From: hash-forum@nist.gov on behalf of Anne Canteaut [Anne.Canteaut@inria.fr]
Sent: Thursday, January 14, 2010 5:01 AM
To: Multiple recipients of list
Subject: OFFICIAL COMMENT: Keccak

Dear all,

The note at
http://www-roc.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pdf

shows how the zero-sum property exhibited by J.P. Aumasson and W. Meier can be extended to 18 rounds of the Keccak-f permutation, i.e., to the permutation used in the original version of Keccak.
This structural property does not seem to affect the security of the hash function, but it points out that the inner permutation with 18 rounds does not have an ideal behavior, contradicting the "hermetic sponge strategy".
Note that the new version of Keccak uses Keccak-f with 24 rounds.

Christina Boura and Anne Canteaut.

From: hash-forum@nist.gov on behalf of Dmitry Khovratovich [khovratovich@gmail.com]
Sent: Thursday, January 14, 2010 8:09 AM
To: Multiple recipients of list
Subject: Re: OFFICIAL COMMENT: Keccak

Dear Anne and Christina,

You say in p.10 "this algorithm leads to several partitions of the input space F^1600_2 into zero-sums of size 2^1370, which is clearly a structural distinguishing property".

I am not sure that it is a structural distinguisher property. Even if we do not consider the lower bound for finding a zero-sum for a random permutation, the upper bound is given by the generalized birthday algorithm, which allows to find a zero-sum of size 2^55 in 2^{55+3200/(1+55)} < 2^115 operations, which is smaller than 2^1370.

On 1/14/10, Anne Canteaut <Anne.Canteaut@inria.fr> wrote:
>
> Dear all,
>
> The note at
> http://www-roc.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pdf
>
> shows how the zero-sum property exhibited by J.P. Aumasson and W.
> Meier can be extended to 18 rounds of the Keccak-f permutation, i.e.,
> to the permutation used in the original version of Keccak.
> This structural property does not seem to affect the security of the
> hash function, but it points out that the inner permutation with 18
> rounds does not have an ideal behavior, contradicting the "hermetic
> sponge strategy".
> Note that the new version of Keccak uses Keccak-f with 24 rounds.
>
> Christina Boura and Anne Canteaut.
>
>

--
Best regards,
Dmitry Khovratovich

University of Luxembourg,
Laboratory of Algorithmics, Cryptography and Security,
+ 352 46 66 44 5478

From: hash-forum@nist.gov on behalf of Anne Canteaut [Anne.Canteaut@inria.fr]
Sent: Thursday, January 14, 2010 10:43 AM
To: Multiple recipients of list
Subject: Re: OFFICIAL COMMENT: Keccak

Dear Dmitry,

The algorithm you mention provides a single zero-sum, while our result (exactly as the previous result due to Jean-Philippe Aumasson and Willi
Meier) provides several partitioning of the input space into zero-sums, i.e., for the considered parameters, some collections of 2^230 disjoint zero-sums covering the entire input space.
This is obviously a much stronger structural property.

Anne Canteaut.

Dmitry Khovratovich wrote:
> Dear Anne and Christina,
>
> You say in p.10 "this algorithm leads to several partitions of the
> input space F^1600_2 into zero-sums of size 2^1370, which is clearly a
> structural distinguishing property".
>
> I am not sure that it is a structural distinguisher property. Even if
> we do not consider the lower bound for finding a zero-sum for a random
> permutation, the upper bound is given by the generalized birthday
> algorithm, which allows to find a zero-sum of size 2^55 in
> 2^{55+3200/(1+55)} < 2^115 operations, which is smaller than 2^1370.
>
> On 1/14/10, Anne Canteaut <Anne.Canteaut@inria.fr> wrote:
>> Dear all,
>>
>> The note at
>> http://www-roc.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pd
>> f
>>
>> shows how the zero-sum property exhibited by J.P. Aumasson and W.
>> Meier can be extended to 18 rounds of the Keccak-f permutation, i.e.,
>> to the permutation used in the original version of Keccak.
>> This structural property does not seem to affect the security of the
>> hash function, but it points out that the inner permutation with 18
>> rounds does not have an ideal behavior, contradicting the "hermetic
>> sponge strategy".
>> Note that the new version of Keccak uses Keccak-f with 24 rounds.
>>
>> Christina Boura and Anne Canteaut.
>>
>>
>
>

