Orange Labs

Issy-les-Moulineaux, France

SHA-3 Proposal: ECHO

Ryad Benadjila
Olivier Billet
Henri Gilbert
Gilles Macario-Rat
Thomas Peyrin
Matt Robshaw
Yannick Seurin

Version 2.0

July 20, 2010

Overview

This document provides an update to the original description of ECHO that was submitted to NIST. ECHO embodies the goal of reusing—and thereby echoing—as many aspects of the Advanced Encryption Standard (AES) [58] as possible. This is not just in terms of operations, though only AES operations are used in ECHO, but also in terms of simplicity and analysis. ECHO replicates the structure of the AES in several ways and has the following features:

1. The smooth support—using the same implementation—of any hash output of length from 128 to 512 bits.

2. The smooth support—using the same implementation—of both the single-pipe and double-pipe strategy.

   UPDATE: *Some SHA-3 submissions use single-pipe constructions. While we prefer the security provided by a double-pipe construction, we note that* ECHO *can easily be adapted to support a single-pipe construction giving a significant performance improvement at the same time.*

3. Avoiding a key schedule. It is well-known that the design of a "key schedule" or some form of "message mixing" for a hash function is deceptively difficult. This has been the problem at the root of the MD/SHA family and has even been reported as a potential problem for the AES itself [13, 12].

   UPDATE: *Any NIST SHA-3 submission with a key schedule needs to substantiate its resistance to such attacks. It was a deliberate design decision to not have a key schedule in* ECHO *and the opportunity to interfere with a compression function computation once it has begun has been eliminated.*

4. An established design approach with attendant security arguments. This allows a particularly accurate differential security analysis and gives a very significant–and independently analysed–margin for security.

   UPDATE: *Many different sophisticated analysis have been recently conducted on* ECHO. *The best distinguishing attack for the* ECHO *compression function covers 4.5 rounds (out of 8) for the 256-bit hash output (which can be improved to 7 rounds if the salt can be controlled by the attacker) and covers 7 rounds (out of 10) for the 512-bit hash output. Concerning collision resistance, the best free-start collision attack for the* ECHO *compression function covers 3 rounds (out of 8) for the 256-bit hash output and covers 3 rounds (out of 10) for the 512-bit hash output. The best collision attack for the* ECHO *hash function covers 4 rounds (out of 8).*

5. The ability to reuse AES implementation advances, whether these offer improved performance or improved resistance to side-channel analysis. Also ECHO can directly exploit AES-inspired processor developments such as Intel's AES instruction set for Westmere chips [29].

UPDATE: *Benchmarks on the recent AES-NI capable Intel Core i5 CPU give 6.8 cycles/byte for the 256-bit version of* ECHO *and 12.6 cycles/byte for the 512-bit version (resp. 5.8 cycles/byte and 8.4 cycles/byte for the single-pipe construction).*

We note that ECHO is the only SHA-3 candidate to support **both** a double-pipe mode and the possibility to exploit the AES instruction set.

In this updated version of the original submission document, we keep a full description of ECHO and an overview of our design considerations. However we have added some new and improved performance figures for all implementations, and we provide some additional information that will help to provide points of comparison with other submissions.

Contents

Part I
Specifications

The hash function ECHO takes a MESSAGE and SALT as input. The output from ECHO can be of any length from 128 to 512 bits. However we fix our attention and our description on the four must-satisfy [63] values of 224, 256, 384, and 512 bits. Similarly, while ECHO has the flexibility to take a message up to $2^{128} - 1$ bits long, our primary focus is on a version of ECHO that hashes messages up to $2^{64} - 1$ bits in length. The SALT is 128 bits long and if for some reason it is unneeded or left unspecified, then it takes the all-zero value by default.

ECHO consists of the serial application[1] of a *compression function* and follows the well-understood Merkle-Damgård paradigm [20, 55]. At the same time we avoid certain deficiencies [21, 37, 43, 42] by carrying a large state from one iteration to the next and by adopting features from the HAIFA model [8, 9]. These features lend additional functionality to the basic design.

