC using the $K \circ M$ construction by hashing a new string X while taking the digest of $K \circ M$ as the initial state (which is equivalent to appending X to M if M is known and calculating the MAC of $M \circ X$, i.e. the hash of $K \circ M \circ X$) and sending the hash of $K \circ M \circ X$ as the MAC for a forged message $M \circ X$.[10] To avoid this, MACs via Merkle-Damgård hashing functions are calculated as *keyed-hash message authentication codes* (HMACs) using a $H(K \circ H(K \circ M))$ construction where H is the hashing function and K and M are as before. Since cryptographic sponges are not vulnerable to length-extension attacks thanks to the squeezing phase, this construction is not required for MAC calculation and the straightforward $H(K \circ M)$ can be used instead.

Other examples of cryptographic functionalities that can be implemented with the sponge construction are *salted hashing* (prepending the salt to the message), *stream ciphers*[11], *single pass authenticated encryption*[12] and *reseedable pseudorandom number generators* (PRNGs) with the duplex construction[13]. The *Strobe* protocol framework[14] is a new framework for cryptographic protocols (first published on January 3, 2017) that offers the flexibility of Keccak in a comprehensive package for deployment to IoT devices. It includes implementations for cryptographic protocol building blocks like *authenticated Diffie-Hellman key exchange*, *password-authenticated key exchange*, *digital signatures* and more.

3. The Embedded IoT Software Platform RIOT

3.1. Overview

There are several requirements that should be satisfied by a flexible and capable IoT operating system. This includes a small memory footprint, support for heterogeneous hardware, network connectivity, energy efficiency, real-time capabilities and security [3]. This is not a trivial task, as an increasing complexity is at odds with the hardware limitations of an IoT device, most notably the limitations of both RAM and flash memory. The *Internet Engineering Task Force* (IETF) has proposed a classification of low-end IoT devices based on memory capacity [11]. While it describes three classes, the lowest class (class 0) features hardware on such a low end that running a proper OS becomes unsuitable[15]. Consequently, a low-end IoT operating system should attempt to satisfy the aforementioned requirements for class 1 devices, which have about 10 KB of RAM and 100 KB of flash memory. A number of open and closed-source operating systems exist but for this thesis we want to focus on the open-source operating system called *RIOT* (refer to [3] for a survey paper about IoT operating system choices).

[10] An alternative would be a $H(M \circ K)$ construction, but this would be broken as soon as a collision attack were found for the underlying hashing function. This would be true of Keccak as well.
[11] [8], page 42
[12] [8], page 43
[13] [8], page 45
[14] https://strobe.sourceforge.io/
[15] [3], page 1

The RIOT developers describe it as "the friendly operating system for the Internet of Things" [12]. It is open-source under the LGPLv2.1 license, allowing redistribution and modification without requiring modified code to be published under the same license. Development started out between *Freie Universität Berlin* (FU Berlin), *Institut national de recherche en informatique et en automatique* (INRIA) and *Hochschule für Angewandte Wissenschaften Hamburg* (HAW Hamburg) and is continued today by an international and open developer community, including the founding institutes. The source code can be found on GitHub.[16] It can be built with standard tools such as gcc, gdb and valgrind, as programming is done in ANSI C and C++. Notable features include a microkernel, partial POSIX compliance (full POSIX compliance being an active development goal), an interrupt latency of around 50 clock cycles and priority-based scheduling (enabling real-time capabilities), multi-threading with a threading overhead of less than 25 bytes per thread, static and dynamic memory allocation, an API for inter-process communication and support for networking technologies on the three lower layers of the Internet protocol suite (6LoWPAN, IPv6, RPL and UDP). Also included are a couple of integrated utilities such as CoAP and CBOR implementations, a system shell and SHA-256 [12] [13], which will be of particular interest for this thesis.

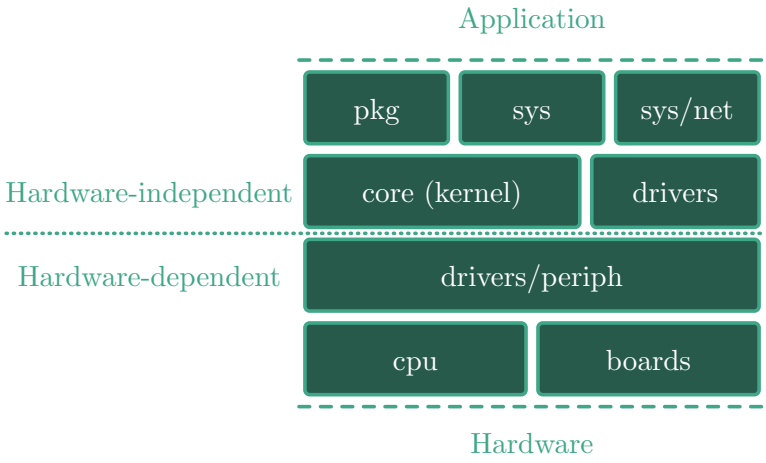

Figure 6: Architecture of RIOT [17]

As can be seen in figure 6, the architecture of RIOT [14] is designed so that the OS itself (i.e. the micro-kernel and modules such as the aforementioned integrated utilities) alongside other hardware-independent code (e.g. external packages) is separated from hardware-dependent code, such as CPU-specific configurations and implementations (including power management, interrupt handling, startup code, clock initialization code and thread handling), board configurations and implementations (e.g. pin mapping and any deviations from the generic configuration of the board's CPU) and drivers for specific peripherals, which are however platform-agnostic and as such only dependent on

[16] https://github.com/RIOT-OS/RIOT
[17] https://riot-os.org/api/riot-structure.svg

the peripheral they're written for.

This architecture is reflected in the source code's directory structure. When deploying a RIOT-based application, the application is built alongside the entire operating system, any modules and packages the application developer may choose to import and the required device drivers (including CPU, board and peripherals), transitively determined by a single variable for the board inside the application's Makefile.

The architecture of RIOT is evidently well abstracted and highly extensible - anyone cannot only develop applications but also new modules, drivers or new packages without breaking or needing to modify any of the existing components. For this thesis, we will develop an application and port a package to RIOT, the process of which will be described in section 3.

A list of pre-existing modules can be found on the website of RIOT's documentation.[18]

3.2. Porting Keccak to RIOT

RIOT is an excellent software platform for putting all the theory about Keccak into practice in the form of functioning and deployable code. Porting the Keccak algorithm, of which an implementation by the Keccak team already exists[19] as a RIOT package allows us to build RIOT applications for IoT devices that make use of the cryptographic sponge construction and cryptographic services built on it, e.g. hashing and MAC calculation. The package should offer an API for both quick and straightforward but also finely tuned uses of Keccak-based cryptographic services, which means that it should offer functions for a sensible pre-selection of input configurations (e.g. the ones in the SHA-3 standard) but also functions that allow the user a high degree of configuration for more advanced use cases.

A pull request for this package was made to RIOT's repository and remains to be merged as of March 23, 2018.[20]

3.2.1. Anatomy of porting a package

A package in RIOT contains libraries or applications from an outside source. Its basic structure is simple: It consists of a *Makefile*, any number (including zero) of *patch files* for changes made to already existing files and any number of additional new code or header files if needed. The Makefile describes how to get the application (e.g. from a Git or SVN repository or simply from an HTTP source), how to build it and how to integrate it as a RIOT module. The patch files are to be applied automatically if changes to the original code base are required in order to make it able to build and run on RIOT. A package can be added to the list of modules to be compiled for a RIOT application by adding the following line to the application's Makefile (where <*pkg_name*> is the package's name):

```
USEPKG += <pkg_name>
```

[18] https://riot-os.org/api/group__sys.html
[19] https://github.com/gvanas/KeccakCodePackage
[20] https://github.com/RIOT-OS/RIOT/pull/7903

If the package provides header files, a way to have the compiler include them is by updating the *INCLUDE* variable inside the application's Makefile (where ... represents any subdirectory, if given):

```
INCLUDE += $(RIOTPKG)/<pkg_name>/...
...
```

Alternatively, a *Makefile.include* file can be added inside the package's directory where the *INCLUDES* environment variable is updated with any subdirectories of the package that include needed header or implementation files:

```
INCLUDES += -I$(PKGDIRBASE)/<pkg_name>/...
...
```

We choose *keccak* as our package's name and include a *Makefile.include* that acts as described. The content of the *Makefile* will be explained in detail in section 3.2.4. The Makefile resides in a subdirectory of the *pkg* directory with the package's name, and the patch files in a subdirectory of the latter called *patches*.

Additional code and header files are placed in a subdirectory of the *keccak* directory called *contrib*. The internal structure of the *contrib* directory is up to each package but it makes sense to have it reflect the directory structure of the source.

In addition, unit tests for the package should be added to RIOT's code base to ensure functionality and compatibility. This will be detailed in section 3.2.5.

In our case, the porting process begins with forking the *KeccakCodePackage* repository from the Keccak team and making any necessary changes to it, which will then be included in the package as new files inside the *contrib* directory (see section 3.2.2) and as patch files that are applied automatically (see section 3.2.3). This way, RIOT can pull the original *KeccakCodePackage* repository when building the package instead of the fork, which makes the package easier to maintain as any future changes could be included in the patch file or in new files instead of needing to update a separate repository. It should be noted that while changes to the repository can be necessary on a technical level, they can also be necessary so that any code that becomes part of the RIOT repository respects its coding and naming conventions. Relevant conventions in this case are, among others, naming in lowercase with underscores instead of CamelCase and avoiding macros in favor of static inline functions wherever possible.[21]

3.2.2. New code and header files

The *Modes* directory of the repository contains implementations and headers for high-level crytographic services such as hashing in the form of the code file *KeccakHash.c* and the header file *KeccakHash.h*. *KeccakHash.h* includes *KeccakSpongeWidth1600.h* from the *Constructions* directory, which is the header file for the sponge construction with a state width of 1600 bits. We want to use the sponge construction with a state width of

[21] https://github.com/RIOT-OS/RIOT/wiki/Coding-conventions

800 bits as well, as we may find that it is better suited for low-powered embedded devices due to its smaller memory footprint. Consequently, we need an additional version of *KeccakHash.h* that includes *KeccakSpongeWidth800.h*. We can do this by adding a copy of *KeccakHash.h* with the modified include statement. We call this copy *keccak800.h* and the implementation *keccak800.c*, which in turn is just a copy of *KeccakHash.c* with a modified include statement and calls to functions from *KeccakSpongeWidth800.h* instead of *KeccakSpongeWidth1600.h*, e.g. *KeccakWidth800_SpongeInitialize()* instead of *KeccakWidth1600_SpongeInitialize()*. In addition to this, we could remove macros for calls to the Keccak initialization function with the SHA-3 configurations from *keccak800.h*, as the SHA-3 standard uses a width of 1600 bits. However, since our package is to offer some functions with pre-selected input configurations for easy use, it makes sense to keep most of said macros, replacing the calls to Keccak-1600's initialize function with calls to that of Keccak-800, and adjust the rate parameter accordingly. This leaves us with the pre-defined input configurations for Keccak-800's initialize function shown in table 2.

Function type	Rate	Capacity	Hash length	Delimited suffix[22]
XOF	544	256	0 (arbitrarily long output)	0x1F
XOF	288	512	0 (arbitrarily long output)	0x1F
Hash	544	256	128	0x06
Hash	352	448	224	0x06
Hash	288	512	256	0x06
Hash	32	768	384	0x06

Table 2: Pre-defined input configurations for Keccak-800's initialize function

Since *KeccakHash.h* or rather the functions declared in it are part of the API offered by the package, it would be good to have its function, variable, enum and struct names be consistent with those of the newly created *keccak800.h*, which in turn is consistent with RIOT's naming convention. This is solved as follows:

1. A new header file *keccak_defs.h* that includes *KeccakHash.h* is created. It provides macros that are consistent with RIOT's naming convention for any variables from *KeccakHash.h* that are needed by both *keccak800.h* and a new header *keccak1600.h* (see below). Consequently, they both include *keccak_defs.h*. This allows us to let them share variables.

2. A new header file *keccak1600.h* is created. It includes *keccak_defs.h* and offers static inline functions acting as aliases for the functions originally declared in

[22]The delimited suffix byte describes the domain separation suffix from the padding rule (see section 2.2.3) and contains the n bits to be appended in least significant bits order and must be delimited with a 1 at position n. For the hashing functions' (SHA3-* for Keccak-1600) domain separation suffix 01 the delimited suffix is 00000110 = 0x06, for the XOFs' (SHAKE* for Keccak-1600) domain separaton suffix 1111 it's 00011111 = 0x1F.

KeccakHash.h and macros for the variables that are not handled by *keccak_defs.h*.