From: hash-forum@nist.gov on behalf of Guido Marco BERTONI [guido.bertoni@st.com]
Sent: Friday, January 15, 2010 9:01 AM
To: Multiple recipients of list
Subject: OFFICIAL COMMENT: Keccak (Round 2)

Attachments: NoteZeroSum.pdf

NoteZeroSum.pdf
(144 KB)

In September last year, Jean-Philippe Aumasson and Willi Meier introduced zero-sum distinguishers, a method to generate zero-sum structures for reduced-round versions of Keccak-f up to 16 rounds.
Recently, Christina Boura and Anne Canteaut extended this to 18 rounds.
Please find in attachment a note, in which we discuss these distinguishers and their implications.

The Keccak Team

Note on zero-sum distinguishers of Keccak-f

In [1], Jean-Philippe Aumasson and Willi Meier introduced *zero-sum distinguishers*, a method to generate zero-sum structures for reduced-round Keccak-f[1600], the permutation underlying our SHA-3 submission Keccak. Their paper contained distinguishers for up to 16 rounds of Keccak-f[1600]. Recently, Christina Boura and Anne Canteaut extended this to 18 rounds in [6]. In this note we argue that the distinguishers are valid and are qualitatively different from generic methods, as they can partition the set of inputs into sets of zero-sum structures of specific sizes. We also put this in perspective, as generic methods allow generating zero-sum structures of small sizes, and the distinguishers covering more rounds have extremely high complexity (e.g., 2^{1369} for 18 rounds).

Nevertheless, after the publication of [1], we decided to increase the number of rounds of Keccak-f. The logic underlying this decision is our adoption of the *hermetic sponge strategy*, in which we tolerate no structural distinguisher for the permutation used in the sponge construction. The strength or applicability of the distinguisher in the context of the sponge construction plays no role in this aspect.

By increasing the number of rounds, we believe to have re-established the security margin of Keccak-f with respect to structural distinguishers.

1 The challenge

A zero sum structure for a function f is defined in [1] as a set $\mathcal{Z}$ of inputs z_i that sum to zero, and for which the corresponding outputs also sum to zero. The challenge is now to generate such a set in an efficient way. Hence the challenge is the following.

Challenge: given a function f from n to m bits and an integer N, construct a set $\mathcal{Z}$ of N inputs z_i (or the set of corresponding f-images) such that:

$$\sum_{0\leq i<N} z_i = 0 \text{ and } \sum_{0\leq i<N} f(z_i) = 0.$$

Given all inputs in $\mathcal{Z}$ except one and the f-outputs of all inputs of $\mathcal{Z}$ but one, the zero-sum structure allows the computation of the missing input and output by simply summing over the known elements and hence without calling f. If the size of $\mathcal{Z}$ is small, this may give an adversary an advantage in an attack. Clearly, the advantage diminishes as the size of $\mathcal{Z}$ grows.

2 A generic method

In this section we present a generic method for constructing a zero-sum structure inspired by Wagner's algorithm for the generalized birthday problem in [7] and by the attack against XHASH in [2], brought to our attention by Jean-Philippe Aumasson. For a method to become a structural distinguisher for a particular function, it shall have a lower complexity than this generic method.

Here is an outline of the method. We use the following notation: $X_i = [x_i|f(x_i)]^{\mathrm{T}}$, i.e., a column vector with components the bits of x_i followed by the bits of $f(x_i)$.