As is well-known, a compression function in the Merkle-Damgård paradigm updates the value of a *chaining variable* under the action of a fixed-size block of message and (optionally) some other inputs. The specifications of ECHO will be divided into the following parts:

- Section 1 establishes the notation and conventions.
- Section 2 describes how the compression function in ECHO is used to hash a message of arbitrary length.
- Section 3 defines the compression function for hash outputs between 128 and 256 bits in length.
- Section 4 defines the compression function for hash outputs between 257 and 512 bits in length.

[1] However the design offers significant opportunities for internal parallelism.

1 Notation and Conventions

The basic computational unit in ECHO is 128 bits long. However there is an underlying byte-oriented structure which gives a flexible implementation profile. A padding rule (see Section 2.2) will be applied to the message $\mathcal{M}$ input to ECHO and this guarantees that the padded message $\mathcal{M}'$ has a length n that is a multiple of 128. A padded message can therefore be represented as a bitstring

$$b_0\, b_1\, b_2\, \ldots\, b_{n-2}\, b_{n-1}$$

or equally, as a sequence of $s = \frac{n}{8}$ bytes (where we use $\|$ to denote concatenation)

$$\begin{array}{rcl} B_0 & = & b_0\|b_1\|\ldots\|b_7 \\ B_1 & = & b_8\|b_9\|\ldots\|b_{15} \\ & \vdots & \\ B_{s-1} & = & b_{n-8}\|b_{n-7}\|\ldots\|b_{n-1} \end{array}$$

ECHO is built around the AES and there will be an interplay between a sequence of 128 bits and their conceptual arrangement in a 4×4 array of bytes. Throughout we use the same convention as the AES [58] and we have the following packing of bytes from a word into an array:

$$B_0\|B_1\|\ldots\|B_{15} \longrightarrow \begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_1 & B_5 & B_9 & B_{13} \\ \hline B_2 & B_6 & B_{10} & B_{14} \\ \hline B_3 & B_7 & B_{11} & B_{15} \\ \hline \end{array}$$

Equally an input string can be viewed as a sequence of $r = \frac{n}{128}$ 128-bit words, and we will denote the packing of bytes into words as

$$\begin{array}{rcl@{\qquad}rcl} w_0 & = & B_0\|B_1\|\ldots\|B_{15} & = & b_0\|b_1\|\ldots\|b_{127} \\ w_1 & = & B_{16}\|B_{17}\|\ldots\|B_{31} & = & b_{128}\|b_{129}\|\ldots\|b_{255} \\ & \vdots & & \vdots & & \\ w_{r-1} & = & B_{s-16}\|B_{s-15}\|\ldots\|B_{s-1} & = & b_{n-128}\|b_{n-127}\|\ldots\|b_{n-1} \end{array}$$

In ECHO the compression function will operate on sixteen 128-bit strings which can be packed into a 4×4 array in a similar way to the AES:

$$w_0\|w_1\|\ldots\|w_{15} \longrightarrow \begin{array}{|c|c|c|c|} \hline w_0 & w_4 & w_8 & w_{12} \\ \hline w_1 & w_5 & w_9 & w_{13} \\ \hline w_2 & w_6 & w_{10} & w_{14} \\ \hline w_3 & w_7 & w_{11} & w_{15} \\ \hline \end{array}$$

The bit-string representation of an integer value will have the least significant byte to the left and, within a byte, the most significant bit to the left.

2 Domain Extension

Depending on the length of the hash output, ECHO will use one of two compression functions; COMPRESS_{512} or COMPRESS_{1024}. The subscript refers to the length of the **chaining variable**, to be denoted CSIZE, and at iteration i both compression functions take four inputs:

1. The current value of the chaining variable, V_{i-1}, which is of length CSIZE.
2. The current message block being processed, M_i. The length is MSIZE bits where MSIZE = 2048 − CSIZE.
3. The total number of unpadded message bits hashed at the end of this iteration, $\mathcal{C}_i$.
4. The SALT.