3.2.3. Changes to existing files from the source repository

Next we want to add an appropriate build target for the binary library we want RIOT applications to use. Instead of a regular single Makefile, *KeccakCodePackage* uses an XML file called *Makefile.build* that is expanded into a regular Makefile using the *xsltproc* utility. Build targets can be defined through *target* tags, e.g.

```
<target name="KeccakHash800ARMv6M.a" inherits="KeccakHash800
↪  optimized800ARMv6Mu1"/>
```

or through *groups*, e.g.

```
<group all="Keccak">
   <product delimiter="/">
      <factor set="reference reference32bits compact generic32
      ↪  generic32lc generic64 generic64lc asmX86-64 asmX86-64shld
      ↪  Nehalem SandyBridge Bulldozer Haswell KnightsLanding ARMv6M
      ↪  ARMv7M ARMv7A ARMv8A AVR8"/>
      <factor set="KeccakTests KeccakSum libkeccak.a"/>
   </product>
</group>
```

where the resulting build targets are all the possible concatenations of one *set* attribute from the first *factor* tag and one *set* attribute from the second *factor* tag, separated by the *delimiter* attribute of the *product* tag. A target whose name attribute ends with *.a* is a target for a code library, otherwise it's a target for an executable program.

The *set* attributes inside the *factor* tags and the *inherits* attributes inside the *target* tags are defined through *fragment* tags, e.g.

```
<fragment name="compact" inherits="compact200 reference400 compact800
↪  compact1600 serial-fallbacks Ket-SnP"/>
```

or

```
<fragment name="KeccakSum" inherits="FIPS202 KangarooTwelve">
   <c>KeccakSum/KeccakSum.c</c>
   <c>KeccakSum/base64.c</c>
   <h>KeccakSum/base64.h</h>
</fragment>
```

where the elements inside *c* tags are C code files and the elements inside *h* tags are C header files. Any number of fragments can be inherited and inheritance is transitive. Another possible tag is *gcc*, which will be addressed at a later point.

The *Makefile.build* file from *KeccakCodePackage* also references other XML files through the XInclude mechanism:

```
<xi:include href="HighLevel.build"/>
<xi:include href="LowLevel.build"/>
<xi:include href="Ketje/Ketje.build"/>
<xi:include href="Keyak/Keyak.build"/>
<xi:include href="SUPERCOP/SUPERCOP.build"/>
<xi:include href="HOWTO-customize.build"/>
```

HighLevel.build and *LowLevel.build* in particular are of importance to our case, as they define fragments for the high-level cryptographic services and the low-level primitives provided by *KeccakCodePackage*, respectively.

We want build targets for the following scenarios:

1. A code library to enable hashing with Keccak-800 and Keccak-1600 in RIOT programs on ARMv6M devices,
2. a code library to enable hashing with Keccak-800 and Keccak-1600 in RIOT programs on ARMv7M devices and
3. a code library to enable hashing with Keccak-800 and Keccak-1600 in RIOT programs on the native RIOT board running on an x86 PC.

Those build targets are defined as follows inside *Makefile.build*:

```
<target name="KeccakHashARMv6M.a" inherits="Keccak800
↪  optimized800ARMv6Mu1 Keccak1600 optimized1600ARMv6Mu2"/>
<target name="KeccakHashARMv7M.a" inherits="Keccak800 optimized800u2
↪  Keccak1600 inplace1600ARMv7M"/>
<target name="KeccakHashNative.a" inherits="Keccak800 optimized800u2
↪  Keccak1600 compact1600"/>
```

Keccak800 and *Keccak1600* are defined as follows inside *HighLevel.build* (as they are high-level cryptographic services):

```
<fragment name="Keccak800" inherits="KeccakSpongeWidth800">
   <c>Modes/keccak800.c</c>
   <h>Modes/keccak800.h</h>
</fragment>

<fragment name="Keccak1600" inherits="KeccakSpongeWidth1600">
   <c>Modes/KeccakHash.c</c>
   <h>Modes/keccak1600.h</h>
</fragment>
```

KeccakSpongeWidth800 and *KeccakSpongeWidth1600* are existing fragments inside *HighLevel.build*, and the fragments inherited by the *.a* build targets are existing fragments inside *LowLevel.build*, the choice of which was informed by benchmark results in section

4.3.

The last change to *KeccakCodePackage* pertains to the following fragment inside *LowLevel.build* that defines compiler flags for the *GNU Compiler Collection* inside the *gcc* tags:

```
<fragment name="optimized">
   <h>Common/brg_endian.h</h>
   <gcc>-fomit-frame-pointer</gcc>
   <gcc>-O2</gcc>
   <gcc>-g0</gcc>
   <gcc>-march=native</gcc>
   <gcc>-mtune=native</gcc>
</fragment>
```

This fragment is inherited by all the fragments from before whose names begin with *optimized* or *inplace*. These compiler flags are for building with *gcc* for an x86 device but we want to build for boards that run on the ARMv6M and ARMv7M architectures, for which we would need different build flags and which are built with the *GNU ARM Embedded Toolchain*. However, RIOT's build system creates a variable called $CFLAGS$ that contains all the appropriate compiler flags at runtime, so we can solve this by replacing the *optimized* fragment with

```
<fragment name="optimized">
   <gcc>-O2</gcc>
   <gcc>$(CFLAGS)</gcc>
</fragment>
```

Only *-O2* remains as that is not included in $CFLAGS$. All other flags can be removed. We can also remove the *Common/brg_endian.h* header from the fragment, as it's already included in a fragment called *common* inside *Makefile.build* that is inherited by all build fragments.

All the changes described in this section are added to a patch file in the *patches* directory of the package, which is automatically applied.

3.2.4. The package's Makefile

The package's Makefile needs to define three variables for RIOT's build system to handle:

PKG_NAME: The name of the package, *keccak* in our case.

PKG_URL: The location of the external source, in our case the URL of the package's Git repository.

PKG_VERSION: In our case the commit ID up to which we want to clone the repository. This is to ensure compatibility between RIOT and the package even if changes have been made to its source repository afterwards.

If building the package as part of a RIOT application requires more preparations than only downloading the external source (e.g. building parts of the external source), as is our case, this needs to be defined by the build target as well. We recall that we added three build targets to *KeccakCodePackage* depending on the architecture we want to build for, so what our Makefile needs to do is recognize the architecture and build the corresponding target. We can use the *CPU* and *CPU_ARCH* variables for this, which are set by RIOT's build system during the building process. When building for the native board, *CPU* is set to *native* and when building for an actual board, *CPU_ARCH* is set to the corresponding architecture. We can catch all these cases in the Makefile as follows:

```
ifeq ($(CPU),native)
   CPU_ARCHIVE_NAME := Native
else
ifeq ($(CPU_ARCH),$(filter $(CPU_ARCH), cortex-m0 cortex-m0plus))
   CPU_ARCHIVE_NAME := ARMv6M
else
ifeq ($(CPU_ARCH),$(filter $(CPU_ARCH), cortex-m3 cortex-m4
↪  cortex-m4f cortex-m7 cortex-m7f))
   CPU_ARCHIVE_NAME := ARMv7M
endif
endif
endif
```

The build target required to use the package as part of a RIOT application is *all*, which is a special target. Special targets are called *phony*. Inside this target, we want to

1. copy the changes made to *KeccakCodePackage* as new files to their right location,
2. build the right target using the *CPU_ARCHIVE_NAME* variable and
3. copy the compiled library file into the RIOT application's build directory *BINDIR*.

This can be done straightforwardly:

```
.PHONY: all

all:
   @cp -r $(RIOTBASE)/pkg/keccak/contrib/Modes $(PKG_BUILDDIR)
   $(MAKE) KeccakHash$(CPU_ARCHIVE_NAME).a -C $(PKG_BUILDDIR)
   @cp $(PKG_BUILDDIR)/bin/KeccakHash$(CPU_ARCHIVE_NAME).a
   ↪  $(BINDIR)/keccak.a
```

If the *PKG_URL* and *PKG_VERSION* variables are set, we can have the external source be automatically downloaded into *PKG_BUILDDIR*. For this we need to include *$(RIOTBASE)/pkg/pkg.mk* (where *$(RIOTBASE)* is RIOT's root directory) in our package's Makefile:

```
include $(RIOTBASE)/pkg/pkg.mk
```

3.2.5. Unit tests

Before the package code can be merged into the master branch of RIOT's repository, it is necessary to prove that the Keccak code works with RIOT by means of a *unit test*. To this end, RIOT uses an extensive externally developed framework called *embUnit* that offers test assertion macros (e.g. *TEST_ASSERT_EQUAL_INT(expected_, actual_)* to assert an integer) and functions to create a test suite.[23] All unit tests are collected under fixtures. [15]. Unit tests can be placed in a subdirectory of *$(RIOTBASE)/tests/unittests* that starts with *tests-* and ends with the name of the module to be tested. A universal Makefile for unit tests exists in *$(RIOTBASE)/tests/unittests*, which makes only the following line necessary for the unit test's Makefile:

```
include $(RIOTBASE)/Makefile.base
```

Additionally, a *Makefile.include* file is necessary inside which the required package is added to the *USEPKG* variable:

```
USEPKG += keccak
```

Then the unit test can be built in *$(RIOTBASE)/tests/unittests* through its build target. The build target has the same name as the subdirectory where the unit test is placed.

Unit tests consist of a test header file and an implementation, where only one function is compulsory:

void tests_<module>(void), serving as an entry point for the test suite,

where *<module>* is the name of the module to be tested. Optional functions for setup and teardown can also be implemented as seen fit.

Our unit test needs to assert that calls to Keccak's hashing functions, i.e.

1. *keccak800hash_initialize()* and *keccak1600hash_initialize()* to initialize the Keccak sponge function instance,
2. *keccak800hash_update()* and *keccak1600hash_update()* to give the instance input data to absorb and
3. *keccak800hash_final()* and *keccak1600hash_final()* to get output bits after all input blocks have been read

work as expected, so we assert that each function's output corresponds to the expected value and finally assert that the acquired digest of a hard-coded input is equal to the expected hash string. We do this for three different input strings of lengths 0, 10 and 155. We can use one of the static inline functions with pre-configured input parameters

[23] http://embunit.sourceforge.net/

instead of *keccak800hash_initialize()* and *keccak1600hash_initialize()*.

A unit test for the Keccak package was written in the context of this thesis. Since there already exists a suite of unit tests for hashing functions in RIOT in the *tests/unittests/tests-hashes* directory, the unit test implementation for Keccak was placed there and the declaration of the unit test's entry point function added to the existing header in that directory.

4. Experimental Comparative Evaluation of Keccak-Based Hashing

4.1. Methodology

4.1.1. Executive summary

We will measure the following:

1. Performance, stack usage and flash memory footprint of SHA-256.
2. Performance, stack usage and flash memory footprint of Keccak-800 and Keccak-1600 for capacities 256, 512 (security levels of 128 and 256 bits, respectively) and more for additional data.

In all cases performance will be measured as the number of ticks needed for one hashing operation. All measurements will be repeated for inputs of 64, 100, 1024 and 10240 bytes on three different ARMv6M and ARMv7M boards (see section 4.2).

For Keccak-800 and Keccak-1600, different implementations will be tested to determine how they compare regarding the things we want to measure.

4.1.2. Discussion

We are interested in measuring how Keccak compares to SHA-2 regarding performance and memory usage. Considering performance, there are a number of choices to be made when designing a benchmark, the first being what exactly is to be measured. In our case, we want to know how long it takes for a hashing operation to complete. Since we're dealing with single-core MCUs with no multi-threading capabilities, a hashing operation should always take the same amount of CPU cycles for a fixed set of parameters ($\pm$ 1 or 2 ticks). The time needed to complete the hashing operation is measured by getting the current system time just before beginning the hashing operation and just after completing it, and calculating the difference of those values.

The next choice to be made is which metrics to normalize the benchmarks around to achieve greater comparability, which parameters to choose accordingly and how to set them. SHA-2 has only the digest length as a modifiable parameter while Keccak also has the rate, the capacity and the delimited suffix (which has no effect on performance,

so 0x06 for SHA3-*'s padding rule can always be used, see section 2.2.3). We do not consider the state size a modifiable parameter for the purposes of this benchmark and stick to the 800 and 1600 bit state sizes provided by the *keccak* package for RIOT.

One possible metric is the digest length so that the performance of both hashing algorithms can be measured for comparable outputs. The digest length is itself a parameter that can be tuned for both SHA-2 and Keccak so we will choose 256 bits as SHA-256 is the only implementation of SHA-2 that RIOT already provides. Since the digest length parameter on Keccak is independent of all other parameters we can set it to 256 bits for all the benchmarks.

Another possible metric is the security level. While SHA-256 provides a security level of 128 bits[24], the security level of Keccak depends on the capacity. A security level of 128 bits is achieved through a capacity of 256 bits (see section 2.1.2).

Nota bene: Although less comparable, a benchmark with a capacity of 512 bits will be run additionally, as it corresponds to the landmark security level of 256 bits and can provide us with additional information for the evaluation of Keccak's performance. For Keccak-1600 this configuration corresponds to SHA3-256 as detailed in table 1.

