The KECCAK SHA-3 submission

Guido BERTONI[1]
Joan DAEMEN[1]
Michaël PEETERS[2]
Gilles VAN ASSCHE[1]

`http://keccak.noekeon.org/`

Version **3**
January 14, 2011

[1]STMicroelectronics
[2]NXP Semiconductors

1 Defining KECCAK

KECCAK is defined in [12]. It is a family of sponge functions with members KECCAK$[r, c]$ characterized by two parameters:

- bitrate r,
- capacity c.

The sum $r + c$ determines the width of the KECCAK-f permutation used in the sponge construction and is restricted to values in $\{25, 50, 100, 200, 400, 800, 1600\}$.

The sponge construction uses $r + c$ bits of state, of which r are updated with message bits between each application of KECCAK-f during the absorbing phase and output during the squeezing phase. The remaining c bits are not directly affected by message bits, nor are they taken as output.

KECCAK allows one to choose its security parameter c independently from the output length. We express our security claim for KECCAK in [12, Section "Security claim…"] as a *flat sponge claim* [6]. This type of claim implies that the expected success probability of any attack should be not higher than that for a random oracle plus the so-called $\mathcal{RO}$ differentiating advantage $N^2/2^{c_{\text{claim}}+1}$. The value c_{claim} is called the *claimed capacity* and fully determines the claimed security level of the variable-output-length function.

The design philosophy underlying KECCAK is the *hermetic sponge strategy*: adopting the sponge construction using a permutation that should not have structural distinguishers [6]. In this approach, we can make a flat sponge claim with claimed capacity c_{claim} equal to the parameter c in the construction and trade in claimed security level for speed by increasing c and decreasing r accordingly.

2 Proposals for the SHA-3 standard

In [20], NIST requires the candidate algorithms to support at least four different output lengths $n \in \{224, 256, 384, 512\}$ with associated security levels. Hence, we define the following four fixed-output-length variants (where $\lfloor\rfloor_n$ indicates truncation to the first n bits):

- $n = 224$: $\lfloor$KECCAK$[r = 1152, c = 448]\rfloor_{224}$
- $n = 256$: $\lfloor$KECCAK$[r = 1088, c = 512]\rfloor_{256}$
- $n = 384$: $\lfloor$KECCAK$[r = 832, c = 768]\rfloor_{384}$
- $n = 512$: $\lfloor$KECCAK$[r = 576, c = 1024]\rfloor_{512}$

In addition, we propose KECCAK[] (with default parameters), where the user may truncate the output at the desired output length.

There are other valid parameter choices and other possible ways of using the KECCAK family, which we now discuss.

2.1 Letting the user choose the output length

In many use cases of hash functions the output length is determined by the application. This is the case for key derivation functions and several important public key signature and key establishment schemes, for instance the widely used RSA padding schemes [16, 17]. In those cases, either the output must be truncated or an additional construction called mask generating function (MGF) must be applied to provide longer outputs [16, 17].

Consider a protocol to be designed with the requirement of a specific digest length ℓ. When using a hash function family that consists of a set of instances with different output lengths and ℓ is not among them, one must first choose an instance and either truncate or specify an MGF construction. When using a variable-output-length hash function, no such choice must be made and it suffices to truncate the output to the desired length. The advantage of a variable-output-length hash function becomes even more important if a protocol or application requires digests whose length is a parameter of the protocol.

2.1.1 What about the security level?

Traditionally, hash function users expect a security level that matches its output length: $2^{n/2}$ for collision-resistance and 2^n for (second) pre-image resistance. As a consequence of the flat sponge claim, a variable-output-length hash function with a claimed capacity c_{claim} shall resist to any attack with complexity below $2^{c_{\text{claim}}/2}$, but nothing is claimed above this level. Hence, the value $2^{c_{\text{claim}}/2}$ acts as a ceiling for the security level.

This ceiling poses no problem if high enough. For instance, the ceiling is at 2^{288} in the case of KECCAK[], as it has capacity $c = 576$. Consider an application where we need a 512-bit output. Traditionally, a (second) pre-image resistance level of 2^{512} would be expected, while for KECCAK[] with output truncated to 512 bits a security level of *only* 2^{288} is claimed. However, the difference between these two security levels is purely philosophical with no practical implications whatsoever. By translating these computation complexities into physical quantities such as time or energy, both are simply out of reach and will remain so in the foreseeable future [4].