For a hash output of length up to 256 bits COMPRESS_{512} will be used. For hash outputs of length between 257 and 512 bits we use COMPRESS_{1024}. Outputs less than 256 or 512 bits in length, respectively, will be obtained by truncation (see Sections 3.5 and 4.1). The initial values to the chaining variable are denoted V_0 (see Section 2.1) and at iteration i, for $128 \leq \text{HSIZE} \leq 256$, we will compute

$$V_i = \text{COMPRESS}_{512}(V_{i-1}, M_i, \mathcal{C}_i, \text{SALT})$$

while for $257 \leq \text{HSIZE} \leq 512$ we will compute

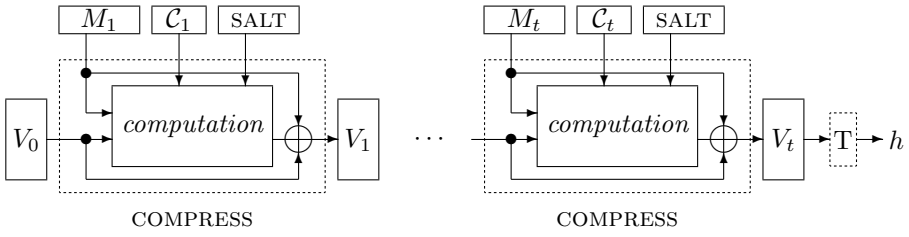
$$V_i = \text{COMPRESS}_{1024}(V_{i-1}, M_i, \mathcal{C}_i, \text{SALT}).$$

As we will see, the two compression functions are almost identical.

While ECHO has the design flexibility to use a 128-bit counter $\mathcal{C}_i$ (see Section 3.1), practical applications requiring such large counters are difficult to envisage. So the primary version of ECHO uses a 64-bit counter $\mathcal{C}_i$ and hashes messages up to $2^{64} - 1$ bits in length. The lengths of the compression function inputs and outputs for the must-satisfy values required by NIST [63] can therefore be summarised as:

hash length (HSIZE)	*uses compression function*	*chaining variable* (CSIZE)	*message block* (MSIZE)	*counter length*	*salt length*
224	COMPRESS_{512}	512	1536	64 or 128	128
256	COMPRESS_{512}	512	1536	64 or 128	128
384	COMPRESS_{1024}	1024	1024	64 or 128	128
512	COMPRESS_{1024}	1024	1024	64 or 128	128

The chained iteration of the compression function over t iterations is illustrated below. Details of the feedforward and exclusive-or will be given later (see Sections 3.4 and 4). As previously mentioned, the initial value to the chaining variable V_0 is defined (see Section 2.1) while T denotes optional truncation (see Sections 3.5 and 4.1).



2.1 Initialisation

At the start of hashing the counter $\mathcal{C}$ is set to $\mathcal{C}_0 = 0$. This counter is used to count the number of message bits being hashed. The initial value of the chaining variable is set so that each word of the chaining variable is the 128-bit encoding of the intended hash output size. For those hash function outputs that use COMPRESS_{512}, namely hash outputs of size $128 \leq \text{HSIZE} \leq 256$, the chaining variable consists of four 128-bit strings $V_0 = (v_0^0, v_0^1, v_0^2, v_0^3)$. For the two NIST must-satisfy values in this range the initial values are, for $0 \leq i \leq 3$,

$$v_0^i = \begin{cases} \texttt{E0000000 00000000 00000000 00000000} & \text{for HSIZE} = 224 \\ \texttt{00010000 00000000 00000000 00000000} & \text{for HSIZE} = 256 \end{cases}$$

For those hash function outputs that use COMPRESS_{1024}, namely hash outputs of size $257 \leq \text{HSIZE} \leq 512$, the chaining variable consists of eight 128-bit strings $V_0 = (v_0^0, v_0^1, v_0^2, v_0^3, v_0^4, v_0^5, v_0^6, v_0^7)$ and for the two NIST must-satisfy values in this range the initial values are, for $0 \leq i \leq 7$,

$$v_0^i = \begin{cases} \texttt{80010000 00000000 00000000 00000000} & \text{for HSIZE} = 384 \\ \texttt{00020000 00000000 00000000 00000000} & \text{for HSIZE} = 512 \end{cases}$$

2.2 Message Padding

Padding of an input message $\mathcal{M}$ is always performed. The result is a padded message $\mathcal{M}'$ that has a length which is a multiple of MSIZE. Assuming that the message to be hashed is of length L bits, then padding is performed by appending the following quantities in the stated order:

1. A single "1" bit is added to the end of the message $\mathcal{M}$.
2. Append x (possibly none) "0" bits where

$$x = \text{MSIZE} - ((L + 144) \mod \text{MSIZE}) - 1.$$

3. A 16-bit binary representation of HSIZE is added next. For the purposes of this document, the possible values as bit strings in hexadecimal notation are `E000`, `0001`, `8001`, and `0002`.