1. Take N random values x_i, compute $f(x_i)$ and form $X_i = [x_i|f(x_i)]^{\mathrm{T}}$.

2. Compute the bitwise sum of the vectors X_i and call the sum A:

$$\sum_{0\leq i<N} X_i = A\ . \tag{1}$$

3. Take $p = n + m + \epsilon$ random values y_i with $0 \leq i < p$ and ϵ a small integer, compute $f(y_i)$ and form $Y_i = [y_i|f(y_i)]^{\mathrm{T}}$.
4. Solve the following linear system of $n + m$ equations in the $n + m + \epsilon$ variables a_i over GF(2), with the bits of $(X_i \oplus Y_i)$ serving as (fixed) coefficients:

$$\sum_{0 \leq i < p} a_i(X_i \oplus Y_i) = A. \tag{2}$$

5. For a solution (a_i), form the set $\mathcal{Z}$ such that

$$z_i = \begin{cases} y_i & \text{if } i < p \text{ and } a_i = 1 \\ x_i & \text{otherwise.} \end{cases}$$

Clearly, the set $\mathcal{Z}$ is a zero-sum structure. Adding equations (1) and (2) gives:

$$\sum_{0 \leq i < N} X_i \oplus \sum_{0 \leq i < p} a_i(X_i \oplus Y_i) = \sum_{0 \leq i < p} (a_i Y_i \oplus \bar{a}_i X_i) \oplus \sum_{p \leq i < N} X_i = \sum_{0 \leq i < N} Z_i = 0.$$

This method requires that $p = n + m + \epsilon \leq N$ for some $\epsilon \geq 0$. The value of ϵ determines the a priori probability that the system of equations (2) has a solution: by increasing ϵ the probability that it has no solution decreases exponentially. If $N \gg n + m$, the probability of failure can be made arbitrarily small by increasing ϵ and the complexity can be approximated by N executions of f.

The computational effort is the sum of:

- $N + n + m + \epsilon$ evaluations of f,
- solving a system of $n + m$ linear equations in $n + m + \epsilon$ variables over GF(2), which can be done very efficiently, and
- taking the bitwise sum of N $(n + m)$-bit vectors.

3 The zero-sum distinguishers on KECCAK-f

The method for constructing zero-sum structures described in [1, 6] exploits the fact that adding a round in KECCAK-f only doubles the degree of the algebraic expression of the output bits in terms of the input bits, and only triples the degree of the algebraic expression of the input bits in terms of the output bits.

We discuss here the aspects that are relevant for the computational complexity of constructing the distinguishers and refer to [1, 6] for the details. We consider the complexity of the method constructing the set $\mathcal{Z}$ or the set of corresponding outputs.

Compared to the generic method, the method in [1, 6] has the following features.

- First, as opposed to the generic method, this method is deterministic.
- Second, in this method the size of $\mathcal{Z}$ cannot be freely chosen (above some minimum) but is limited to powers of two (above some minimum).
- Third, the non-maximal degree of the two parts of the (reduced-round) KECCAK-f permutation can be used to create partitions of inputs in many different zero-sum structures. The size of such partitions, using this method, is a multiple of the size of the individual zero-sum structures. Producing a single zero-sum structure still leads to the fastest distinguisher in this context.

Rounds	inv. + forw.	N	Rounds	inv. + forw.	N
6	$2+4$	2^{10}	12	$5+7$	2^{129}
7	$3+4$	2^{15}	13	$6+7$	2^{244}
8	$3+5$	2^{18}	14	$6+8$	2^{257}
9	$4+5$	2^{30}	15	$6+9$	2^{513}
10	$4+6$	2^{60}	16	$6+10$	2^{1025}
11	$5+6$	2^{60}	18	$7+11$	2^{1370}

Table 1: Size of zero-sum structures for reduced-round Keccak-$f[1600]$ given in [1, 6]

For constructing $\mathcal{Z}$, one takes N states y_i (that forms a vector space of limited dimension) in some intermediate round and computes a number of rounds backwards to obtain the inputs z_i. Clearly, the complexity of this option (*the backward option*) is hence N times the computation of these inverse rounds. For constructing the outputs corresponding to $\mathcal{Z}$ (*the forward option*), one must compute a number of rounds forwards and the complexity is N times this forward computation. Table 1 lists the values of N for reduced-round versions of Keccak-$f[1600]$ for given number of rounds. It also gives the number of inverse rounds and forward rounds.