Another metric could be how often the algorithm's state undergoes an iteration. In Keccak, the input is read into the state in blocks of size r (the rate), which means that the higher r, the less often this (and the squeezing steps) is done. For SHA-256, the input is read into the state in 512-bit blocks; this step is repeated $m_1 = \left\lfloor \frac{n}{512} \right\rfloor + 1$ times where n is the size of the input in bits. The number of absorbing and squeezing steps of Keccak (i.e. the total number of iteration steps during which the state is transformed) is $m_2 = \left\lfloor \frac{n}{r} \right\rfloor + 1 + \left\lceil \frac{256}{r} \right\rceil$ because we have $\left\lfloor \frac{n}{r} \right\rfloor + 1$ absorption steps and $\left\lceil \frac{256}{r} \right\rceil$ squeezing steps for a digest length of 256 bits (and r in bits). Consequently, it may make sense to choose r such that $m_1 = m_2$, i.e.

$$\begin{aligned} & m_1 && = m_2 \\ \iff & \left\lfloor \tfrac{n}{512} \right\rfloor + 1 && = \left\lfloor \tfrac{n}{r} \right\rfloor + 1 + \left\lceil \tfrac{256}{r} \right\rceil \\ \implies & \left\lceil \tfrac{nr}{512} \right\rceil && \approx n + 256 \\ \implies & r && \approx \left\lceil 512 + \tfrac{2^{17}}{n} \right\rceil \text{ bits.} \end{aligned} \tag{7}$$

This way, we may get better insight into roughly how long each iteration takes for SHA-256 compared to Keccak. Basically, this approaches attempts to focus the benchmark on the performance of the Merkle-Damgård construction vs the sponge construction with the Keccak sponge permutation function.

Nota bene: An additional alternative idea for a metric was developed but turned out to be unsuitable for the benchmark. It is shortly described in appendix B for reference. However, benchmarks with rate/capacity values based on an experimental equation not described in this thesis were also run to gather data for additional rate/capacity values beyond the ones described here.

[24] [16], page 1

An important choice to be made pertains to the input sizes, which should cover a range from small to large. If the input size is too small, notable performance differences become less likely and the benchmark becomes uninformative. On the other hand, if the input size becomes too large, the runtime increases disproportionately to any information we may derive from it but not from smaller input sizes. There also remains to consider that we are constrained in how much memory can be allocated for the input due to the hardware. At the same time, what is considered small or large depends on the application and the input sizes for the benchmark should reflect real-world applications of the hashing algorithms. It is reasonable to assume that data to be hashed on IoT devices is the payload of a radio packet (of which the size would be about 64 bytes to around 100 bytes) or a firmware image (about 10 KB to 100 KB of size), for which the integrity of the sent data would need to be verified. Although on devices deployed at a large scale (which is the most likely real-world scenario) a firmware image would be received via radio and consequently be broken into parts of radio packet size, it could be interesting to evaluate if Keccak may be better suited to hash larger inputs less often (so several or all parts of the firmware image at a time) instead of smaller inputs more often. In the context of this thesis, input sizes of 64, 100, 1024 and 10240 bytes were chosen for their applicability to real-world scenarios and their covering of a sufficiently broad range of sizes. An input size of 100 KB was omitted because it's too big for many IoT devices and MCUs, which this thesis focuses on.

Recall from section 3.2.3 the build targets for the Keccak library to use in RIOT:

```
<target name="KeccakHashARMv6M.a" inherits="Keccak800
↪  optimized800ARMv6Mu2 Keccak1600 optimized1600ARMv6Mu2"/>
<target name="KeccakHashARMv7M.a" inherits="Keccak800
↪  optimized800ARMv7Mu2 Keccak1600 inplace1600ARMv7M"/>
```

There exist a number of choices for the inherited fragments, each offering a different balance of performance and memory usage. We will run the benchmarks that implement all the conclusions above once for each of the possible low-level Keccak implementations (names and descriptions taken directly from *LowLevel.build* in *KeccakCodePackage*):

> *optimized800u2*: a generically optimized implementation for 32-bit platforms with 2 rounds unrolled
>
> *optimized800ufull*: same as optimized800u2 but with all rounds unrolled
>
> *optimized800ARMv6Mu1*: an assembly-optimized implementation for ARMv6M (no round unrolling)
>
> *optimized800ARMv6Mu2*: same as optimized800ARMv6Mu1 but with 2 rounds unrolled
>
> *optimized800ARMv7Mu2*: an assembly-optimized implementation for ARMv7M with 2 rounds unrolled

compact1600: an implementation aimed at minimizing code and memory sizes

inplace1600ARMv7M: same as inplace1600bi, but specifically for ARMv7M

optimized1600ARMv6Mu1: an assembly-optimized implementation for ARMv6M (no round unrolling)

optimized1600ARMv6Mu2: same as optimized1600ARMv6Mu1 but with 2 rounds unrolled

Nota bene: The implementations *compact800*, *optimized800ARMv7Mufull*, *inplace1600bi* and *inplace1600ARMv6M* could not be used as they caused various compilation errors due to RIOT's highly error and warning sensitive build flags causing building to fail even when only warnings are produced.

Finally, for each benchmark we will measure the differential stack usage by running the hashing in a separate thread of which the free stack space is measured right before starting and right after terminating, and taking the difference. We will also measure the differential flash usage of each hashing function by comparing the ELF binary executable's text+data value of a dummy version of the application that lacks the hashing function[25] with the actual version of the application that does include the hashing function.

Benchmark applications implementing all the conclusions of this section were placed inside the *tests/hashing* directory of the author's fork of the RIOT repository.[26] It was not part of the pull request to RIOT's master branch.

4.2. Results

The benchmark results can be found in appendix C. The different benchmarks of Keccak are referred to as follows:

Keccak A: Benchmark with c=256 for an equivalent security level to SHA-256 of 128 bits

Keccak B: Benchmark with c=512 for the landmark security level of 256 bits (for Keccak-1600 this is the SHA3-256 configuration, marked in red)

Keccak C: Benchmark normalized around the number of iterations (rounded to a multiple of eight due to Keccak's implementation)

Keccak D: Benchmark with roughly half the rate of *Keccak C* for additional data (rounded to a multiple of eight due to Keccak's implementation)

[25] This means that instead of starting a new thread that calls the hashing function's respective *initialize*, *update* and *finalize* functions, a new empty thread is started that immediately returns NULL.
[26] https://github.com/adrianghc/RIOT/tree/master/tests/hashing

Keccak E: Experimental benchmark with additional capacity/rate values

The following data is presented:

The input size,

the number of ticks (one tick takes one microsecond) for one hash operation,

the ratio of ticks per hashed bytes,

the stack usage in bytes,

the capacity (where applicable) and

the rate (where applicable).

Data was gathered for the following hardware:

STMicroelectronics *STM32F3Discovery* with an ARM Cortex-M4 (ARMv7M) CPU, 256 KB flash memory and 48 KB RAM[27]

Atmel SAM R21 Xplained Pro (*ATSAMR21G18A*) with an ARM Cortex-M0+ (ARMv6M) CPU, 256 KB flash memory and 32 KB RAM[28]

STMicroelectronics *STM32F0Discovery* with an ARM Cortex-M0 (ARMv6M) CPU, 64 KB flash memory and 8 KB RAM[29]

This hardware was chosen because it offers a variety of RAM sizes and three different CPUs running on both ARMv6M and ARMv7M featuring different levels of performance.

The differential flash usages are as shown in table 3. They are independent of the specific boards and depend only on the CPU platform.

4.2.1. Notes

1. For the *STM32F0Discovery* with its small RAM, stack sizes had to be adjusted on the benchmark applications for them to fit on the device and run properly, so the stack sizes that were used (4392 bytes for the main thread, 1400 bytes for the hashing thread) differ from the values inside the author's project repository.
2. Due to *STM32F0Discovery*'s limited RAM, strings of 10240 bytes of length could not be hashed on this device.
3. For unknown reasons, hashing with the *optimized800u2* and *optimized800ufull* implementations of Keccak-800 causes hard faults on ARMv6M boards when using certain values for the rate and capacity. Allocating a larger stack does not fix these hard faults and a fix has not been found as of February 19, 2018. The benchmarks *Keccak C*, *Keccak D* and *Keccak E* were omitted for these implementations on both ARMv6M boards used.

[27] http://www.st.com/en/evaluation-tools/stm32f3discovery.html
[28] http://www.atmel.com/devices/ATSAMR21G18A.aspx
[29] http://www.st.com/en/evaluation-tools/stm32f0discovery.html

	ARMv6M	ARMv7M
SHA-256	1028	1008
optimized800u2	2472	2924
optimized800ufull	14508	19376
optimized800ARMv6Mu1	4500	-
optimized800ARMv6Mu2	5036	-
optimized800ARMv7Mu2	-	5000
compact1600	1788	1692
inplace1600ARMv7M	-	11548
optimized1600ARMv6Mu1	8384	-
optimized1600ARMv6Mu2	9552	-

Table 3: Differential flash usages of hashing algorithms and implementations

4.2.2. Comparison to x86 performance

To have a point of reference regarding how the boards perform compared to a normal x86 PC, an additional benchmark was run on a virtual machine on a PC with the following configuration:

Intel Core i5-7300U CPU,

virtual machine running on *Oracle VM VirtualBox*,

guest OS *Ubuntu 16.04.3 LTS* and

host OS *Windows 10 1709*.

On a PC, performance data is much more volatile with the methodology used because of its multi-threading capabilities and the significantly greater number of running threads/processes, so the collected values should only be considered for comparative purposes to the values collected on the IoT boards and not as as a full-fledged benchmark for all intents and purposes.

The data can be found in table 7. One can see that performance on a PC is more than an entire order of magnitude better than even on the fastest board tested for 64 byte input (e.g. 19 ticks for one SHA-256 hash vs 277 ticks on *STM32F3Discovery*) and between two and three orders of magnitude better (closer to three) for larger input sizes (e.g. 294 ticks for one SHA-256 hash vs 17933 ticks on *STM32F3Discovery* for 10240 bytes input). It can also be found that Keccak performs faster on a PC compared to SHA-256 than on the boards (e.g. for 102040 bytes input: 127 ticks for *optimized800u2* with c=256 vs 294 for SHA-256 on the PC and 34074 ticks for *optimized800u2* with c=256 vs 42449 for SHA-256 on *ATSAMR21G18A*). A detailed analysis of how Keccak's performance compares to SHA-256 follows in section 4.3.2.

4.2.3. Comparison to RIOT's new SHA-3 implementation

Two weeks after the completion of this manuscript's original version on February 19, 2018, a pull request[30] was accepted for porting SHA-3 into RIOT not as a package, but as a sys module. This implementation of SHA-3 and Keccak-1600 is based on the standalone *CompactFIPS202* implementation of Keccak[31] that is designed for compactness and readability. It includes functions for all the Keccak instances approved in the FIPS 202 standard and allows for custom input parameters to Keccak-1600 as well. Additional measurements were conducted for this implementation and can be found in table 8 of appendix C. They show considerably worse performance than all other implementations tested across all the hardware, underlining the significance of hardware-dependent optimizations for performance. Stack usage was slightly higher than on the other implementations on x86 but lower than on all the other implementations except *inplace1600ARMv7M* on all tested ARM boards.

4.3. Analysis of the measurements

4.3.1. Executive summary

A number of observations can be derived from the data. They will be discussed in full detail in section 4.3.2, but a number of key insights will be presented in this section.

Recall that Keccak-1600 with a capacity of 512 bits corresponds to the SHA3-256 configuration (see table 1).

1. Asymptotically, a higher capacity leads to longer runtime of Keccak. This correlation appears to be very roughly exponential (with flatter growth in some ranges) and independent of input size (see figure 7).

2. For both 128 and 256 bits of security, all Keccak implementations but one of Keccak-1600 perform consistently better than SHA-256 or on comparable levels that are never significantly worse. Performance is particularly good for Keccak-1600, which on all but one implementation outperforms SHA-256 even for 256 bits of security (i.e. SHA3-256). While this is not the case for Keccak-800, its performance for 256 bits of security is still much better than if there was a linear correlation to the security level (i.e. the number of ticks per hash is significantly lower than double that of SHA-256, which has half the security level). Figures 8 and 9 show the best-performing implementations on ARMv6M and ARMv7M compared to SHA-256 at both security levels.

3. Both Keccak-800 and Keccak-1600 consistently require a smaller stack than SHA-256 regardless of the specific implementation, input size and capacity. However, this is offset in the case of Keccak-1600 for some implementations when also taking the state size into account, which is 200 bytes compared to 32 bytes for SHA-256.