4. Finally the 128-bit representation of the length L is included.

The result is a padded message of the following form (with bit lengths indicated):

$$\mathcal{M}' = \overbrace{\overbrace{\mathcal{M}}^{L} \parallel 1 \parallel \overbrace{0 \cdots\cdots 0}^{variable} \parallel \overbrace{\textsc{Hsize}}^{16} \parallel \overbrace{L}^{128}}^{n}$$

Some implications of this style of padding are detailed in Section 3.1.

3 Compression for $128 \leq$ HSIZE ≤ 256

The padded message $\mathcal{M}'$ is divided into t message blocks $M_1 \ \ldots \ M_t$, each MSIZE = 1536 bits long. These are processed in turn using the compression function $\textsc{compress}_{512}$. The $t + 1$ values of the chaining variable that will be generated during the hash computation are denoted V_i, for $0 \leq i \leq t$. The initial value V_0 is given in Section 2.1 and, at iteration i, we compute

$$V_i = \textsc{compress}_{512}(V_{i-1}, M_i, \mathcal{C}_i, \textsc{salt})$$

where $\mathcal{C}_i$ equals the number of message bits that will be processed by the end of the iteration. Each message block M_i can be split into 128-bit strings:

$$M_i = m_i^0 \parallel m_i^1 \parallel m_i^2 \parallel m_i^3 \parallel m_i^4 \parallel m_i^5 \parallel m_i^6 \parallel m_i^7 \parallel m_i^8 \parallel m_i^9 \parallel m_i^{10} \parallel m_i^{11} \ .$$

The chaining variable V_{i-1} is viewed as a sequence of four 128-bit values:

$$V_{i-1} = v_{i-1}^0 \parallel v_{i-1}^1 \parallel v_{i-1}^2 \parallel v_{i-1}^3 \ .$$

The mixing of the chaining variable and message during compression requires a series of operations on the state S, and S can be viewed as a 4×4 array:

w_0	w_4	w_8	w_{12}
w_1	w_5	w_9	w_{13}
w_2	w_6	w_{10}	w_{14}
w_3	w_7	w_{11}	w_{15}

At the start of the i^{th} iteration of the compression function, the chaining variable and message are loaded as follows:

v_{i-1}^0	m_i^0	m_i^4	m_i^8
v_{i-1}^1	m_i^1	m_i^5	m_i^9
v_{i-1}^2	m_i^2	m_i^6	m_i^{10}
v_{i-1}^3	m_i^3	m_i^7	m_i^{11}

The other inputs to the compression function, apart from the chaining variable and the message block, are the SALT and counter $\mathcal{C}_i$. The SALT will be used *as is* during compression while $\mathcal{C}_i$ is used to provide the initial value to an internal counter κ. The computation in COMPRESS_{512} runs over eight steps of BIG.ROUND which, in turn, consists of the sequential application of the following three functions:

BIG.SUBWORDS(S, SALT, κ)
BIG.SHIFTROWS(S)
BIG.MIXCOLUMNS(S)

This is the same form as one round of the AES and the similarities are even closer if we consider each operation in turn.

3.1 BIG.SUBWORDS(S, SALT, κ)

This operation is, in effect, an S-box look-up. The S-box is a substitution on 128-bit words and directly uses two AES rounds *without change*. Given a 128-bit word w, we can denote the action of one AES round function on w, using subkey k, by

$$w' = \text{AES}(w, k)$$

Here the AES round function is the full round that consists of `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey` in this order (using the FIPS 197 terminology in [58]).