As seen in Table 1, for all cases the number of inverse rounds is smaller than the number of forward rounds. Hence at first sight the backward option seems to be the most efficient one. However, mainly due to the complexity of the inverse of θ [5], the computation of the inverse round of Keccak-$f[1600]$ has much higher complexity than the round itself. We think it is safe to assume that the inverse round takes twice as many computations as the forward round. In this light the forward option becomes the most efficient one. As the number of forward rounds is greater than half the number of rounds, the complexity of the method can be expressed as the computational equivalent of at least $N/2$ calls to the function under attack.

4 Implications for Keccak-f

For the values of N given in Table 1 (and any larger power of two), the method for generating zero-sum structures of [1, 6] is more efficient than the generic method by a factor 2. Hence, the zero-sum distinguishers of [1, 6] are valid, albeit with a very small advantage. For instance, consider the case of Keccak-$f[1600]$ reduced to 18 rounds. The method of [6] for the smallest value of N would have complexity 2^{1369} while for the generic method this is 2^{1370}. Note however that the generic method additionally allows generating zero-sum structures with any size $N > 3200$ at the cost of about $N + 3200$ calls to the function under attack.

We think it is very unlikely that the zero-sum distinguishers can result in the speedup of actual attacks against Keccak calling reduced-round versions of Keccak-f. Still, the distinguishers described in [1, 6] show non-ideal properties of the (reduced-round) Keccak-f permutation and suggested us to increase the number of rounds in Keccak-f [4].

The main reason behind this is our adoption of the hermetic sponge strategy [3]. This strategy imposes Keccak-f to be free from structural distinguishers, without considering their strength or relevance for the Keccak sponge function.

The existence of the distinguisher in [1] over 16 (out of 18) rounds of Keccak-$f[1600]$ left only a security margin of 2 rounds. Moreover, we wanted to increase the security margin against other possible distinguishers that start from the middle and compute back- and forwards to get the corresponding in- and outputs. In this method adding two rounds to Keccak-f only increases the algebraic degree to be considered in the attack by a factor 3. This is due to the fact that a Keccak-f round has degree 2 and its inverse only 3 [5]. We estimated that other types of distinguishers may be found that also exploit this fact or that the distinguishers may be further refined (e.g., as done in [6]). Therefore we decided to address this in round 2 of the SHA-3 competition by increasing the number of rounds (e.g., for Keccak-$f[1600]$ from 18 to 24 rounds).

The Keccak Team, January 2010
Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche

References

[1] J.-P. Aumasson and W. Meier, *Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi*, Available online, 2009, `http://131002.net/data/papers/AM09.pdf`.

[2] M. Bellare and D. Micciancio, *A new paradigm for collision-free hashing: Incrementality at reduced cost*, Eurocrypt, 1997, pp. 163–192.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic sponges*, 2009, `http://sponge.noekeon.org/`.

[4] ———, Keccak *specifications, version 2*, NIST SHA-3 Submission, September 2009, `http://keccak.noekeon.org/`.

[5] ———, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), September 2009, `http://keccak.noekeon.org/`.

[6] C. Boura and A. Canteaut, *A zero-sum property for the Keccak-f permutation with 18 rounds*, Available online, 2010, `http://www-roc.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pdf`.

[7] D. Wagner, *A generalized birthday problem*, CRYPTO (M. Yung, ed.), Lecture Notes in Computer Science, vol. 2442, Springer, 2002, pp. 288–303.

From: Gilles VAN ASSCHE [gilles.vanassche@st.com]
Sent: Tuesday, February 23, 2010 7:48 AM
To: hash-function@nist.gov
Cc: hash-forum@nist.gov
Subject: OFFICIAL COMMENT: Keccak (Round 2)

Attachments: NoteOnKeccakParametersAndUsage.pdf

NoteOnKeccakParametersAndUsage...

The Keccak sponge function family is characterized by three parameters: the bitrate r, the capacity c and the diversifier d. In the Keccak specifications we propose four instances that can be taken as functions for the four (fixed) output lengths NIST requires for SHA-3 and a variable-output-length instance, with default values for the parameters.