2.1.2 What about diversification?

A single function for all output lengths may pose problems when a scheme requires that different output lengths are generated with different hash function instances. Diversification is actually a requirement that may arise for other aspects than different output lengths. A scheme or protocol may require different hash function instances even if their output lengths are the same. Diversification can be established at very small cost using a well-established technique called *domain separation*. Domain separation is an efficient means to construct different function instances from a single underlying function. If the underlying function is secure, the derived functions can be considered as independent functions.

One can implement domain separation by appending or prepending different constants to the input for each of the function instances: $f_i(M) = \text{KECCAK}(M||C_i)$ or $f_i(M) = \text{KECCAK}(C_i||M)$. As a concrete example, one can use a convention based on namespaces such as described in [6, Section "Domain separation"].

2.2 Letting the user choose the capacity

For standardization, one option is to impose a small set of (or just a single instance of) parameter values. Another option is to allow the user to freely choose them. We consider in particular the case where a user can freely choose the capacity of KECCAK with $r = 1600 - c$ so that the width of KECCAK-f is fixed. In this section, we describe the advantages and disadvantages of this option.

As explained in [6], the hermetic sponge strategy allows the user to trade in speed for claimed security, or vice versa, by choosing the capacity. Relative performance estimates for various (r, c) pairs are listed in Table 1.

r	c	Relative performance
576	1024	$\div 1.778$
832	768	$\div 1.231$
1024	576	1
1088	512	$\times 1.063$
1152	448	$\times 1.125$
1216	384	$\times 1.188$
1280	320	$\times 1.250$
1344	256	$\times 1.312$
1408	192	$\times 1.375$

Table 1: Relative performance of Keccak$[r, c]$ with respect to Keccak$[]$.

If the user decides to lower the capacity to $c = 256$, providing a claimed security level equivalent to that of AES-128, the performance will be 31% greater than for the default value $c = 576$. If the user wants an output truncated to 512 bits to provide the traditionally expected (second) pre-image resistance of 2^{512} by setting the capacity to $c = 1024$, she can do this at the cost of a performance decreased by 78%.

A variable capacity can also result in important efficiency gain in applications dealing with (mostly) short messages. Consider for example an application with messages that are exactly 1024 bits long. The padding will extend these messages by 2 bits resulting in a two-block message and hence applying Keccak$[]$ results in two calls to Keccak-f. If we decrease the capacity by 2 bits to 574 (still providing an astronomical security level), a padded message fits in a single block and only one call to Keccak-f must be made.

Another important use case of variable capacities is the duplex construction. Here the block length of the modes that run on top of the duplex construction is a few bits shorter than the bitrate. By increasing the bitrate by a few bits (thus reducing the capacity by the same amount), a block length that is a multiple of 8, 32 or 64 can be maintained. If the initial capacity is high enough (e.g., 576 bits) reducing it by a few bits does not result in a noticeable reduction in security.

In [4] we provide a simple application to help determine the capacity value and output length given required security levels for collision-resistance and (second) pre-image resistance.

2.2.1 What about the bound on the $\mathcal{RO}$ differentiating advantage?

The proven upper bound on the $\mathcal{RO}$ differentiating advantage of [3] assumes the capacity is fixed and does not as such apply to a set of sponge functions calling the same underlying function with different capacity values. However, for the padding function used in Keccak, we have proven in [6] that the $\mathcal{RO}$ differentiating advantage of the set is upper bound by that of the member with the lowest capacity.

2.2.2 What about the implementation cost?

An argument against tunable parameters in a standard is that it makes implementations more expensive, as they usually have to support all parameter values to fully implement the standard. However, for Keccak, the main implementation cost is for the Keccak-$f[1600]$ permutation that is the same for all capacity values. The additional cost of the variable capacity value consists of the required support for the configurable bitrate r determining the length of the message blocks to be XORed into the state. The cost of supporting a variable capacity value with a fixed state width is therefore quite limited.