The "subkeys" for the AES rounds will be given by the SALT and the internal counter κ. The internal counter κ is initialised with the value of $\mathcal{C}_i$. The counters κ and $\mathcal{C}_i$ are the same length[2] and the internal counter κ will be incremented during the compression computation. $\mathcal{C}_i$ is not incremented during compression, but instead $\mathcal{C}_i$ is used to count the total number of message bits hashed by the end of the iteration. Note that this is the number of *message* bits, and **not** the number of *padded message* bits. Thus $\mathcal{C}_i$ will typically have a value that is a strict multiple of MSIZE. The only exceptions to this concern the penultimate and final iterations where the compression function might process both padding bits and message bits. If both message and padding bits are processed in the penultimate or final compression function then $\mathcal{C}_{t-1}$ or $\mathcal{C}_t$, respectively, will take the value L. If the final iteration **only** processes padding bits and no message

[2] As stated before, a 128-bit counter is naturally supported (see later in this section).

bits (which can happen for example when L is a multiple of MSIZE) then we set $\mathcal{C}_t$ to the value 0.

In the basic version, the internal counter κ is 64 bits long and the two round subkeys k_1 and k_2 are derived as:

$$k_1 = \kappa \parallel \overbrace{0 \cdots 0}^{64} \quad \text{and} \quad k_2 = \text{SALT}.$$

The operation BIG.SUBWORDS(S, SALT, κ) can now be described and each word w_i of the state S is updated to give the word w'_i as

$$\begin{aligned} w'_0 &= \text{AES}(\text{AES}(w_0, k_1), k_2), \text{ and then increment } \kappa \text{ by one} \\ w'_1 &= \text{AES}(\text{AES}(w_1, k_1), k_2), \text{ and then increment } \kappa \text{ by one} \\ &\vdots \\ w'_{15} &= \text{AES}(\text{AES}(w_{15}, k_1), k_2), \text{and then increment } \kappa \text{ by one} \end{aligned}$$

Note that after every computation of some w'_i we increment the counter κ by one (or modulo 2^{128} for the longer counter below). Thus the "subkey" k_1 that is used in the AES operation changes from one step to the next. Since the offset we apply to κ for any given word of the state can be readily computed, the operation BIG.SUBWORDS can be efficiently parallelised. Note that we continue to increment κ throughout all the rounds, and so the second BIG.ROUND of computation will begin with $\kappa = \mathcal{C}_i + 16$.

Applications that require exceptionally long message inputs.

In practical applications, it is hard to conceive of situations where ECHO would be used with messages having a length close to 2^{64} bits. Consequently a 64-bit counter is almost certainly going to be more than adequate. However for those that need it, the design naturally supports exceptionally long messages up to $2^{128} - 1$ bits. For this the counters $\mathcal{C}_i$ and κ are implemented over 128 bits. The two subkeys k_1 and k_2 used in the AES rounds will then be defined as:

$$k_1 = \kappa \quad \text{and} \quad k_2 = \text{SALT}.$$

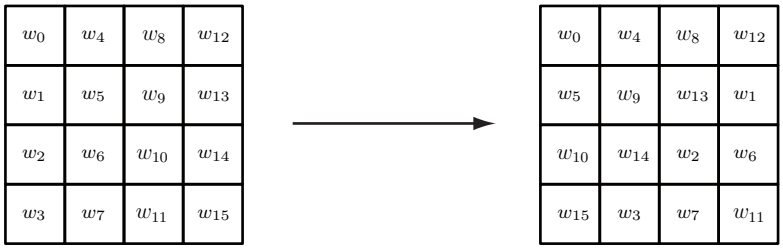
To ensure compatibility between versions of ECHO using different length counters, we require that a 128-bit counter must be used when the message length is $2^{64} - 160$ or greater.