[30] https://github.com/RIOT-OS/RIOT/pull/7881

[31] https://github.com/gvanas/KeccakCodePackage/blob/master/Standalone/CompactFIPS202/C/Keccak-readable-and-compact.c

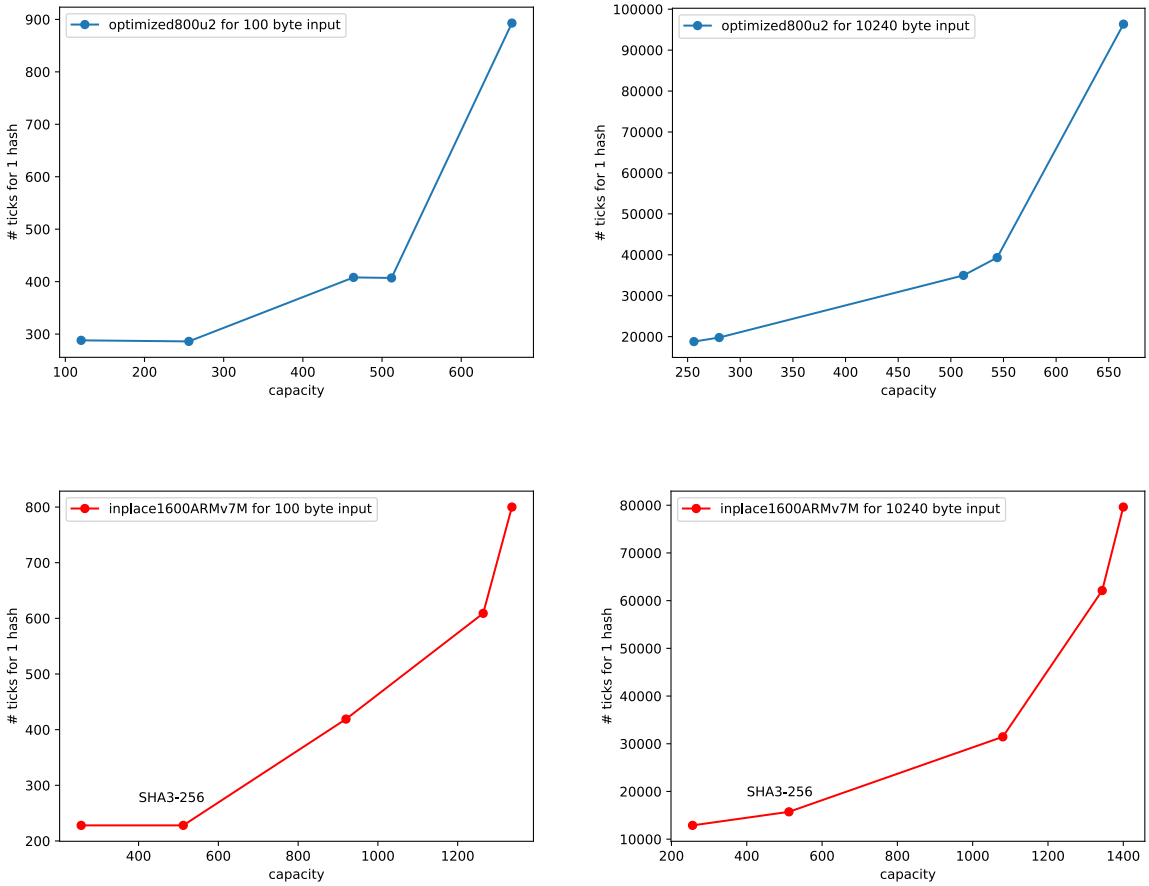

Figure 7: Correlation of runtime and capacity for Keccak-800 and Keccak-1600 (*STM32F3Discovery*)

For Keccak-800, memory usage is always better than SHA-256 even when taking the state size into account (100 bytes for Keccak-800). Examples are presented in figure 10.

From this, it follows that if RAM is a concern and ignoring other factors, a configuration of Keccak-800 appears to be the best choice out of the three hashing algorithms.

4. However, as can be seen in table 3, the flash memory footprint of both Keccak-800 and Keccak-1600 is higher than that of SHA-256 for all implementations - on average, 7.55 times as high for Keccak-800 and 4.64 times as high for Keccak-1600.

The trade-off of Keccak's higher performance with a high flash memory footprint can

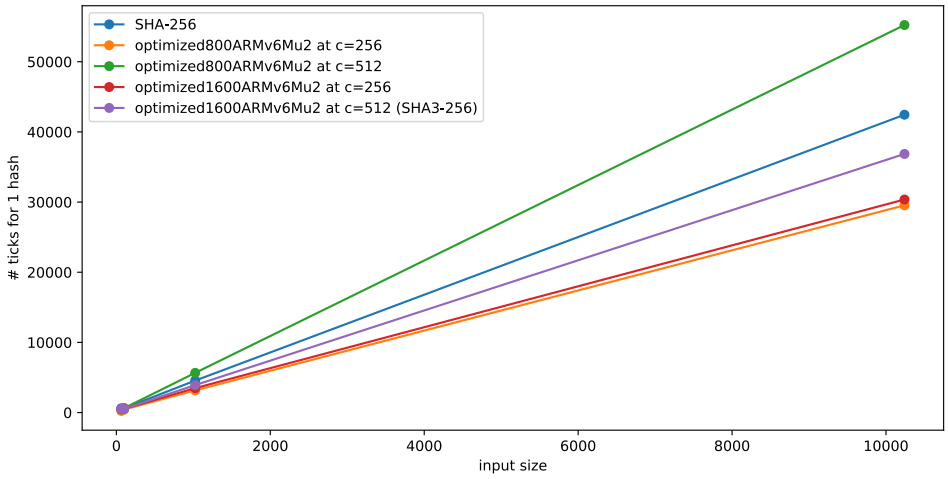

Figure 8: Performance of Keccak-800 and Keccak-1600 vs SHA-256 on ARMv6M (*ATSAMR21G18A*)

be solved by using high-performing Keccak-1600 implementations when higher performance is desirable (even at the price of a higher memory footprint) while using slower-performing Keccak-800 implementations with a lower flash memory footprint if it can be accepted that performance is sometimes lower than SHA-256's in favor of a lower flash memory footprint.

While SHA3-256 was the only configuration from the SHA-3 standard that was tested due to the 32 byte digest length (see table 1), one can extrapolate from the measured increase of runtime with growing capacity and conclude that SHA3-256 could be considered the best choice out of the SHA-3 standard for its balance of a high security level (256 bits) and performance above SHA-256.

4.3.2. Discussion

Looking at the capacity values, it can be observed that the greater the capacity, the longer the benchmark takes. From this it follows that the high capacity values of *Keccak D* and *E* (growing to numbers greater than 512, which corresponds to 256 bits of security) lead to performance values that are above what would be used for real-world scenarios as the big capacity leads to performance decreases for only marginal gain, as 256 bit security can already be achieved with a smaller capacity.

The *compact1600* implementation consistently performs significantly worse than both SHA-256 and all other Keccak implementations. The difference in performance is so vast (e.g. 140736 vs 42449 ticks for SHA-256 for 10240 byte input and $c = 256$ on the *ATSAMR21G18A* board) that the benefit of reduced code and memory sizes of this

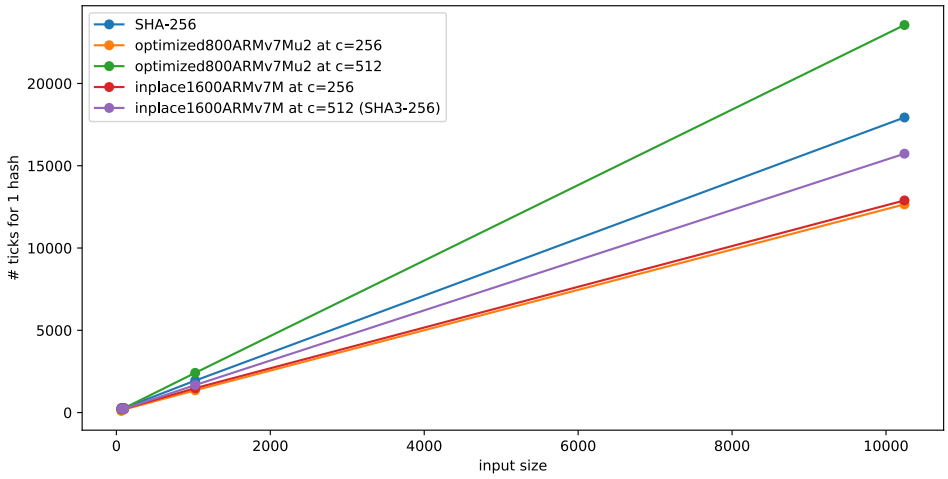

Figure 9: Performance of Keccak-800 and Keccak-1600 vs SHA-256 on ARMv7M (*STM32F3Discovery*)

implementation is not worth the performance tradeoff unless the memory limitations are very severe.

This said, despite being intended as an implementation that minimizes code and memory size, its stack usage is actually considerably greater than *inplace1600ARMv7M*'s on the ARMv7M board *STM32F3Discovery* (84 vs 240 bytes for a 1024 byte input) and while it's smaller than those of the other Keccak-1600 implementations on ARMv6M, they still do require a smaller stack than SHA-256 does so the benefit of *compact1600*'s low stack usage becomes even more questionable.

A similar observation also holds for all Keccak-800 implementations, whose stack usage is consistently smaller than that of SHA-256.

Another observation to be drawn from the benchmarks is that Keccak-800 and Keccak-1600 consistently perform better than SHA-256 for the same security level of 128 bits (i.e. a capacity of 256 bits) across all ARMv6M implementations except *compact1600*. This advantage is far less pronounced in ARMv7M, where only *optimized800ARVMv7Mu2* (for Keccak-800) and *inplace1600ARMv7M* (for Keccak-1600) show consistently better results and the others show slightly worse but comparable results. In any case, however, the stack usage is always smaller than for SHA-256, as already observed.

However, it should be taken into account that in addition to the stack size of the algorithm, the state takes up its share of memory as well - 200 bytes for Keccak-1600 and 100 bytes for Keccak-800 but only 32 bytes for SHA-256. This does make the memory advantage of Keccak less pronounced but it does not eliminate it for any Keccak-800 implementation.

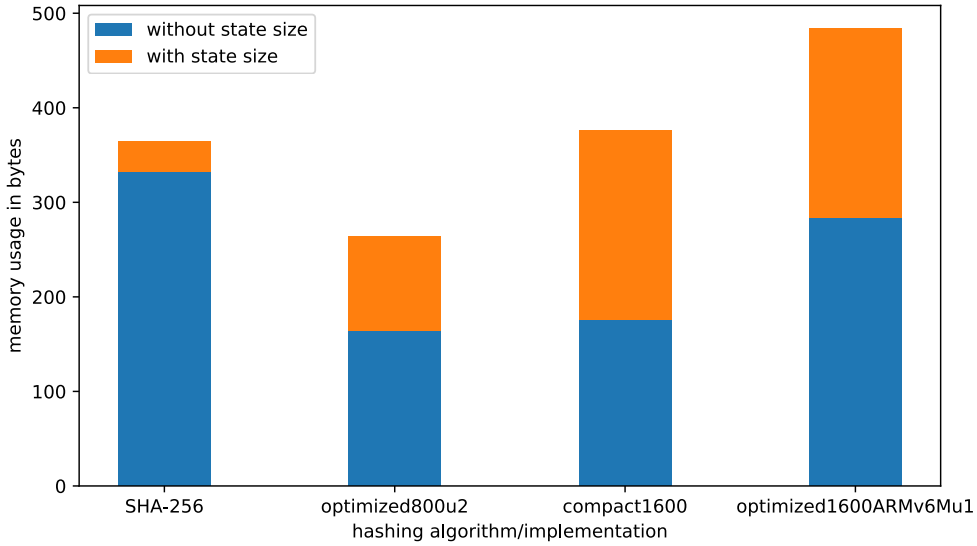

Figure 10: Memory usage of Keccak-800 and Keccak-1600 vs SHA-256 for 1024 byte input (*ATSAMR21G18A*)

Unfortunately, none of the ARMv6M implementations of Keccak-1600 that were tested have an overall advantage over SHA-256 concerning memory usage when the state size is taken into account - however, on ARMv7M this is still the case for *inplaceARMv7M*.

For a security level of 256 bits, the performance results are promising. On ARMv7M (i.e. on the *STM32F3Discovery* hardware), while no Keccak-800 implementation performs consistently better than SHA-256, *optimized800ARMv7Mu2* performs roughly as good or better than SHA-256 for 64 to 100 byte inputs (185 vs 277 and 267 vs 278 ticks, respectively) and only 1.3 times slower than SHA-256 for a 10240 byte input (23551 vs 17933 ticks), which are good results considering the doubled security level and the roughly half as big stack usage (164 vs 316 bytes for all tested input sizes) that still translates to 84 bytes of memory savings (264 vs 348 bytes) even when adding the respective state sizes.

For Keccak-1600 (i.e. SHA3-256) the *inplace1600ARMv7M* implementation even performs consistently better than SHA-256 (1.14 times faster for a 10240 byte input) with as little as 26.6% of its stack usage (84 vs 316 bytes) and 81.6% of its total memory usage adding the state sizes (284 vs 348 bytes). This performance advantage decreases only very slowly with growing input sizes - e.g. for a 64 byte input *inplace1600ARMv7M* performs 1.24 times faster than SHA-256 and 1.16 times faster for a 1024 byte input, which suggests a merely logarithmic decrease. The results for ARMv6M are on a very comparable scale, with both *optimized1600ARMv6Mu1* and *optimized1600ARMv6Mu2*

outperforming SHA-256 with still only 85.5% of the latter's stack usage for 10240 bytes of input on the *ATSAMR21G18A* board.