2.2.3 What about the burden of choice for the user?

Another argument against tuneable parameters in a standard is that it puts the burden of choice on the hash function user, typically a designer of a protocol or scheme. In particular, the choice of the capacity value determines a ceiling to the security level that the sponge function provides and one could argue that the user usually does not have the responsibility or the expertise to make that choice. In our opinion, the security claim of KECCAK is easy to understand and the user can be guided in the choice of the capacity by some simple recommendations. For example, one could fix a maximum capacity value $c_{\max}$ and recommend taking a capacity depending on the output length n. The capacity would be $c = 2n$ when $n < c_{\max}/2$ and $c = c_{\max}$ when $n \geq c_{\max}/2$.

3 Rationale

In KECCAK, there are basically three security-relevant parameters that can be varied: the width b of KECCAK-f, the capacity c, limited by $c < b$, and the number n_{r} of rounds in KECCAK-f. The parameters of the proposals of Section 2 have been chosen for the following reasons.

- $b = 1600$: The width of the KECCAK-f permutation is chosen to support all required capacity values using the same permutation. See Section 3.1 for further discussions.
- $c = 2n$: For the fixed-output-length candidates, we chose a capacity equal to twice the output length n. This is the smallest capacity value such that there are no generic attacks with expected complexity below 2^n.
- $c = 576$: For KECCAK[] with default parameters, we chose a rate value that is a power of two and a capacity not smaller than 512 bits and such that their sum equals 1600. This results in $r = 1024$ and $c = 576$. This capacity value precludes generic attacks with expected complexity below 2^{288}.
- $n_{\mathrm{r}} = 24$: The value of n_{r} has been chosen to have a good safety margin and still good performance, as detailed in Section 4.2.

3.1 Rationale for proposing KECCAK-f[1600]

All KECCAK members we propose for standardization make use of the same permutation: KECCAK-f[1600]. A single implementation of this permutation supports all the proposed variants, hence reducing cost, for instance, in hardware implementations. Furthermore, the choice of KECCAK-f[1600] favors 64-bit CPUs and yet remains efficient on 32-bit (and smaller) processors.

Software implementations of KECCAK-f use bitwise Boolean operations and (cyclic) shifts on CPU words. A typical implementation maps each lane to a CPU word, resulting in the state of KECCAK represented in 25 words of 64 bits each. The choice of the lane size therefore favors CPUs with the corresponding word size. Specifically, the implementation of KECCAK-f[1600] on a 64-bit CPU can exploit 64-bit wide Boolean operations and 64-bit rotations.

Because of the bit-oriented design of Keccak-f, other approaches are possible. For instance, Keccak-$f[1600]$ can be efficiently implemented on a 32-bit CPU by using the *bit interleaving technique* [13]. Note that the use of, for example, modular addition would have prevented the bit interleaving technique.

Some families of hash functions make use of two distinct compression functions, one oriented to 32-bit words and one to 64-bit words, in order to provide different output lengths and/or security levels. A full implementation on a given platform of such a family includes two separate compression functions, and hence at least one of the two will have a word length different from that of the CPU. In contrast, all Keccak members we propose for standardization can be implemented with a single permutation Keccak-$f[1600]$ that thanks to bit interleaving can work with either 25 words on a 64-bit CPU or 50 words on a 32-bit CPU.

In terms of memory footprint, Keccak-$f[1600]$ requires 200 bytes of RAM for the state and some working memory [13]. The sponge construction allows implementations to XOR the message block into the state directly, relieving the application from dedicating a memory area for it. This optimization applies where the hashing API is composed of functions such as `Init`, `Update` and `Final`. In general a message queue must be allocated, which can be avoided for sponge functions or similar.

The choice of width 1600 allows for a high bitrate even for high capacity values. For instance, Keccak can process 800 more input bits per evaluation of Keccak-$f[1600]$ than of Keccak-$f[800]$ when c is fixed. However, the designer of an application on a memory-constrained device may opt for a smaller state size by using an alternate set of parameters. Keccak$[r = 288, c = 512]$ for instance uses 100 bytes of RAM. And if 256 bits of capacity are enough for such an application, Keccak$[r = 144, c = 256]$ uses only 50 bytes. Similar ideas apply to hardware implementations, where Keccak-$f[800]$ and Keccak-$f[400]$ can be seen as compact alternatives. Using a smaller width has a price, though, as it requires to support another Keccak-f permutation. This may be acceptable if such an application is exceptional or operates in a rather closed system, freeing the standard from supporting anything else other than Keccak-$f[1600]$.