3.2 BIG.SHIFTROWS(S)

The BIG.SHIFTROWS operation is an exact analogue of the `ShiftRows` operation in the AES. The 4×4 array that holds the state S is permuted by shifting rows of 128-bit words in exactly the same fashion as the byte-array is permuted in the AES. We can therefore describe the action of BIG.SHIFTROWS on the words $w_0, \ldots, w_{15}$ as

$$w'_{i+4j} = w_{i+4((i+j) \bmod 4)} \text{ for } 0 \le i, j \le 3 \ .$$

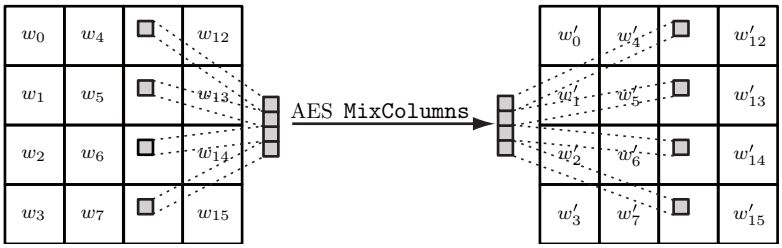
This can be described pictorially as



3.3 BIG.MixColumns(S)

This operation is an exact analogue of the `MixColumns` operation in the AES. The `MixColumns` operation performs an MDS-based mixing of four bytes, and in the AES it processes each of the four columns of the state table in turn.

This process is extended in the obvious way to ECHO where we view the four 128-bit columns of the state S as 64 columns that are each a byte wide. Then we apply the AES `MixColumns` operation to each of these columns in S.



More formally, consider four 128-bit entries w_i, ..., w_{i+3} for $i \in \{0, 4, 8, 12\}$ that form a column in S. Writing these as byte strings we have

$$\begin{aligned} w_i &= (B_{16i}, B_{16i+1}, \ldots, B_{16i+15}) \\ w_{i+1} &= (B_{16i+16}, B_{16i+17}, \ldots, B_{16i+31}) \\ w_{i+2} &= (B_{16i+32}, B_{16i+33}, \ldots, B_{16i+47}) \\ w_{i+3} &= (B_{16i+48}, B_{16i+49}, \ldots, B_{16i+63}) \end{aligned}$$

Using FIPS-197 notation [58] we compute, for $i \in \{0, 4, 8, 12\}$ and $0 \le j \le 15$,

$$\begin{pmatrix} B'_{16i+j} \\ B'_{16i+16+j} \\ B'_{16i+32+j} \\ B'_{16i+48+j} \end{pmatrix} = \begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix} \cdot \begin{pmatrix} B_{16i+j} \\ B_{16i+16+j} \\ B_{16i+32+j} \\ B_{16i+48+j} \end{pmatrix}$$

This is merely the AES `MixColumns` operation where the field arithmetic is defined by the *Rijndael polynomial* $x^8 + x^4 + x^3 + x + 1$; more details are given in [58].

3.4 Finalising Compression for $128 \leq$ HSIZE ≤ 256

The complete compression function COMPRESS$_{512}$ can be described in terms of the constituent operations:

```
repeat
        BIG.SUBWORDS(S, SALT, κ)
        BIG.SHIFTROWS(S)
        BIG.MIXCOLUMNS(S)
eight times
BIG.FINAL
```

The operation BIG.FINAL is used to derive the output value of the chaining variable. It incorporates a feedforward of the inputs and if we denote the final values in S as $w_0, \ldots, w_{15}$ then BIG.FINAL (for COMPRESS$_{512}$) is described as:

$$\begin{aligned} v_i^0 &= v_{i-1}^0 \oplus m_i^0 \oplus m_i^4 \oplus m_i^8 \oplus w_0 \oplus w_4 \oplus w_8 \oplus w_{12} \\ v_i^1 &= v_{i-1}^1 \oplus m_i^1 \oplus m_i^5 \oplus m_i^9 \oplus w_1 \oplus w_5 \oplus w_9 \oplus w_{13} \\ v_i^2 &= v_{i-1}^2 \oplus m_i^2 \oplus m_i^6 \oplus m_i^{10} \oplus w_2 \oplus w_6 \oplus w_{10} \oplus w_{14} \\ v_i^3 &= v_{i-1}^3 \oplus m_i^3 \oplus m_i^7 \oplus m_i^{11} \oplus w_3 \oplus w_7 \oplus w_{11} \oplus w_{15} \end{aligned}$$

3.5 The Hash Output for $128 \leq$ HSIZE ≤ 256

The final value of the chaining variable is viewed as a 512-bit string

$$v_t^0 \parallel v_t^1 \parallel v_t^2 \parallel v_t^3 \ .$$

To provide a hash output h of HSIZE bits we output the HSIZE leftmost bits. Thus, for instance, for HSIZE $= 256$ the output is given by

$$h = v_t^0 \parallel v_t^1 \ .$$

The hash outputs for other values of HSIZE can be easily derived.