Whilst we are happy with our choice, there are other valid parameter choices that NIST or others may prefer. In this note we discuss our choice of parameters and other possible ways of using the Keccak family.

Kind regards,
The Keccak team

Note on Keccak parameters and usage

The Keccak sponge function family is characterized by three parameters: the bitrate r, the capacity c (where $r + c$ is the width of the underlying permutation) and the diversifier d. We propose in [5] four instances that can be taken as functions for the four (fixed) output lengths NIST requires for SHA-3 and a variable-output-length instance, denoted by Keccak[], with default values for the parameters. Section 1 below recalls the Keccak offering: its parameters, security claim and design strategy, and our proposal to NIST.

Whilst we are happy with our choice, there are other valid parameter choices that NIST or others may prefer. In this note we discuss our choice of parameters and other possible ways of using the Keccak family.

With its arbitrary length, the output of Keccak can be truncated at the length requested by the user. In Section 2 we discuss how using a single function has clear advantages and, if needed, simple ways to achieve diversification. The capacity c is the security parameter of Keccak and the use of a single instance with fixed capacity puts a ceiling on the achievable security level. We explain in Section 2 why this limit for Keccak[] is high enough not to be a problem.

In Section 3 we examine the issues of letting the user choose the capacity c while keeping $c + r = 1600$, which allows trading off claimed security for speed by increasing c and decreasing r, or vice versa.

In Section 4 we explain the choice of Keccak-f[1600] in our standard proposal and discuss where the Keccak-f permutations with other widths may be adequate.

A way to exploit parallelism is to apply tree hashing. This is especially relevant on modern CPUs with their multiple cores and SIMD architecture within a core. In Section 5 we explain that a tree hashing mode calling Keccak as a compression function can take advantage of both.

Finally, we address the question of migration possibilities to a more secure version, should Keccak be chosen as a standard and a weakness be discovered later. We propose in Section 6 two techniques based on input pre-processing with very limited impact on implementations.

1 The Keccak offering

As defined in [5], Keccak is a family of sponge functions with members Keccak$[r, c, d]$ characterized by three parameters:

- bitrate r,
- capacity c and
- diversifier d.

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 diversifier value satisfies $0 \leq d < 256$.

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.

The purpose of the diversifier is to provide diversification, i.e., two instances of Keccak with two different values of d behave as two independent hash functions (even with same values of r and c). See Section 2.2 for a discussion.

1.1 The security claim and design strategy

KECCAK allows one to choose its security parameter c independently from the output length. We express our security claim for KECCAK in [5] as a *flat sponge claim* [2]. This type of claim implies that the expected complexity of any attack should be the same as for a random oracle, up to $2^{c_{\text{claim}}/2}$. 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, Chapter 4]. 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.

Additional information on the security claim and design strategy is given in [4].

1.2 Our proposal for SHA-3

In [14], 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 have defined four fixed-output-length variants (where $\lfloor \rfloor_n$ indicates truncation to the first n bits):

- $n = 224$: $\lfloor \text{KECCAK}[r = 1156, c = 448, d = 28] \rfloor_{224}$
- $n = 256$: $\lfloor \text{KECCAK}[r = 1088, c = 512, d = 32] \rfloor_{256}$
- $n = 384$: $\lfloor \text{KECCAK}[r = 832, c = 768, d = 48] \rfloor_{384}$
- $n = 512$: $\lfloor \text{KECCAK}[r = 576, c = 1024, d = 64] \rfloor_{512}$

The capacity values were chosen to meet the requirement that (second) preimage resistance should be 2^n (with n the output length). The different diversifier values $d = n/8$ address a requirement expressed by NIST on the hash forum mailing list [13, 23-Jun-2008], that a hash function with a given output length should not be the prefix of another one with larger output length.

In addition, we proposed KECCAK[] (with default parameters), where the user may truncate the output at the desired output length. The default bitrate $r = 1024$ is a power of two to ease data alignment and the resulting capacity is $c = 1600 - 1024 = 576$. The default value for the diversifier d is 0.

2 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 [11, 12]. 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 [11, 12].

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 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) preimage resistance. As stated in Section 1.1, 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) preimage 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 [8].