4 Safety margin

In this section, we explain how the safety margin in Keccak can be increased or decreased simply by changing the number of rounds in Keccak-f and explain why we think the nominal number of rounds provide a high safety margin. Finally, we describe two techniques to build a *safe mode* into Keccak implementations at little additional cost, which one could migrate to in the hypothetical case that a weakness in Keccak is found.

4.1 Changing the number of rounds

The number of rounds of the Keccak-f permutations is defined and fixed in [12] and reflects the trade-off between performance and safety margin made in the design. Nevertheless, the specifications make it easy to define Keccak with an increased or decreased number of rounds. With the exception of the addition of a round constant, the rounds are identical. As the round constants are defined for any number of rounds, it is sufficient to modify the total number of rounds in the specifications.

So, someone who would like to use Keccak but does not feel comfortable with its safety margin can simply adopt a version with more rounds. Someone who feels that Keccak has an excessive safety margin can adopt a version with fewer rounds.

4.2 The safety margin with the nominal number of rounds

As reflected in our estimates for the safety margin against different types of attack, we think Keccak-f has about twice as many rounds (24 vs 13) as strictly required for Keccak to stand up to its security claim, for any choice of the capacity.

In [12, Section "Choice of parameters: the number of rounds"], we estimate the number of rounds that is sufficient to provide resistance against four types of distinguisher or attack. For the proposals of Section 2, i.e., using $b = 1600$, these estimates translate as follows:

- Keccak-f distinguisher below $2^{b/2}$: 21 rounds;
- Keccak distinguisher: 13 rounds;
- Inner collision: 11 rounds;
- State recovery: 11 rounds.

4.3 Migration path in the presence of a deployed standard

We expect a hash standard to be ubiquitous both in software and dedicated hardware implementations. If a weakness is discovered that has a real-world security impact, it is beneficial to have an affordable migration path towards a version without this weakness. On the NIST SHA-3 mailing list Ron Rivest [18, 2-Aug-2009] and other researchers proposed having a security parameter (e.g., the number of rounds) to be determined by the user. Disadvantages of this approach were discussed and the most important ones are the increased implementation cost due to the additional parameter, the burden of having to choose the security parameter value by the hash function user and the risk of denial-of-service attacks. Moreover, the support of a smooth choice for the security parameter may actually introduce new weaknesses, as observed by Stefan Lucks in his message to the NIST SHA-3 mailing list [18, 3-Aug-2009].

In the most lightweight version of this approach the security parameter would have only two values: one nominal value and one high-security value (e.g., tripling the number of rounds). In case of emergency, it would then be possible to migrate to the high-security value. We describe here two methods for migrating to a more secure version that applies to Keccak without impact on the hash function implementation itself.

Both methods we propose consist of an input pre-processing step. In all use cases the input to a sponge function is a bitstring, typically made of message bits and possible key bits. After padding, the input consists of a sequence of r-bit blocks. Before presenting it to the sponge construction, this input can then be expanded by inserting bytes with fixed values in certain places. Depending on where these bytes are inserted, this has an effect similar to reducing the rate of the sponge function or multiplying the number of rounds of the underlying permutation.

The first option is to reduce the effective bitrate from r bits to $r - \delta$ bits by inserting after every input block of $r - \delta$ bits a block of δ bits equal to zero. This reduces the number of bits an attacker can exploit from r to $r - \delta$. Note that with this approach the hermetic sponge strategy is abandoned as the effective capacity is increased while the claimed capacity stays fixed.

The second option is to multiply the effective number of rounds of the underlying permutation by a factor α by inserting after every input block of r bits $\alpha - 1$ blocks of r bits with fixed and well-defined values. The α applications of the underlying permutation interleaved with the application of the fixed blocks, can then be seen as a single permutation with α as many rounds as the original one and with the fixed-value blocks as round constants. As it is generally expected that increasing the number of rounds increases the safety margin with respect to almost all attacks, this provides a migration path to a security fix in case of a hypothetical security weakness. In this case the hermetic sponge strategy can be maintained as the single permutation with α as many rounds is assumed to have no structural distinguishers.