4 Compression for $257 \leq$ HSIZE ≤ 512

If a hash output larger than 256 bits is required, we use the compression function COMPRESS$_{1024}$. This is identical to COMPRESS$_{512}$ except:

1. The padded[3] message $\mathcal{M}'$ is divided into t message blocks $M_1 \ldots M_t$, each of which is 1024 bits long.

[3] Recall from Section 2.2 that padding includes an encoding of the hash output length.

2. The chaining variable consists of eight 128-bit words $V_i = v_i^0 \dots v_i^7$ and is initialised as described in Section 2.1.

3. At the start of compression, the chaining variable and message block are loaded into the 4×4 state array as:

v_{i-1}^0	v_{i-1}^4	m_i^0	m_i^4
v_{i-1}^1	v_{i-1}^5	m_i^1	m_i^5
v_{i-1}^2	v_{i-1}^6	m_i^2	m_i^6
v_{i-1}^3	v_{i-1}^7	m_i^3	m_i^7

4. Computation consists of ten iterations of BIG.ROUND (instead of the eight iterations used in COMPRESS_{512}).

5. If we denote the final values in S as $w_0, \dots, w_{15}$, the operation BIG.FINAL to derive the new chaining variable is given by:

$$\begin{aligned} v_i^0 &= v_{i-1}^0 \oplus m_i^0 \oplus w_0 \oplus w_8 & \qquad v_i^4 &= v_{i-1}^4 \oplus m_i^4 \oplus w_4 \oplus w_{12} \\ v_i^1 &= v_{i-1}^1 \oplus m_i^1 \oplus w_1 \oplus w_9 & \qquad v_i^5 &= v_{i-1}^5 \oplus m_i^5 \oplus w_5 \oplus w_{13} \\ v_i^2 &= v_{i-1}^2 \oplus m_i^2 \oplus w_2 \oplus w_{10} & \qquad v_i^6 &= v_{i-1}^6 \oplus m_i^6 \oplus w_6 \oplus w_{14} \\ v_i^3 &= v_{i-1}^3 \oplus m_i^3 \oplus w_3 \oplus w_{11} & \qquad v_i^7 &= v_{i-1}^7 \oplus m_i^7 \oplus w_7 \oplus w_{15} \end{aligned}$$

4.1 The Hash Output for $257 \leq \text{HSIZE} \leq 512$

The final value of the chaining variable is viewed as a 1024-bit string:

$$v_t^0 \parallel v_t^1 \parallel v_t^2 \parallel v_t^3 \parallel v_t^4 \parallel v_t^5 \parallel v_t^6 \parallel v_t^7.$$

To provide a hash output h of HSIZE bits we output the HSIZE leftmost bits. Thus, for instance, for $\text{HSIZE} = 384$ we have

$$h = v_t^0 \parallel v_t^1 \parallel v_t^2$$

while for $\text{HSIZE} = 512$ the output is given by

$$h = v_t^0 \parallel v_t^1 \parallel v_t^2 \parallel v_t^3.$$

The hash outputs for other values of HSIZE can be easily derived.

5 Supporting Single-Pipe Applications

For those preferring a single-pipe mode, it is straightforward to adapt ECHO to accommodate this. We will refer to this version of ECHO as ECHO-SP.

5.1 Compression for $128 \leq$ HSIZE ≤ 256 with Single-Pipe

In this case the input message block becomes

$$M_i = m_i^0 \parallel m_i^1 \parallel m_i^2 \parallel m_i^3 \parallel m_i^4 \parallel m_i^5 \parallel m_i^6 \parallel m_i^7 \parallel m_i^8 \parallel m_i^9 \parallel m_i^{10} \parallel m_i^{11} \parallel m_i^{12} \parallel m_i^{13}$$

the chaining variable becomes

$$V_{i-1} = v_{i-1}^0 \parallel v_{i-1}^1$$

and the chaining variable and message are loaded as:

v_{i-1}^0	m_i^2	m_i^6	m_i^{10}
v_{i-1}^1	m_i^3	m_i^7	m_i^{11}
m_i^0	m_i^4	m_i^8	m_i^{12}
m_i^1	m_i^5	m_i^9	m_i^{13}