2.2 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. In Keccak[] the diversifier is fixed to 0 and as such does not appear to address this requirement. However, 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 KeccakNS for diversification [6, Section 6.3]. The use of the diversifier d is actually a built-in way to achieve domain separation.

3 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[1] 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 [4], 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.

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) preimage 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

[1]We limit the choice to multiples of 8 to avoid intra-byte bit shuffling.

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$[]$.

by 32 bits resulting in a two-block message and hence applying Keccak[] results in two calls to Keccak-f. If we decrease the capacity by 32 bits to 544 (still providing an astronomical security level), a padded message fits in a single block and only one call to Keccak-f must be made.

In [8] we provide a simple application to help determine the capacity value and output length given required security levels for collision-resistance and (second) preimage resistance.

3.1 What about the indifferentiability?

The sponge indifferentiability proof of [3] assumes the capacity is fixed and does not prove indifferentiability of 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 an indifferentiability theorem in [6, Section 3.1.2] for the case of variable capacity and diversifier values. We refer to that section for a more in-depth explanation.

3.2 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 and of the coding of the bitrate in the padding. The cost of supporting a variable capacity value with a fixed state width is therefore quite limited.

3.3 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 equal to twice the output length for output lengths below $c_{\max}/2$ bits and a capacity equal to $c_{\max}$ bits for higher output lengths.

4 Parameters of the Keccak-f permutation

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* [6, Section 7.2.2]. Here the odd and even bits of each lane are split, and the state of Keccak-f[1600] is represented as 50 words of 32 bits. Rotations are then performed as cyclic shifts on 32-bit words, making them efficient on a 32-bit processor. There is a cost associated to the conversion of the input message into this representation, but this cost remains small compared to the evaluation of the permutation itself (see [6, Section 7.2.2] for the performance penalty). 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 [6, Section 7.2]. 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].

5 Tree hashing

A way to exploit parallelism is to use tree hashing [7]. This technique can exploit SIMD architectures, multiple cores, or both. Like most hash functions, KECCAK can benefit from this technique. We do not propose tree hashing in the specifications because a sound and well-defined tree hashing construction can work above the mode of operation and so using an unmodified instance of KECCAK. The drawbacks of this technique, though, are the larger memory footprint and the extra fixed processing cost, which can be significant for smaller messages.

In the light of two recent papers [10, 7], a sound tree hashing mode can be easily built as an application on top of existing hash functions and does not have to be embedded in the mode of operation. We define in [6, Section 6.4] an example of such a tree hashing application called KECCAKTREE.

Tree hashing can not only benefit from multiple cores, they can also exploit SIMD architectures on a single core. For instance, a specific instance of KECCAKTREE can reach about 9 cycles/byte (single core) on NIST's reference platform using SSE2 instructions [6, Section 7.3.3]. Further improvements may be obtained, in the future, with larger SIMD registers, and of course by moving to a multiple core implementation.

This technique is not useful for short messages, however, as there is a fixed additional cost corresponding to the processing of a couple of extra blocks (the number depending on the chosen parameters). Also, the memory footprint increases with the number of KECCAK-f permutations that can be evaluated in parallel.

On platforms with less parallelism, KECCAKTREE can only partially exploit the parallelism available in the chosen tree structure or can even be implemented sequentially (and is thus not significantly slower than KECCAK itself for long messages). Except for the memory footprint and for short messages, it can be advantageous to use a tree enabling a high level of parallelism and let the target platform organize the computation to take advantage of this parallelism or less.

Finally, it is worth noting that the arbitrarily-long output length of KECCAK comes in handy for tree hashing. Referring to [6, Section 6.4] and [7] for the technical explanations, the intermediate hashing nodes need to produce at least c bits of output, while the four fixed-output-length variants output only $n = c/2$ bits. This is another reason for proposing an arbitrary output length.

6 On the 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.

6.1 Changing the number of rounds

The number of rounds of the KECCAK-f permutations is defined and fixed in [5] 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.