Both methods have the advantage of leaving KECCAK-f untouched, which limits the cost of migrating should the need occur.

5 Usage

In a sponge function, the input is like a *white page*: It does not impose any specific structure to it. Additional optional inputs (e.g., key, nonce, personalization data) can be appended or prepended to the input message according to a well-defined convention, possibly under the hood of diversification as proposed in [6, Section "Domain separation"].

KECCAK supports all the possible applications of sponge functions and duplex objects described in [6, Chapters "Sponge applications" and "Duplex applications"]. These include hash function, randomized hash function, hash function instance differentiation, slow one-way function, parallel and tree hashing, mask generating function, key derivation function, deterministic random bit generator, reseedable pseudo random bit sequence generator, message authentication code (MAC) function, stream cipher, random-access stream cipher and authenticated encryption.

5.1 Backward compatibility with old standards

In addition to the functionality natively provided by sponge functions, we describe how to provide backward compatibility with old standards when required.

5.1.1 Input block length and output length

Several standards that make use of a hash function assume it has an input block length and a fixed output length. A sponge function supports inputs of any length and returns an output of arbitrary length. When a sponge function is used in those cases, an input block length and an output length must be chosen. We distinguish two cases.

- For the four SHA-3 candidates where the digest length is fixed, the input block length is assumed to be the bitrate r and the output length is the digest length of the candidate $n \in \{224, 256, 384, 512\}$.
- For an instance with variable-length output, the output length n must be explicitly chosen to fit a particular standard. Since the input block length is usually assumed to be greater than or equal to the output length, the input block length can be taken as an integer multiple of the bitrate, mr, to satisfy this constraint.

5.1.2 Initial value

Some constructions that make use of hash functions assume the existence of a so-called initial value (IV) and use this as additional input. In the sponge construction the root state could be considered as such an IV. However, for the security of the sponge construction it is crucial that the root state is fixed and cannot be manipulated by the adversary. If KECCAK sponge functions are used in constructions that require it to have an initial value as supplementary input, e.g., as in NMAC [1], this initial value shall just be pre-pended to the regular input.

5.1.3 HMAC

HMAC [1, 25] is fully specified in terms of a hash function, so it can be applied as such using one of the KECCAK candidates. It is parameterized by an input block length and an output length, which we propose to choose as in Section 5.1.1 above.

Apart from length extension attacks, the security of HMAC comes essentially from the security of its inner hash. The inner hash is obtained by prepending the message with the key, which gives a secure MAC. The outer hash prepends the inner MAC with the key (but padded differently), so again giving a secure MAC. Of course, it is also possible to use the generic MAC construction given in [6], which requires only one application of the sponge function.

From the security claim in [12], a PRF constructed using HMAC shall resist a distinguishing attack that requires much fewer than $2^{c/2}$ queries and significantly less computation than a pre-image attack.

5.1.4 NIST and other relevant standards

The following standards are based either generically on a hash function or on HMAC. In all cases, at least one of the KECCAK candidates can readily be used as the required hash function or via HMAC.

- IEEE P1393 [16] requires a hash function for a key derivation function (X9.42) and a mask generating function (MGF-hash). Note that the MGF-hash construction could be advantageously replaced by the arbitrarily-long output mode of KECCAK[].
- PKCS #1 [17] also requires a hash function for a mask generating function (MGF1).
- The key derivation functions in NIST SP 800-108 [26] rely on HMAC.
- The key derivation functions in NIST SP 800-56a [22] are generically based on a hash function.
- The digital signature standard (DSS) [19] makes use of a hash function with output size of 160, 224 or 256 bits. Output truncation is permitted so any of the five KECCAK candidates can be chosen to produce the 160 bits of output.
- In the randomized hashing digital signatures of NIST SP 800-106 [21], the message is randomized prior to hashing, so this is independent of the hash function used. With a sponge function, this can also be done by prepending the random value to the message.
- The deterministic random bit generation (DRBG) in NIST SP 800-90 [23] is based on either a hash function or on HMAC. Note that an efficient and simple DRBG can be implemented based on a sponge function [6]

6 Implementations

In [11], we propose a reference implementation in C, which covers the instances of Section 2, namely, based on KECCAK-f[1600] and r a multiple of 64 bits. It also implements the duplex construction, without restrictions on r.