However, looking at the differential flash usage in table 3, it becomes clear that the performance benefits of Keccak come at the price of much higher program sizes than SHA-256. For Keccak-800, the flash memory footprint can be as big as 19 times that of SHA-256 as is the case for *optimized800ufull* on ARMv7M. Fortunately, this is not much of an issue as this implementation is neither the best performing nor the one with the smallest stack usage for ARMv6M or ARMv7M. The smallest amount of flash memory is used by *optimized800u2* on both ARMv6M and ARMv7M. It is the worst performing Keccak-800 implementation on both platforms - which sounds worse than it is as it still performs consistently better than SHA-256 for the same security level on ARMv6M and only up to 1.05 times worse on ARMv7M for 128 bits of security (e.g. 18796 vs 17933 ticks for 10240 byte input). Consequently, it is reasonable to accept the slightly worse performance than SHA-256's (in the case of ARMv7M) in exchange for the smallest differential stack usage of all Keccak-800 implementations, as the other implementations (excluding *optimized800ufull*) would require up to roughly five times the flash memory of SHA-256 (e.g. 5000 bytes for *optimized800ARMv7Mu2* vs 1008 bytes for SHA-256). Unfortunately, we must recall that *optimized800u2* caused hard faults for certain capacity values on ARMv6M (see section 4.2.1), not making it advisable to use this implementation on this hardware before the underlying cause can be found and fixed. On ARMv6M, the next smallest amount of flash memory (though still 4.38 times more than SHA-256 with 4500 vs 1028 bytes) is used by *optimized800ARMv6Mu1*.

For Keccak-1600, *compact1600* expectedly requires far less flash memory than any other Keccak implementation tested but we recall that it performed significantly worse than the other Keccak-1600 implementations and SHA-256. The other Keccak-1600 implementations perform very well but, as in the case of *inplace1600ARMv7M*, require up to 6.82 times more flash memory than *compact1600* (11548 vs 1692 bytes) and up to 11.46 times more than SHA-256 (11548 vs 1008 bytes). This is a problem because *inplace1600ARMv7M* and *compact1600* are the only implementations for ARMv7M that could be tested, which means that there is either a very big performance or a very big flash memory trade-off. The numbers are similar for ARMv6M.

A solution could be offered by the following approach: Using Keccak-800 when flash memory is a bigger concern than performance and Keccak-1600 when better performance is advisable even at the price of a higher flash memory footprint. Following from our findings, these fragments were chosen for the build targets of the RIOT package for the pull request:

1. *optimized800u2* for Keccak-800 on ARMv7M as the Keccak implementation with the lowest flash memory footprint but in some cases worse performance than SHA-256.

2. *optimized800ARMv6Mu1* for Keccak-800 on ARMv6M as the Keccak implementation with the lowest flash memory footprint after *optimized800u2*. If a solution

to the problem identified in section 4.2.1 is found, *optimized800u2* can be used instead.

3. *inplace1600ARMv7M* for Keccak-1600 on ARMv7M for the best performance on ARMv7M at the price of the highest flash memory footprint.

4. *optimized1600ARMv6Mu2* for Keccak-1600 on ARMv6M for the best performance on ARMv6M at the price of the highest flash memory footprint.

Recall that the *Keccak C* benchmark uses the rate necessary to have Keccak perform roughly as many iterations as SHA-256. One can see that in the case of Keccak-1600, performance is only better than SHA-256's for the smallest input size tested (64 bytes) and always worse otherwise. On the other hand, all Keccak-800 implementations but *optimized800u2* and *optimized800ufull* perform significantly better than SHA-256, and even said two implementations perform only slightly worse than SHA-256 (e.g. 19783 ticks for *optimized800u2* vs 17933 ticks for SHA-256 for 10240 bytes input on the *STM32F3Discovery* board). However, while the performance difference is vast for small inputs (e.g. 238 ticks for *optimized800ARMv6Mu2* vs 609 ticks for SHA-256 for 64 bytes input on the *ATSAMR21G18A*), it becomes significantly smaller for larger inputs (e.g. 32075 ticks for *optimized800ARMv6Mu2* vs 42449 ticks for SHA-256 for 10240 bytes input on the same board). It follows that Keccak-800 appears to perform iterations faster than or almost as fast as SHA-256 for a broad range of input sizes (when setting the capacity as explained) but at the same time, iteration costs in that case increase faster for Keccak-800 than SHA-256, so it is to be expected that Keccak-800's advantage in speed per iteration disappears asymptotically with larger input sizes. In addition, the rate decreases with increasing input sizes (e.g. 768 bytes for 64 bytes input vs 520 bytes for 10240 bytes input on the *STM32F3Discovery*), increasing the capacity and thus the security level as we maintain the same number of iterations as SHA-256, so keeping the same security level for Keccak requires a smaller number of iterations than SHA-256 for growing input sizes while maintaining performance benefits.

Finally, we want to answer the question whether Keccak is better suited to hash a bigger number of smaller input or a smaller number of larger input. Focusing on the implementations identified earlier as well-performing, it appears that while there is not a linearity to it, the ratio of ticks per hashed bytes does consistently decrease with asymptotically growing input sizes (e.g. for *optimized800ARMv7Mu2* and $c = 256$, the ratio is 1.6094 for 64 bytes of input, 1.86 for 100 bytes, 1.3242 for 1024 bytes and 1.2357 for 10240 bytes). However, asymptotically the decrease appears to be only logarithmic so the approach of hashing larger input at a time offers only marginal performance gains. Nonetheless, there is no apparent drawback as based on the benchmark data, the stack usage remains constant for asymptotically growing input sizes.

These observations hold true for Keccak-800 and Keccak-1600 with different capacity/rate values on both ARMv6M and ARMv7M, although the specific ratios and ratio

decreases vary.

Regarding SHA-3 standards compliance, only SHA3-256 was tested as it's the only SHA-3 configuration in the standard that has a digest length of 32 bytes (not counting SHAKE* for fixed digest lengths). However, as we have found that performance decreases as capacity increases, one can extrapolate that SHA-3 configurations with a higher capacity have worse performance than the tested SHA3-256 (reinforced further when considering that the digest is longer for those configurations as well so the number of iterations increases, see table 1), and SHA-3 configurations with a lower capacity show better performance (improved further considering that the digest is shorter for those configurations). Considering the apparently roughly exponential growth of hashing runtime with increasing capacity (see figure 7) and SHA3-256's excellent security level of 256 bits while at the same time keeping a higher performance than SHA-256, SHA3-256 could be considered as offering the best balance of security and performance from the SHA-3 standard configurations.

It can be concluded from these benchmarks that while Keccak is not universally better than SHA-256, many combinations of parameters and implementations exist for which both Keccak-800 and Keccak-1600 outperform SHA-256 without sacrificing security on low-powered IoT boards for input sizes that are of interest to their use cases. A higher security level can be achieved with only little trade-offs or even memory and performance benefits, the latter of which is especially the case for Keccak-1600 (in the SHA3-256 configuration) in particular. If flash memory is a concern, there are still implementations that perform at similar levels as SHA-256 at the same security level.

5. Conclusions

New technology is rarely without any trade-offs compared to its predecessors so the question is always whether the benefits outweigh the disadvantages. In the case of SHA-3 or Keccak, one can argue in both directions.

An argument in favor of Keccak is that it has proven security levels against the attacks discussed in section 2.1.2 that are higher than those of SHA-256 (for the right choice of capacity) and as found in section 4.3, such a security level can be achieved in a configuration with satisfying memory usage and performance (SHA3-256).

An argument against Keccak is that it is not better than SHA-256 in every regard for any configuration so one can argue that this makes the benefits questionable. Since SHA-256's publication in 2001, it has remained unbroken and while this is no guarantee for the future, it does suggest a certain robustness that SHA-3 has a smaller track record of in practice so far. However, it should be noted that despite its young age, Keccak has seen a remarkable amount of third-party scrutiny and analysis [17]. It also has the advantage of having been developed in the open, i.e. as part of an open call and with an entirely open code and design rationale, which enables more sophisticated research. This is not the case for SHA-256, which was developed behind closed doors at NSA and

standardized without significant public scrutiny [18].

This said, the fact remains that even as SHA-3 was officially announced, NIST considered SHA-256 to be "secure and suitable for general use"[32]. For this reason, SHA-3 was not conceived of as a direct replacement to SHA-256 but rather as an additional choice and a secure backup in case an exploit to SHA-256 is found. It follows that at this point in time, there is no pressing need to replace SHA-256 with SHA-3 in the field.

However, the case of IoT must be particularly considered here. For one, the scenario of a mass deployment must be taken into account where updates are potentially accompanied by significant inconvenience and cost, so this may be an additional factor playing into a decision to opt for SHA-256's proven robustness - or alternatively, for SHA3-256's higher security level, so the argument goes both ways. Keccak lends itself to very fast hardware implementations but so far these do not exist for common IoT hardware platforms, so for now, this benefit remains more theoretical than practical. This said, a more practical benefit of Keccak is its versatility enabling it for more cryptographic uses beyond hashing (and MAC). IoT devices with their memory constraints may thus benefit from being equipped with a Keccak-based cryptographic suite that could be more versatile than a SHA-256-based suite for hashing and HMAC calculation. Whether this versatility is required depends on the specific use case and application.

Ultimately, it turns out that there is no definitive answer to whether SHA-256 or SHA-3 is better suited for IoT devices. While Keccak does provide tangible benefits over SHA-256 in some configurations, the trade-offs (mostly flash usage) are often very noticeable as well. However, if we assume that exploits are equally likely to be found for SHA-256 and Keccak in the future, Keccak has its versatility and higher possible security level to its advantage. Generally, Keccak and SHA3-256 in particular appears to be a good choice on IoT devices and beyond, and possibly the better choice in the long term.

As far as RIOT is concerned, this thesis has shown an example of a package for which the porting process to RIOT was satisfactorily straightforward. RIOT's clear, logical and abstracting structure, good legibility (thanks to its coding and naming conventions) and the integration of the *embUnit* module, which greatly facilitates the implementation of unit tests, contribute to this just as much as the open development culture, the constructive criticism and the responsiveness of the maintainers. Setting up a development environment for RIOT is as simple as cloning a repository and, only when running Windows, setting up a Vagrant box provided in the repository with one single command. Therefore, it appears fitting to conclude that RIOT provides a very good gateway into the world of IoT programming for beginners and experts alike.

5.1. Perspectives

Keccak's high versatility above only hashing is a unique characteristic that warrants additional analysis far beyond what this thesis can provide. A good starting point for future analysis and benchmarking work may be the flexible *Strobe* framework for cryptographic

[32] https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition

protocols that was mentioned at the end of section 2.3. It supports "authenticated DH or FHMQV, signatures, password-authenticated key exchange, DPA-resistant key diversification, ratcheting for forward secrecy, and steganographic connections with length hiding"[33]. A possible area of focus could be an exploration of the advantages in terms of flash and random access memory that a cryptographic protocol framework based entirely on Keccak could provide compared to a framework with discrete implementations for the cryptographic protocols. Since Strobe is still in its infancy, a unique opportunity to contribute to its development could arise from findings out of future analysis work in this direction.

References

[1] Roundup Of Internet Of Things Forecasts And Market Estimates, 2016 - Louis Columbus (https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/)

[2] The Internet of Things (IoT) units installed base by category from 2014 to 2020 (in billions) (https://www.statista.com/statistics/370350/internet-of-things-installed-base-by-category/)

[3] Operating Systems for Low-End Devices in the Internet of Things: a Survey - Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, Nicolas Tsiftes (https://hal.inria.fr/hal-01245551/document)

[4] Securing the Internet of Things: A Proposed Framework (http://www.cisco.com/c/en/us/about/security-center/secure-iot-proposed-framework.html)

[5] NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition (https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition)

[6] Cryptographic sponge functions - Guido Bertoni, Joan Daemen, Gilles Van Assche, Michaël Peeters, [January 14, 2011] (http://sponge.noekeon.org/CSF-0.1.pdf)

[7] The sponge and duplex constructions (https://keccak.team/sponge_duplex.html)

[8] Keccak and the SHA-3 Standardization - Guido Bertoni, Joan Daemen, Gilles Van Assche, Michaël Peeters [February 6, 2013] (http://csrc.nist.gov/groups/ST/hash/sha-3/documents/Keccak-slides-at-NIST.pdf)

[9] The Keccak reference - Guido Bertoni, Joan Daemen, Gilles Van Assche, Michaël Peeters [January 14, 2011] (http://keccak.noekeon.org/Keccak-reference-3.0.pdf)

[33] https://strobe.sourceforge.io/

[10] SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions - Morris J. Dworkin [August 4, 2015] (https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions?pub_id=919061)

[11] Terminology for Constrained-Node Networks - Internet Engineering Task Force (IETF) (http://www.ietf.org/rfc/rfc7228.txt)

[12] RIOT - The friendly Operating System for the Internet of Things (https://riot-os.org)

[13] https://github.com/RIOT-OS/RIOT/wiki/Introduction

[14] RIOT Documentation (https://riot-os.org/api/index.html)

[15] https://github.com/RIOT-OS/RIOT/blob/master/tests/unittests/README.md

[16] Descriptions of SHA-256, SHA-384, and SHA-512 (https://web.archive.org/web/20130526224224/↩ http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf)

[17] Third-party cryptanalysis (https://keccak.team/third_party.html)

[18] Keccak: open-source cryptography [September 26, 2017] (https://keccak.team/2017/open_source_crypto.html)

A. Notation and Conventions

1. For a set X, $|X|$ denotes the cardinality of X. For a string X, $|X|$ denotes the length of the string.
2. $X \circ Y$ denotes the concatenation of two strings X and Y.
3. $\mathbb{Z}_2^*$ denotes the set of all bit strings (i.e. 0 or 1) of any length.