6.2 The safety margin with the nominal number of rounds

As reflected in our estimates for the safety margin of Keccak against different types of attack in [6, Section 5.4], we think Keccak-f has about twice as many rounds as strictly required for Keccak to stand up to its security claim, for any choice of the capacity. The high number of rounds in Keccak-f[1600] is due to our adoption of the hermetic sponge strategy [4] and our wish to keep a safety margin against all distinguishers, irrespective of their strength and applicability to Keccak itself, e.g., see [9].

In September 2009 we have increased the number of rounds from 18 to 24 in Keccak-f[1600]. We took this decision after the publication of a valid but non-threatening 16-round structural distinguisher in [1]. We refer to [9] for a treatment of this.

6.3 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 [13, 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 [13, 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.

The KECCAK Team, February 2010
Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche

References

[1] J.-P. Aumasson and W. Meier, *Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi*, Available online, 2009, http://131002.net/data/papers/AM09.pdf.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Sponge functions*, Ecrypt Hash Workshop 2007, May 2007, also available as public comment to NIST from http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html.

[3] ———, *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] ———, *Cryptographic sponges*, 2009, http://sponge.noekeon.org/.

[5] ———, KECCAK *specifications, version 2*, NIST SHA-3 Submission, September 2009, http://keccak.noekeon.org/.

[6] ———, KECCAK *sponge function family main document*, NIST SHA-3 Submission (updated), September 2009, http://keccak.noekeon.org/.

[7] ———, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210, 2009, http://eprint.iacr.org/.

[8] ———, *Tune* KECCAK *to your requirements*, 2009, http://keccak.noekeon.org/tune.html.

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

[10] Y. Dodis, L. Reyzin, R. Rivest, and E. Shen, *Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6*, Fast Software Encryption (O. Dunkelman, ed.), Lecture Notes in Computer Science, vol. 5665, Springer, 2009, pp. 104–121.

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

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

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

[14] ———, *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.

From: joan daemen [joan.daemen@st.com]
Sent: Friday, June 25, 2010 5:16 AM
To: hash-function@nist.gov
Cc: hash-forum@nist.gov
Subject: OFFICIAL COMMENT: Keccak (Round 2)

Dear all,

we released version 2.1 of the Keccak main document, that besides some restructuring and editorial improvements also brings new content. All modifications are listed in a change log in its appendix. At the same time, we released version 2.1 of KeccakTools, a set of documented C++ classes that can help analyze Keccak-f, bringing some new classes.

Both are available from the Keccak web page:
http://keccak.noekeon.org/Keccak-main-2.1.pdf
http://keccak.noekeon.org/KeccakTools-2.1.zip

Note that the Keccak specification has not changed since the round-2 submission and is still:
http://keccak.noekeon.org/Keccak-specifications-2.pdf

Kind regards,

The Keccak team

From: hash-forum@nist.gov on behalf of Gilles VAN ASSCHE [gilles.vanassche@st.com]
Sent: Friday, November 05, 2010 12:28 PM
To: Multiple recipients of list
Subject: OFFICIAL COMMENT: Keccak (Round 2)

Dear all,

Please note that we recently published a new set of implementations of Keccak, thanks to contributions from our colleague Ronny Van Keer, STMicroelectronics. We focused on implementations suitable for small platforms, such as 32-bit and 8-bit embedded processors. More details can be found on our web page, and more specifically here:
http://keccak.noekeon.org/optimized_2.2.html
http://keccak.noekeon.org/optimized_2.3.html

Among the new implementations, a few of them are meant to be reasonably compact in terms of both code size and memory usage. In terms of RAM specifically, all the operations are done in place, with a limited number of temporary variables. So, they need just a bit more than the 200 bytes necessary to store the state. In addition, we provide an API for giving partial input chunks (such as Init, Update, Final), where we exploit a unique feature of sponge functions and similar: These partial input chunks are XORed directly into the state of Keccak, thereby removing the need of a separate message queue.

According to our internal tests, we expect these new implementations to perform much better than what can be found in the sphlib 2.1 report or in the recent study of Mourad Gouicem. A subset of these new variants and implementations has been submitted to eBASH and to XBX.

Kind regards,

The Keccak team