In addition, KECCAKTOOLS can serve as a reference implementation in C++ [9]. All the members KECCAK$[r, c]$ can be instantiated, including those based on the smaller permutation widths, and without restrictions on the values of r and c.

All the implementation-related aspects are treated in [13].

6.1 Bit and byte numbering

In this section, we detail the mapping between the bits of the input to the KECCAK sponge functions, and their representation in the SHA-3 API defined by NIST [24]. The bits of the input message M are numbered from $i = 0$ to $i = |M| - 1$. When the bits are gathered in bytes, it implies the equivalent numbering $i = i_{\text{bit}} + 8i_{\text{byte}}$ with $0 \leq i_{\text{bit}} < 8$.

In our internal convention, $i_{\text{bit}} = 0$ indicates the least significant bit (LSB) of a byte while $i_{\text{bit}} = 7$ indicates the most significant one (MSB) [13]. The NIST convention [24] is different: The bits of a byte are numbered from 0 for the MSB to 7 for the LSB. To be compatible with the API convention outside and with our convention inside, the following *formal* bit-reordering is performed on the input bit string M before it is processed:

- For all bytes that contain 8 bits, position $i_{\text{bit}} + 8i_{\text{byte}}$ is mapped to position $7 - i_{\text{bit}} + 8i_{\text{byte}}$.
- For the last byte if it contains $p < 8$ bits, position $i_{\text{bit}} + 8i_{\text{byte}}$ is mapped to position $(p-1) - i_{\text{bit}} + 8i_{\text{byte}}$.

This mapping is bijective and does not affect the security.

In practice, the above operation cancels with the change of convention, so there is nothing to do, except:

- For the last byte if it contains $p < 8$ bits, the bits are shifted by $8 - p$ positions towards the LSB (i.e., to the "right").

The rationale behind this convention can be found in [13].

7 Change history

In this section, we summarize the changes from the round 1 to the round 2 proposals, then from the round 2 to the round 3 proposals.

7.1 From round 1 to round 2

From round 1 to round 2, we made the following changes.

- We changed the number of rounds of KECCAK-f from $12 + \ell$ to $12 + 2\ell$.
- We changed the capacity and bitrate of the four fixed-output-length candidates. For each of them we set the capacity to twice the output length.

7.2 From round 2 to round 3

For round 3, we made the following changes.

- We shortened and simplified the padding rule. The new padding rule is the pad10*1 rule [6].
- We removed the diversifier parameter d.
- We removed the restriction on the supported values of r. Previously, the bitrate r could only take values that are multiple of 8 bits. Now all the values $0 < r \leq b$ are supported.

Note that we made no changes to KECCAK-f.

7.3 Rationale for changing the padding rule

As explained in [6], the sponge construction offers protection against generic attacks if the padding rule is sponge-compliant, i.e., it is injective and ensures that the last block is different from the all-zero block. The simplest padding rule that satisfies these requirements is the simple padding pad10*.

KECCAK uses the same permutation with different bitrates. In this case, sponge-compliance is not sufficient anymore to offer protection against generic attacks. To overcome this, the bitrate was explicitly encoded in the padding rule of the round 1 and round 2 proposals. This way, the security of a set of sponge functions using of the same permutation was formally proved to be equal to that of its member with the smallest capacity.

In [6] we prove that the padding rule pad10*1 also satisfies this property. It is simpler to describe and to implement than the rounds 1-2 padding rule. It is also more efficient, as it appends a minimum of 2 bits instead of 25. For long messages, the gain is negligible, but short messages can be 3 bytes longer for the same number of calls to KECCAK-f. This aspect is especially relevant when using the duplex construction.

7.4 Rationale for removing the diversifier parameter

In the round 1 proposal, some fixed-output-length candidates shared the same bitrate value. Hence, the diversifier was added to the padding rule to ensure diversification between the different fixed-output-length candidates. This addressed a requirement expressed by NIST on the hash forum mailing list [18, 23-Jun-2008], that a hash function with a given output length should not be the prefix of another one with larger output length.