B. An Unused Benchmarking Idea

Recall from section 4.1.2 that we wanted to find metrics to normalize the benchmarks around. One possible metric was the number of iterations the state undergoes but an alternative idea was that instead of trying to achieve the same number of iterations we may want to achieve the same total number of state bits transformed over the entirety of all iterations. For SHA-256, the state has 256 bits, so for each iteration, up to 256 bits would be transformed (so $q_1 = 256 \cdot m_1$ overall for a number of $m_1 = \left\lfloor \frac{n}{512} \right\rfloor + 1$ iterations). For Keccak-p with a state size of p bits the number would be $q_2 = p \cdot m_2$ with $m_2 = \left\lfloor \frac{n}{r} \right\rfloor + 1 + \left\lceil \frac{256}{r} \right\rceil$ as before, so for a fixed input size of n, we would want to choose r such that $q_1 = q_2$, i.e.

$$\begin{array}{lll} & q_1 & = q_2 \\ \iff & 256 \cdot (\left\lfloor \frac{n}{512} \right\rfloor + 1) & = p \cdot (\left\lfloor \frac{n}{r} \right\rfloor + 1 + \left\lceil \frac{256}{r} \right\rceil) \\ \implies & rn + 512r & \approx 2p \cdot (n + r + 256) \\ \implies & r & \approx \left\lceil \frac{2pn+512p}{n-2p+512} \right\rceil \text{ bits.} \end{array} \tag{8}$$

However, one can see that it is not possible for r to reach values between 0 and 800 (or 1600) for positive values of n. It also appears that the left side of the equation is always smaller than the right for those values of r and positive n. Consequently, it appears that Keccak-800 and Keccak-1600 always transform a larger number of bits over all iterations than SHA-256 does.

C. Benchmark Results

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
SHA-256	64	277	4.3281	316	-	-
Keccak-800 A (optimized800u2)	64	164	2.5625	152	256	544
Keccak-800 A (optimized800ufull)	64	159	2.4844	156	256	544
Keccak-800 A (optimized800ARMv7Mu2)	64	103	1.6094	140	256	544
Keccak-800 B (optimized800u2)	64	284	4.4375	176	512	288
Keccak-800 B (optimized800ufull)	64	275	4.2969	180	512	288
Keccak-800 B (optimized800ARMv7Mu2)	64	185	2.8906	164	512	288
Keccak-800 C (optimized800u2)	64	163	2.5469	152	32	768
Keccak-800 C (optimized800ufull)	64	159	2.4844	156	32	768
Keccak-800 C (optimized800ARMv7Mu2)	64	103	1.6094	140	32	768
Keccak-800 D (optimized800u2)	64	285	4.4531	176	416	384
Keccak-800 D (optimized800ufull)	64	275	4.2969	180	416	384
Keccak-800 D (optimized800ARMv7Mu2)	64	185	2.8906	164	416	384
Keccak-800 E (optimized800u2)	64	650	10.1563	176	648	152
Keccak-800 E (optimized800ufull)	64	626	9.7813	180	648	152
Keccak-800 E (optimized800ARMv7Mu2)	64	430	6.7188	164	648	152
Keccak-1600 A (compact1600)	64	1336	20.8750	180	256	1344
Keccak-1600 A (inplace1600ARMv7M)	64	223	3.4844	60	256	1344
Keccak-1600 B (compact1600)	64	1336	20.8750	180	512	1088
Keccak-1600 B (inplace1600ARMv7M)	64	223	3.4844	60	512	1088
Keccak-1600 C (compact1600)	64	1352	21.1250	180	832	768

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 C (inplace1600ARMv7M)	64	223	3.4844	60	832	768
Keccak-1600 D (compact1600)	64	2604	40.6875	204	1216	384
Keccak-1600 D (inplace1600ARMv7M)	64	412	6.4375	84	1216	384
Keccak-1600 E (compact1600)	64	2604	40.6875	204	1296	304
Keccak-1600 E (inplace1600ARMv7M)	64	415	6.4844	84	1296	304
SHA-256	100	278	2.7800	316	-	-
Keccak-800 A (optimized800u2)	100	286	2.8600	176	256	544
Keccak-800 A (optimized800ufull)	100	277	2.7700	180	256	544
Keccak-800 A (optimized800ARMv7Mu2)	100	186	1.8600	164	256	544
Keccak-800 B (optimized800u2)	100	407	4.0700	176	512	288
Keccak-800 B (optimized800ufull)	100	393	3.9300	180	512	288
Keccak-800 B (optimized800ARMv7Mu2)	100	267	2.6700	164	512	288
Keccak-800 C (optimized800u2)	100	288	2.8800	176	120	680
Keccak-800 C (optimized800ufull)	100	278	2.7800	180	120	680
Keccak-800 C (optimized800ARMv7Mu2)	100	187	1.8700	164	120	680
Keccak-800 D (optimized800u2)	100	408	4.0800	176	464	336
Keccak-800 D (optimized800ufull)	100	394	3.9400	180	464	336
Keccak-800 D (optimized800ARMv7Mu2)	100	267	2.6700	164	464	336
Keccak-800 E (optimized800u2)	100	893	8.9300	176	664	136
Keccak-800 E (optimized800ufull)	100	860	8.6000	180	664	136
Keccak-800 E (optimized800ARMv7Mu2)	100	593	5.9300	164	664	136
Keccak-1600 A (compact1600)	100	1342	13.4200	180	256	1344

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 A (inplace1600ARMv7M)	100	228	2.2800	60	256	1344
Keccak-1600 B (compact1600)	100	1342	13.4200	180	512	1088
Keccak-1600 B (inplace1600ARMv7M)	100	228	2.2800	60	512	1088
Keccak-1600 C (compact1600)	100	2651	26.5100	204	920	680
Keccak-1600 C (inplace1600ARMv7M)	100	419	4.1900	84	920	680
Keccak-1600 D (compact1600)	100	3879	38.7900	204	1264	336
Keccak-1600 D (inplace1600ARMv7M)	100	609	6.0900	84	1264	336
Keccak-1600 E (compact1600)	100	5149	51.4900	204	1336	264
Keccak-1600 E (inplace1600ARMv7M)	100	800	8.0000	84	1336	264
SHA-256	1024	1943	1.8975	316	-	-
Keccak-800 A (optimized800u2)	1024	2024	1.9766	176	256	544
Keccak-800 A (optimized800ufull)	1024	1949	1.9033	180	256	544
Keccak-800 A (optimized800ARMv7Mu2)	1024	1356	1.3242	164	256	544
Keccak-800 B (optimized800u2)	1024	3593	3.5088	176	512	288
Keccak-800 B (optimized800ufull)	1024	3457	3.3760	180	512	288
Keccak-800 B (optimized800ARMv7Mu2)	1024	2414	2.3574	164	512	288
Keccak-800 C (optimized800u2)	1024	2038	1.9902	176	272	528
Keccak-800 C (optimized800ufull)	1024	1959	1.9131	180	272	528
Keccak-800 C (optimized800ARMv7Mu2)	1024	1361	1.3291	164	272	528
Keccak-800 D (optimized800u2)	1024	3975	3.8818	176	536	264
Keccak-800 D (optimized800ufull)	1024	3825	3.7354	180	536	264
Keccak-800 D (optimized800ARMv7Mu2)	1024	2661	2.5986	164	536	264

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 E (optimized800u2)	1024	9912	9.6797	176	696	104
Keccak-800 E (optimized800ufull)	1024	9533	9.3096	180	696	104
Keccak-800 E (optimized800ARMv7Mu2)	1024	6647	6.4912	164	696	104
Keccak-1600 A (compact1600)	1024	9133	8.9189	204	256	1344
Keccak-1600 A (inplace1600ARMv7M)	1024	1483	1.4482	84	256	1344
Keccak-1600 B (compact1600)	1024	10402	10.1582	204	512	1088
Keccak-1600 B (inplace1600ARMv7M)	1024	1672	1.6328	84	512	1088
Keccak-1600 C (compact1600)	1024	20893	20.4033	204	1072	528
Keccak-1600 C (inplace1600ARMv7M)	1024	3204	3.1289	84	1072	528
Keccak-1600 D (compact1600)	1024	40845	39.8877	204	1336	264
Keccak-1600 D (inplace1600ARMv7M)	1024	6252	6.1055	84	1336	264
Keccak-1600 E (compact1600)	1024	52259	51.0342	204	1392	208
Keccak-1600 E (inplace1600ARMv7M)	1024	7965	7.7783	84	1392	208
SHA-256	10240	17933	1.7513	316	-	-
Keccak-800 A (optimized800u2)	10240	18796	1.8355	176	256	544
Keccak-800 A (optimized800ufull)	10240	18091	1.7667	180	256	544
Keccak-800 A (optimized800ARMv7Mu2)	10240	12654	1.2357	164	256	544
Keccak-800 B (optimized800u2)	10240	34967	3.4147	176	512	288
Keccak-800 B (optimized800ufull)	10240	33637	3.2849	180	512	288
Keccak-800 B (optimized800ARMv7Mu2)	10240	23551	2.2999	164	512	288
Keccak-800 C (optimized800u2)	10240	19783	1.9319	176	280	520
Keccak-800 C (optimized800ufull)	10240	18998	1.8553	180	280	520

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 C (optimized800ARMv7Mu2)	10240	13268	1.2957	164	280	520
Keccak-800 D (optimized800u2)	10240	39311	3.8390	176	544	256
Keccak-800 D (optimized800ufull)	10240	37812	3.6926	180	544	256
Keccak-800 D (optimized800ARMv7Mu2)	10240	26477	2.5856	164	544	256
Keccak-800 E (optimized800u2)	10240	96349	9.4091	176	696	104
Keccak-800 E (optimized800ufull)	10240	92651	9.0479	180	696	104
Keccak-800 E (optimized800ARMv7Mu2)	10240	64662	6.3146	164	696	104
Keccak-1600 A (compact1600)	10240	79421	7.7560	204	256	1344
Keccak-1600 A (inplace1600ARMv7M)	10240	12892	1.2590	84	256	1344
Keccak-1600 B (compact1600)	10240	98448	9.6141	204	512	1088
Keccak-1600 B (inplace1600ARMv7M)	10240	15732	1.5363	84	512	1088
Keccak-1600 C (compact1600)	10240	205902	20.1076	204	1080	520
Keccak-1600 C (inplace1600ARMv7M)	10240	31454	3.0717	84	1080	520
Keccak-1600 D (compact1600)	10240	409216	39.9625	204	1344	256
Keccak-1600 D (inplace1600ARMv7M)	10240	62124	6.0668	84	1344	256
Keccak-1600 E (compact1600)	10240	523378	51.1111	204	1400	200
Keccak-1600 E (inplace1600ARMv7M)	10240	79620	7.7754	84	1400	200

Table 4: A benchmark sample on an *STM32F3Discovery* board

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
SHA-256	64	609	9.5156	332	-	-
Keccak-800 A (optimized800u2)	64	287	4.4844	140	256	544
Keccak-800 A (optimized800ufull)	64	281	4.3906	144	256	544
Keccak-800 A (optimized800ARMv6Mu1)	64	246	3.8438	132	256	544
Keccak-800 A (optimized800ARMv6Mu2)	64	238	3.7188	132	256	544
Keccak-800 B (optimized800u2)	64	509	7.9531	164	512	288
Keccak-800 B (optimized800ufull)	64	498	7.7813	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	64	447	6.9844	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	64	430	6.7188	156	512	288
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	64	246	3.8438	132	32	768
Keccak-800 C (optimized800ARMv6Mu2)	64	238	3.7188	132	32	768
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	64	447	6.9844	156	416	384
Keccak-800 D (optimized800ARMv6Mu2)	64	430	6.7188	156	416	384
Keccak-800 E (optimized800u2)	-	-	-	-	-	-
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	64	1058	16.5313	156	648	152
Keccak-800 E (optimized800ARMv6Mu2)	64	1015	15.8594	156	648	152
Keccak-1600 A (compact1600)	64	2362	36.9063	144	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	64	531	8.2969	252	256	1344

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 A (optimized1600ARMv6Mu2)	64	517	8.0781	252	256	1344
Keccak-1600 B (compact1600)	64	2362	36.9063	144	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	64	531	8.2969	252	512	1088
Keccak-1600 B (optimized1600ARMv6Mu2)	64	517	8.0781	252	512	1088
Keccak-1600 C (compact1600)	64	2367	36.9844	144	832	768
Keccak-1600 C (optimized1600ARMv6Mu1)	64	531	8.2969	252	832	768
Keccak-1600 C (optimized1600ARMv6Mu2)	64	517	8.0781	252	832	768
Keccak-1600 D (compact1600)	64	4630	72.3438	176	1216	384
Keccak-1600 D (optimized1600ARMv6Mu1)	64	978	15.2813	284	1216	384
Keccak-1600 D (optimized1600ARMv6Mu2)	64	951	14.8594	284	1216	384
Keccak-1600 E (compact1600)	64	4630	72.3438	176	1296	304
Keccak-1600 E (optimized1600ARMv6Mu1)	64	985	15.3906	284	1296	304
Keccak-1600 E (optimized1600ARMv6Mu2)	64	957	14.9531	284	1296	304
SHA-256	100	609	6.0900	332	-	-
Keccak-800 A (optimized800u2)	100	511	5.1100	164	256	544
Keccak-800 A (optimized800ufull)	100	499	4.9900	168	256	544
Keccak-800 A (optimized800ARMv6Mu1)	100	449	4.4900	156	256	544
Keccak-800 A (optimized800ARMv6Mu2)	100	432	4.3200	156	256	544
Keccak-800 B (optimized800u2)	100	733	7.3300	164	512	288
Keccak-800 B (optimized800ufull)	100	715	7.1500	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	100	649	6.4900	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	100	624	6.2400	156	512	288