Starting from the round 2 proposal, all the fixed-output-length candidates have different bitrates, hence making the diversifier parameter useless for this requirement. Of course, diversification between functions using the same bitrate is still possible, as discussed in Section 2.1.2.

7.5 Rationale for removing restrictions on the bitrate

In the round 1 and 2 proposals, the bitrate was encoded in terms of bytes in the padding rule. Furthermore, the padding rule was designed only for bitrates multiple of 8 bits. With the simpler padding rule pad10*1, this restriction is no longer needed.

A bitrate multiple of 8 bits is a natural choice for sponge functions in most usage scenarios. Without it, undesirable intra-byte bit shuffling is going to happen on the input blocks.

For a duplex object, however, the situation is different [6]. The duplex construction accepts input blocks up to the maximum duplex rate $\rho_{\max}$, which is just two bit shorter than the bitrate, $\rho_{\max} = r - 2$. Furthermore, some modes that run on top of the duplex construction can make use of a frame bit. The application-level block size is therefore reduced by 2 or 3 bits. To ensure that the application-level block size ρ is a multiple of 8, 32 or 64 bits, the bitrate must be chosen to be 2 or 3 modulo 8, 32 or 64. For instance, $r = 1026$ allows a maximum duplex rate of $\rho_{\max} = 1024$ bits, and $r = 1027$ ensures that the block size of the duplex-based authenticated encryption scheme SPONGEWRAP is conveniently 1024 bits.

7.6 Rationale for *not* changing the number of rounds

In September 2009, we increased the number of rounds from 18 to 24 in KECCAK-f[1600] due to our adoption of the hermetic sponge strategy [6] and our wish to keep a safety margin against all distinguishers [5].

The zero-sum distinguishers recently found by Boura, Canteaut and De Cannière and by Duan and Lai distinguish the 24 rounds of Keccak-$f[1600]$ from a randomly-chosen permutation, although without implying a distinguisher on Keccak itself [14, 15]. Not increasing the number of rounds, strictly speaking, contradicts the hermetic sponge strategy. Still, we decided to stick the number of rounds specified by $12 + 2\ell$ due to the fact that these distinguishers can in no way be used to attack Keccak for many reasons. One of them is that the required number of queries, something like 2^{1575}, is not only extremely high but also way above the claimed security of Keccak. Hence, it is easy to prove that it has no impact on its security, as detailed in[12, Section "Zero-sum distinguishers"] and in [6, Section "The usability of structural distinguishers"].

According to our estimates (see Section 4.2), the current nominal number of rounds already provides a comfortable safety margin.

8 NIST requirements

In this section, we provide a mapping from the items required by NIST to the appropriate sections in this and other documents.

- Requirements in [20, Section 2.B.1]
  - The complete specifications can be found in [12, Chapter "Keccak specifications"].
  - Design rationale: a summary is provided in [12, Chapter "Design rationale summary"], with pointers to sections with more details.
  - Any security argument and a preliminary analysis: this is the purpose of the document [12].
  - Tunable parameters: a summary is provided in Section 3, with pointers to sections with more details.
  - Recommended value for each digest size: see [12, Section "Choice of parameters: the number of rounds"] for the number of rounds and Section 2 for the other parameters.
  - Bounds below which we expect cryptanalysis to become practical: this can be found in Section 4.2.
- Requirements in [20, Section 2.B.2]
  - For the measured efficiency on various platforms, we refer to the eBASH and XBX benchmarks [2, 27].
  - We also refer to [13] for more information on the implementation aspects and to [7, 8] for summaries of hardware and software performance figures.
  - The speed estimate on the reference platform can also be found in eBASH [2, e.g., machine `cobra`].
- Requirements in [20, Section 2.B.3]
  - The known answer and Monte Carlo results can be found in [10].
- Requirements in [20, Section 2.B.4]
  - The expected strength of Keccak is stated in [12, Section "Security claim…"].

  - The link between the security claim and the expected strength criteria listed in [20, Section 4.A] can be found in [6, Sections "Sponge functions used as a hash function" and "Implications of the bound on the RO differentiating advantage"].
  - Other pseudo random functions (PRF) constructions: some modes of use are proposed in [6, Section "Modes of use…"].

- Requirements in [20, Section 2.B.5]
  - We formally state that we have not inserted any trapdoor or any hidden weakness in KECCAK. Moreover, we believe that the structure of the KECCAK-f permutation does not offer enough degrees of freedom to hide a trapdoor or any other weakness.

- Requirements in [20, Section 2.B.6]
  - Advantages and limitations: a summary is provided in [12, Chapter "Design rationale summary"], with pointers to sections with more details.

References

[1] M. Bellare, R. Canetti, and H. Krawczyk, *Keying hash functions for message authentication*, Advances in Cryptology – Crypto '96 (N. Koblitz, ed.), LNCS, no. 1109, Springer-Verlag, 1996, pp. 1–15.

[2] D. J. Bernstein and T. Lange (editors), *eBACS: ECRYPT Benchmarking of cryptographic systems*, `http://bench.cr.yp.to`, accessed 21 December 2010.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, `http://sponge.noekeon.org/`, pp. 181–197.

[4] ______, *Tune* KECCAK *to your requirements*, 2009, `http://keccak.noekeon.org/tune.html`.

[5] ______, *Note on zero-sum distinguishers of* KECCAK-f, Comment on the NIST Hash Competition Forum, January 2010, `http://keccak.noekeon.org/NoteZeroSum.pdf`.

[6] ______, *Cryptographic sponge functions*, January 2011, `http://sponge.noekeon.org/`.

[7] ______, KECCAK *hardware performance figures page*, 2011, `http://keccak.noekeon.org/hw_performance.html`.

[8] ______, KECCAK *software performance figures page*, 2011, `http://keccak.noekeon.org/sw_performance.html`.

[9] ______, KECCAKTOOLS *software*, January 2011, `http://keccak.noekeon.org/`.

[10] ______, *Known-answer and Monte Carlo test results*, 2011, available from `http://keccak.noekeon.org/`.

[11] ______, *Reference and optimized implementations of* KECCAK, 2011, available from `http://keccak.noekeon.org/`.

[12] ______, *The* KECCAK *reference*, January 2011, `http://keccak.noekeon.org/`.

[13] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, Keccak *implementation overview*, January 2011, `http://keccak.noekeon.org/`.

[14] C. Boura, A. Canteaut, and C. De Cannière, *Higher-order differential properties of Keccak and Luffa*, Fast Software Encryption 2011, 2011, to appear, draft available from Cryptology ePrint Archive, Report 2010/589.

[15] M. Duan and X. Lai, *Improved zero-sum distinguisher for full round keccak-f permutation*, Cryptology ePrint Archive, Report 2011/023, 2011, `http://eprint.iacr.org/`.

[16] IEEE, *P1363-2000, standard specifications for public key cryptography*, 2000.

[17] RSA Laboratories, *PKCS # 1 v2.1 RSA Cryptography Standard*, 2002.

[18] NIST, *Mailing list on NIST's cryptographic hash workshops and hash algorithm competition*, `http://csrc.nist.gov/groups/ST/hash/email_list.html`.

[19] ———, *Federal information processing standard 186-3, digital signature standard (DSS)*, March 2006.

[20] ———, *Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family*, Federal Register Notices **72** (2007), no. 212, 62212–62220, `http://csrc.nist.gov/groups/ST/hash/index.html`.

[21] ———, *NIST special publication 800-106 draft, randomized hashing digital signatures*, July 2007.

[22] ———, *NIST special publication 800-56a, recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)*, March 2007.

[23] ———, *NIST special publication 800-90, recommendation for random number generation using deterministic random bit generators (revised)*, March 2007.

[24] ———, *ANSI C cryptographic API profile for SHA-3 candidate algorithm submissions, revision 5*, February 2008, available from `http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/crypto_API.html`.

[25] ———, *Federal information processing standard 198, the keyed-hash message authentication code (HMAC)*, July 2008.

[26] ———, *NIST special publication 800-108, recommendation for key derivation using pseudorandom functions*, April 2008.

[27] C. Wenzel-Benner and J. Gräf, *XBX: keeping two eyes on embedded hashing*, `http://xbx.das-labor.org/`, accessed 21 December 2010.