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	100	451	4.5100	156	120	680
Keccak-800 C (optimized800ARMv6Mu2)	100	434	4.3400	156	120	680
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	100	656	6.5600	156	464	336
Keccak-800 D (optimized800ARMv6Mu2)	100	630	6.3000	156	464	336
Keccak-800 E (optimized800u2)	-	-	-	-	-	-
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	100	1464	14.6400	156	664	136
Keccak-800 E (optimized800ARMv6Mu2)	100	1403	14.0300	156	664	136
Keccak-1600 A (compact1600)	100	2370	23.7000	144	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	100	547	5.4700	252	256	1344
Keccak-1600 A (optimized1600ARMv6Mu2)	100	534	5.3400	252	256	1344
Keccak-1600 B (compact1600)	100	2370	23.7000	144	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	100	547	5.4700	252	512	1088
Keccak-1600 B (optimized1600ARMv6Mu2)	100	534	5.3400	252	512	1088
Keccak-1600 C (compact1600)	100	4647	46.4700	176	920	680
Keccak-1600 C (optimized1600ARMv6Mu1)	100	997	9.9700	284	920	680
Keccak-1600 C (optimized1600ARMv6Mu2)	100	969	9.6900	284	920	680
Keccak-1600 D (compact1600)	100	6906	69.0600	176	1264	336
Keccak-1600 D (optimized1600ARMv6Mu1)	100	1447	14.4700	284	1264	336

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 D (optimized1600ARMv6Mu2)	100	1406	14.0600	284	1264	336
Keccak-1600 E (compact1600)	100	9173	91.7300	176	1336	264
Keccak-1600 E (optimized1600ARMv6Mu1)	100	1902	19.0200	284	1336	264
Keccak-1600 E (optimized1600ARMv6Mu2)	100	1847	18.4700	284	1336	264
SHA-256	1024	4556	4.4492	332	-	-
Keccak-800 A (optimized800u2)	1024	3663	3.5771	164	256	544
Keccak-800 A (optimized800ufull)	1024	3568	3.4844	168	256	544
Keccak-800 A (optimized800ARMv6Mu1)	1024	3303	3.2256	156	256	544
Keccak-800 A (optimized800ARMv6Mu2)	1024	3165	3.0908	156	256	544
Keccak-800 B (optimized800u2)	1024	6546	6.3926	164	512	288
Keccak-800 B (optimized800ufull)	1024	6372	6.2227	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	1024	5908	5.7695	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	1024	5658	5.5254	156	512	288
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	1024	3383	3.3037	156	272	528
Keccak-800 C (optimized800ARMv6Mu2)	1024	3244	3.1680	156	272	528
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	1024	6629	6.4736	156	536	264
Keccak-800 D (optimized800ARMv6Mu2)	1024	6352	6.2031	156	536	264
Keccak-800 E (optimized800u2)	-	-	-	-	-	-

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	1024	16453	16.0674	156	696	104
Keccak-800 E (optimized800ARMv6Mu2)	1024	15753	15.3838	156	696	104
Keccak-1600 A (compact1600)	1024	16187	15.8076	176	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	1024	3576	3.4922	284	256	1344
Keccak-1600 A (optimized1600ARMv6Mu2)	1024	3480	3.3984	284	256	1344
Keccak-1600 B (compact1600)	1024	18454	18.0215	176	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	1024	4023	3.9287	284	512	1088
Keccak-1600 B (optimized1600ARMv6Mu2)	1024	3913	3.8213	284	512	1088
Keccak-1600 C (compact1600)	1024	36665	35.8057	176	1072	528
Keccak-1600 C (optimized1600ARMv6Mu1)	1024	7676	7.4961	284	1072	528
Keccak-1600 C (optimized1600ARMv6Mu2)	1024	7456	7.2813	284	1072	528
Keccak-1600 D (compact1600)	1024	72872	71.1641	176	1336	264
Keccak-1600 D (optimized1600ARMv6Mu1)	1024	14893	14.5439	284	1336	264
Keccak-1600 D (optimized1600ARMv6Mu2)	1024	14454	14.1152	284	1336	264
Keccak-1600 E (compact1600)	1024	93280	91.0938	176	1392	208
Keccak-1600 E (optimized1600ARMv6Mu1)	1024	18916	18.4727	284	1392	208
Keccak-1600 E (optimized1600ARMv6Mu2)	1024	18353	17.9229	284	1392	208
SHA-256	10240	42449	4.1454	332	-	-
Keccak-800 A (optimized800u2)	10240	34074	3.3275	164	256	544
Keccak-800 A (optimized800ufull)	10240	33171	3.2394	168	256	544
Keccak-800 A (optimized800ARMv6Mu1)	10240	30839	3.0116	156	256	544

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 A (optimized800ARMv6Mu2)	10240	29534	2.8842	156	256	544
Keccak-800 B (optimized800u2)	10240	63783	6.2288	164	512	288
Keccak-800 B (optimized800ufull)	10240	62079	6.0624	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	10240	57695	5.6343	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	10240	55231	5.3937	156	512	288
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	10240	33441	3.2657	156	280	520
Keccak-800 C (optimized800ARMv6Mu2)	10240	32075	3.1323	156	280	520
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	10240	64909	6.3388	156	544	256
Keccak-800 D (optimized800ARMv6Mu2)	10240	62133	6.0677	156	544	256
Keccak-800 E (optimized800u2)	-	-	-	-	-	-
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	10240	160119	15.6366	156	696	104
Keccak-800 E (optimized800ARMv6Mu2)	10240	153288	14.9695	156	696	104
Keccak-1600 A (compact1600)	10240	140736	13.7438	176	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	10240	31200	3.0469	284	256	1344
Keccak-1600 A (optimized1600ARMv6Mu2)	10240	30363	2.9651	284	256	1344
Keccak-1600 B (compact1600)	10240	174746	17.0650	176	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	10240	37900	3.7012	284	512	1088

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 B (optimized1600ARMv6Mu2)	10240	36857	3.5993	284	512	1088
Keccak-1600 C (compact1600)	10240	361387	35.2917	176	1080	520
Keccak-1600 C (optimized1600ARMv6Mu1)	10240	75530	7.3760	284	1080	520
Keccak-1600 C (optimized1600ARMv6Mu2)	10240	73361	7.1642	284	1080	520
Keccak-1600 D (compact1600)	10240	730246	71.3131	176	1344	256
Keccak-1600 D (optimized1600ARMv6Mu1)	10240	147331	14.3878	284	1344	256
Keccak-1600 D (optimized1600ARMv6Mu2)	10240	142924	13.9574	284	1344	256
Keccak-1600 E (compact1600)	10240	934328	91.2430	176	1400	200
Keccak-1600 E (optimized1600ARMv6Mu1)	10240	189318	18.4881	284	1400	200
Keccak-1600 E (optimized1600ARMv6Mu2)	10240	183676	17.9371	284	1400	200

Table 5: A benchmark sample on an *ATSAMR21G18A* board

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
SHA-256	64	544	8.5000	332	-	-
Keccak-800 A (optimized800u2)	64	248	3.8750	140	256	544
Keccak-800 A (optimized800ufull)	64	242	3.7813	144	256	544
Keccak-800 A (optimized800ARMv6Mu1)	64	203	3.1718	132	256	544
Keccak-800 A (optimized800ARMv6Mu2)	64	196	3.0625	132	256	544
Keccak-800 B (optimized800u2)	64	432	6.7500	164	512	288
Keccak-800 B (optimized800ufull)	64	422	6.5938	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	64	372	5.8125	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	64	357	5.5781	156	512	288
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	64	204	3.1875	132	32	768
Keccak-800 C (optimized800ARMv6Mu2)	64	196	3.0625	156	32	768
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	64	373	5.8281	156	416	384
Keccak-800 D (optimized800ARMv6Mu2)	64	357	5.5781	156	416	384
Keccak-800 E (optimized800u2)	-	-	-	-	-	-
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	64	889	13.8906	156	648	152
Keccak-800 E (optimized800ARMv6Mu2)	64	851	13.2969	156	648	152
Keccak-1600 A (compact1600)	64	2231	34.8594	144	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	64	446	6.9688	252	256	1344

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 A (optimized1600ARMv6Mu2)	64	435	6.7969	252	256	1344
Keccak-1600 B (compact1600)	64	2231	34.8594	144	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	64	446	6.9688	252	512	1088
Keccak-1600 B (optimized1600ARMv6Mu2)	64	434	6.7813	252	512	1088
Keccak-1600 C (compact1600)	64	2231	34.8594	144	832	768
Keccak-1600 C (optimized1600ARMv6Mu1)	64	446	6.9688	252	832	768
Keccak-1600 C (optimized1600ARMv6Mu2)	64	435	6.7969	252	832	768
Keccak-1600 D (compact1600)	64	4364	68.1875	176	1216	384
Keccak-1600 D (optimized1600ARMv6Mu1)	64	822	12.8438	284	1216	384
Keccak-1600 D (optimized1600ARMv6Mu2)	64	800	12.5000	284	1216	384
Keccak-1600 E (compact1600)	64	4363	68.1719	176	1296	304
Keccak-1600 E (optimized1600ARMv6Mu1)	64	828	12.9375	284	1296	304
Keccak-1600 D (optimized1600ARMv6Mu2)	64	806	12.5938	284	1296	304
SHA-256	100	543	5.4300	332	-	-
Keccak-800 A (optimized800u2)	100	435	4.3500	164	256	544
Keccak-800 A (optimized800ufull)	100	425	4.2500	168	256	544
Keccak-800 A (optimized800ARMv6Mu1)	100	374	3.7400	156	256	544
Keccak-800 A (optimized800ARMv6Mu2)	100	359	3.5900	156	256	544
Keccak-800 B (optimized800u2)	100	621	6.2100	164	512	288
Keccak-800 B (optimized800ufull)	100	606	6.0600	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	100	543	5.4300	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	100	520	5.2000	156	512	288

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	100	378	3.7800	156	120	680
Keccak-800 C (optimized800ARMv6Mu2)	100	362	3.6200	156	120	680
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	100	552	5.5200	156	464	336
Keccak-800 D (optimized800ARMv6Mu2)	100	528	5.2800	156	464	336
Keccak-800 E (optimized800u2)	-	-	-	-	-	-
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	100	1231	12.3100	156	664	136
Keccak-800 E (optimized800ARMv6Mu2)	100	1177	11.7700	156	664	136
Keccak-1600 A (compact1600)	100	2242	22.4200	144	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	100	460	4.6000	252	256	1344
Keccak-1600 A (optimized1600ARMv6Mu2)	100	449	4.4900	252	256	1344
Keccak-1600 B (compact1600)	100	2241	22.4100	144	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	100	461	4.6100	252	512	1088
Keccak-1600 B (optimized1600ARMv6Mu2)	100	450	4.5000	252	512	1088
Keccak-1600 C (compact1600)	100	4373	43.7300	176	920	680
Keccak-1600 C (optimized1600ARMv6Mu1)	100	840	8.4000	284	920	680
Keccak-1600 C (optimized1600ARMv6Mu2)	100	818	8.1800	284	920	680
Keccak-1600 D (compact1600)	100	6506	65.0600	176	1264	336
Keccak-1600 D (optimized1600ARMv6Mu1)	100	1219	12.1900	284	1264	336

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 D (optimized1600ARMv6Mu2)	100	1185	11.8500	284	1264	336
Keccak-1600 E (compact1600)	100	8639	86.3900	176	1336	264
Keccak-1600 E (optimized1600ARMv6Mu1)	100	1602	16.0200	284	1336	264
Keccak-1600 E (optimized1600ARMv6Mu2)	100	1558	15.5800	284	1336	264
SHA-256	1024	4002	3.9082	332	-	-
Keccak-800 A (optimized800u2)	1024	3090	3.0176	164	256	544
Keccak-800 A (optimized800ufull)	1024	3009	2.9385	168	256	544
Keccak-800 A (optimized800ARMv6Mu1)	1024	2787	2.7217	156	256	544
Keccak-800 A (optimized800ARMv6Mu2)	1024	2663	2.6006	156	256	544
Keccak-800 B (optimized800u2)	1024	5501	5.3721	164	512	288
Keccak-800 B (optimized800ufull)	1024	5354	5.2285	168	512	288
Keccak-800 B (optimized800ARMv6Mu1)	1024	4974	4.8574	156	512	288
Keccak-800 B (optimized800ARMv6Mu2)	1024	4750	4.6387	156	512	288
Keccak-800 C (optimized800u2)	-	-	-	-	-	-
Keccak-800 C (optimized800ufull)	-	-	-	-	-	-
Keccak-800 C (optimized800ARMv6Mu1)	1024	2893	2.8252	156	272	528
Keccak-800 C (optimized800ARMv6Mu2)	1024	2769	2.7041	156	272	528
Keccak-800 D (optimized800u2)	-	-	-	-	-	-
Keccak-800 D (optimized800ufull)	-	-	-	-	-	-
Keccak-800 D (optimized800ARMv6Mu1)	1024	5637	5.5049	156	536	264
Keccak-800 D (optimized800ARMv6Mu2)	1024	5390	5.2637	156	536	264
Keccak-800 E (optimized800u2)	-	-	-	-	-	-

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 E (optimized800ufull)	-	-	-	-	-	-
Keccak-800 E (optimized800ARMv6Mu1)	1024	13884	13.5586	156	696	104
Keccak-800 E (optimized800ARMv6Mu2)	1024	13257	12.9463	156	696	104
Keccak-1600 A (compact1600)	1024	15305	14.9463	176	256	1344
Keccak-1600 A (optimized1600ARMv6Mu1)	1024	3401	3.3213	284	256	1344
Keccak-1600 A (optimized1600ARMv6Mu2)	1024	2963	2.8936	284	256	1344
Keccak-1600 B (compact1600)	1024	17436	17.0273	176	512	1088
Keccak-1600 B (optimized1600ARMv6Mu1)	1024	3417	3.3369	284	512	1088
Keccak-1600 B (optimized1600ARMv6Mu2)	1024	3329	3.2510	284	256	1344
Keccak-1600 C (compact1600)	1024	34492	33.6836	176	1072	528
Keccak-1600 C (optimized1600ARMv6Mu1)	1024	6499	6.3467	284	1072	528
Keccak-1600 C (optimized1600ARMv6Mu2)	1024	6322	6.1738	284	1072	528
Keccak-1600 D (compact1600)	1024	68605	66.9971	176	1336	264
Keccak-1600 D (optimized1600ARMv6Mu1)	1024	12581	12.2861	284	1336	264
Keccak-1600 D (optimized1600ARMv6Mu2)	1024	12228	11.9414	284	1336	264
Keccak-1600 E (compact1600)	1024	87793	85.7354	176	1392	208
Keccak-1600 E (optimized1600ARMv6Mu1)	1024	15972	15.5977	284	1392	208
Keccak-1600 E (optimized1600ARMv6Mu2)	1024	15520	15.1563	284	1392	208

Table 6: A benchmark sample on an *STM32F0Discovery* board

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
SHA-256	64	19	0.2969	744	-	-
Keccak-800 A (optimized800u2)	64	13	0.2031	352	256	544
Keccak-800 B (optimized800u2)	64	8	0.1250	384	512	288
Keccak-800 C (optimized800u2)	64	8	0.1250	352	32	768
Keccak-800 D (optimized800u2)	64	8	0.1250	384	416	384
Keccak-800 E (optimized800u2)	64	14	0.2188	792	648	152
Keccak-1600 A (compact1600)	64	13	0.2031	356	256	1344
Keccak-1600 B (compact1600)	64	9	0.1406	356	512	1088
Keccak-1600 C (compact1600)	64	8	0.1250	356	832	768
Keccak-1600 D (compact1600)	64	20	0.3125	388	1216	384
Keccak-1600 E (compact1600)	64	15	0.2344	388	1296	304
SHA-256	100	13	0.1300	744	-	-
Keccak-800 A (optimized800u2)	100	7	0.0700	384	256	544
Keccak-800 B (optimized800u2)	100	9	0.0900	384	512	288
Keccak-800 C (optimized800u2)	100	7	0.0700	384	120	680
Keccak-800 D (optimized800u2)	100	8	0.0800	384	464	336
Keccak-800 E (optimized800u2)	100	13	0.1300	384	664	136
Keccak-1600 A (compact1600)	100	12	0.1200	356	256	1344
Keccak-1600 B (compact1600)	100	13	0.1300	356	512	1088
Keccak-1600 C (compact1600)	100	13	0.1300	388	920	680
Keccak-1600 D (compact1600)	100	17	0.1700	388	1264	336
Keccak-1600 E (compact1600)	100	21	0.2100	388	1336	264
SHA-256	1024	89	0.0869	744	-	-

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-800 A (optimized800u2)	1024	19	0.0186	384	256	544
Keccak-800 B (optimized800u2)	1024	24	0.0234	384	512	288
Keccak-800 C (optimized800u2)	1024	20	0.0195	384	272	528
Keccak-800 D (optimized800u2)	1024	29	0.0283	384	536	264
Keccak-800 E (optimized800u2)	1024	74	0.0723	384	696	104
Keccak-1600 A (compact1600)	1024	35	0.0342	388	256	1344
Keccak-1600 B (compact1600)	1024	40	0.0391	388	512	1088
Keccak-1600 C (compact1600)	1024	76	0.0742	388	1072	528
Keccak-1600 D (compact1600)	1024	149	0.1455	388	1336	264
Keccak-1600 E (compact1600)	1024	388	0.3789	388	1392	208
SHA-256	10240	294	0.0287	744	-	-
Keccak-800 A (optimized800u2)	10240	127	0.0124	384	256	544
Keccak-800 B (optimized800u2)	10240	211	0.0206	384	512	288
Keccak-800 C (optimized800u2)	10240	135	0.0132	384	280	520
Keccak-800 D (optimized800u2)	10240	237	0.0231	384	544	256
Keccak-800 E (optimized800u2)	10240	607	0.0593	384	696	104
Keccak-1600 A (compact1600)	10240	295	0.0288	388	256	1344
Keccak-1600 B (compact1600)	10240	521	0.0509	388	512	1088
Keccak-1600 C (compact1600)	10240	744	0.0727	388	1080	520
Keccak-1600 D (compact1600)	10240	2768	0.2703	388	1344	256
Keccak-1600 E (compact1600)	10240	3334	0.3256	388	1400	200

Table 7: A benchmark sample on an x86 PC for comparative purposes (Intel Core i5-7300U, *Oracle VM VirtualBox*, guest OS *Ubuntu 16.04.3 LTS*, host OS *Windows 10 1709*)

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 A (*STM32F3Discovery*)	64	8353	130.5156	104	256	1344
Keccak-1600 B (*STM32F3Discovery*)	64	8353	130.5156	104	512	1088
Keccak-1600 C (*STM32F3Discovery*)	64	8353	130.5156	104	832	768
Keccak-1600 D (*STM32F3Discovery*)	64	16639	259.9844	104	1216	384
Keccak-1600 E (*STM32F3Discovery*)	64	16639	259.9844	104	1296	304
Keccak-1600 A (*ATSAMR21G18A*)	64	12322	192.5313	104	256	1344
Keccak-1600 B (*ATSAMR21G18A*)	64	12322	192.5313	104	512	1088
Keccak-1600 C (*ATSAMR21G18A*)	64	12322	192.5313	104	832	768
Keccak-1600 D (*ATSAMR21G18A*)	64	24533	383.3281	104	1216	384
Keccak-1600 E (*ATSAMR21G18A*)	64	24533	383.3281	104	1296	304
Keccak-1600 A (*STM32F0Discovery*)	64	13796	215.5625	104	256	1344
Keccak-1600 B (*STM32F0Discovery*)	64	13797	215.5781	104	512	1088
Keccak-1600 C (*STM32F0Discovery*)	64	13797	215.5781	104	832	768
Keccak-1600 D (*STM32F0Discovery*)	64	27479	429.3594	104	1216	384
Keccak-1600 E (*STM32F0Discovery*)	64	27479	429.3594	104	1296	304
Keccak-1600 A (PC)	64	36	0.5625	416	256	1344
Keccak-1600 B (PC)	64	31	0.4844	416	512	1088
Keccak-1600 C (PC)	64	30	0.4688	416	832	768
Keccak-1600 D (PC)	64	129	2.0156	432	1216	384
Keccak-1600 E (PC)	64	53	0.8281	432	1296	304
Keccak-1600 A (*STM32F3Discovery*)	100	8364	83.6400	104	256	1344
Keccak-1600 B (*STM32F3Discovery*)	100	8364	83.6400	104	512	1088
Keccak-1600 C (*STM32F3Discovery*)	100	16649	166.4900	104	920	680
Keccak-1600 D (*STM32F3Discovery*)	100	24936	249.3600	104	1264	336
Keccak-1600 E (*STM32F3Discovery*)	100	33222	332.2200	104	1336	264
Keccak-1600 A (*ATSAMR21G18A*)	100	12346	123.4600	104	256	1344

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 B (*ATSAMR21G18A*)	100	12346	123.4600	104	512	1088
Keccak-1600 C (*ATSAMR21G18A*)	100	24557	245.5700	104	920	680
Keccak-1600 D (*ATSAMR21G18A*)	100	36767	367.6700	104	1264	336
Keccak-1600 E (*ATSAMR21G18A*)	100	48978	489.7800	104	1336	264
Keccak-1600 A (*STM32F0Discovery*)	100	13822	138.2200	104	256	1344
Keccak-1600 B (*STM32F0Discovery*)	100	13822	138.2200	104	512	1088
Keccak-1600 C (*STM32F0Discovery*)	100	27504	275.0400	104	920	680
Keccak-1600 D (*STM32F0Discovery*)	100	41185	411.8500	104	1264	336
Keccak-1600 E (*STM32F0Discovery*)	100	54868	548.6800	104	1336	264
Keccak-1600 A (PC)	100	30	0.3000	416	256	1344
Keccak-1600 B (PC)	100	27	0.2700	416	512	1088
Keccak-1600 C (PC)	100	44	0.4400	432	920	680
Keccak-1600 D (PC)	100	75	0.7500	432	1264	336
Keccak-1600 E (PC)	100	93	0.9300	432	1336	264
Keccak-1600 A (*STM32F3Discovery*)	1024	58350	56.9824	104	256	1344
Keccak-1600 B (*STM32F3Discovery*)	1024	66637	65.0752	104	512	1088
Keccak-1600 C (*STM32F3Discovery*)	1024	132927	129.8115	104	1072	528
Keccak-1600 D (*STM32F3Discovery*)	1024	265507	259.2842	104	1336	264
Keccak-1600 E (*STM32F3Discovery*)	1024	340084	332.1133	104	1392	208
Keccak-1600 A (*ATSAMR21G18A*)	1024	86225	84.2041	104	256	1344
Keccak-1600 B (*ATSAMR21G18A*)	1024	98435	96.1279	104	512	1088
Keccak-1600 C (*ATSAMR21G18A*)	1024	196118	191.5215	104	1072	528
Keccak-1600 D (*ATSAMR21G18A*)	1024	391484	382.3086	104	1336	264
Keccak-1600 E (*ATSAMR21G18A*)	1024	501378	489.6270	104	1392	208
Keccak-1600 A (*STM32F0Discovery*)	1024	96568	94.3047	104	256	1344
Keccak-1600 B (*STM32F0Discovery*)	1024	110250	107.6660	104	512	1088

Benchmark	input size	#ticks for 1 hash	#ticks per hashed bytes	stack usage	capacity	rate
Keccak-1600 C (*STM32F0Discovery*)	1024	219706	214.5566	104	1072	528
Keccak-1600 D (*STM32F0Discovery*)	1024	438619	428.3389	104	1336	264
Keccak-1600 E (*STM32F0Discovery*)	1024	561757	548.5908	104	1392	208
Keccak-1600 A (PC)	1024	145	0.1416	432	256	1344
Keccak-1600 B (PC)	1024	161	0.1572	432	512	1088
Keccak-1600 C (PC)	1024	326	0.3184	432	1072	528
Keccak-1600 D (PC)	1024	1391	1.3584	432	1336	264
Keccak-1600 E (PC)	1024	1107	1.0811	432	1392	208
Keccak-1600 A (*STM32F3Discovery*)	10240	508496	49.6578	104	256	1344
Keccak-1600 B (*STM32F3Discovery*)	10240	632791	61.7960	104	512	1088
Keccak-1600 C (*STM32F3Discovery*)	10240	1312265	128.1509	104	1080	520
Keccak-1600 D (*STM32F3Discovery*)	10240	2662925	260.0513	104	1344	256
Keccak-1600 E (*STM32F3Discovery*)	10240	3408689	332.8798	104	1400	200
Keccak-1600 A (*ATSAMR21G18A*)	10240	751729	73.4110	104	256	1344
Keccak-1600 B (*ATSAMR21G18A*)	10240	934884	91.2973	104	512	1088
Keccak-1600 C (*ATSAMR21G18A*)	10240	1936135	189.0757	104	1080	520
Keccak-1600 D (*ATSAMR21G18A*)	10240	3926425	383.4399	104	1344	256
Keccak-1600 E (*ATSAMR21G18A*)	10240	5025360	490.7578	104	1400	200
Keccak-1600 A (PC)	10240	1083	0.1058	432	256	1344
Keccak-1600 B (PC)	10240	1332	0.1301	432	512	1088
Keccak-1600 C (PC)	10240	3377	0.3298	432	1080	520
Keccak-1600 D (PC)	10240	8696	0.8492	432	1344	256
Keccak-1600 E (PC)	10240	10539	1.0292	432	1400	200

Table 8: Additional benchmarks of RIOT's newly included Keccak-1600 implementation on the previous hardware