The rest of compression and message padding continues as before but with CSIZE set to 256. The output chaining variable is given by

$$v_i^0 \parallel v_i^1$$

with $v_i^0 = x_i^0 \oplus x_i^2$ and $v_i^1 = x_i^1 \oplus x_i^3$ where

$$\begin{aligned}
x_i^0 &= v_{i-1}^0 \oplus m_i^2 \oplus m_i^6 \oplus m_i^{10} \oplus w_0 \oplus w_4 \oplus w_8 \oplus w_{12} \\
x_i^1 &= v_{i-1}^1 \oplus m_i^3 \oplus m_i^7 \oplus m_i^{11} \oplus w_1 \oplus w_5 \oplus w_9 \oplus w_{13} \\
x_i^2 &= m_i^0 \oplus m_i^4 \oplus m_i^8 \oplus m_i^{12} \oplus w_2 \oplus w_6 \oplus w_{10} \oplus w_{14} \\
x_i^3 &= m_i^1 \oplus m_i^5 \oplus m_i^9 \oplus m_i^{13} \oplus w_3 \oplus w_7 \oplus w_{11} \oplus w_{15} \ .
\end{aligned}$$

5.2 Compression for $257 \leq$ HSIZE ≤ 512 with Single-Pipe

In this case the input message block is given by

$$M_i = m_i^0 \parallel m_i^1 \parallel m_i^2 \parallel m_i^3 \parallel m_i^4 \parallel m_i^5 \parallel m_i^6 \parallel m_i^7 \parallel m_i^8 \parallel m_i^9 \parallel m_i^{10} \parallel m_i^{11}$$

the chaining variable is given by

$$V_{i-1} = v_{i-1}^0 \parallel v_{i-1}^1 \parallel v_{i-1}^2 \parallel v_{i-1}^3 \ .$$

The chaining variable and message are loaded as:

v_{i-1}^0	m_i^0	m_i^4	m_i^8
v_{i-1}^1	m_i^1	m_i^5	m_i^9
v_{i-1}^2	m_i^2	m_i^6	m_i^{10}
v_{i-1}^3	m_i^3	m_i^7	m_i^{11}

The rest of compression and message padding continues as before but with CSIZE set to 512. The output chaining variable is given by

$$v_i^0 \parallel v_i^1 \parallel v_i^2 \parallel v_i^3$$

where

$$\begin{aligned}
v_i^0 &= v_{i-1}^0 \oplus m_i^0 \oplus m_i^4 \oplus m_i^8 \oplus w_0 \oplus w_4 \oplus w_8 \oplus w_{12} \\
v_i^1 &= v_{i-1}^1 \oplus m_i^1 \oplus m_i^5 \oplus m_i^9 \oplus w_1 \oplus w_5 \oplus w_9 \oplus w_{13} \\
v_i^2 &= v_{i-1}^2 \oplus m_i^2 \oplus m_i^6 \oplus m_i^{10} \oplus w_2 \oplus w_6 \oplus w_{10} \oplus w_{14} \\
v_i^3 &= v_{i-1}^3 \oplus m_i^3 \oplus m_i^7 \oplus m_i^{11} \oplus w_3 \oplus w_7 \oplus w_{11} \oplus w_{15} \ .
\end{aligned}$$

Part II
Design Rationale and Implementation

In this second part we highlight some of our design decisions and we provide an overview of the anticipated performance profile of ECHO.

Summary

- ECHO has a simple design that stays close to the AES and facilitates a clear and in-depth security assessment. This is corroborated by the important amount of cryptanalysis published so far on ECHO.
- ECHO can directly leverage advances in the implementation of the AES. This gives performance benefits and improved know-how with regards to side-channel analysis.
- We provide estimates for the software performance of ECHO with a wide range of estimates going down to 6 cycles per byte using the new Intel AES instruction set. We also cover implementation estimates for smart cards and the performance of ECHO in hardware.
- ECHO supports randomized hashing via the SALT and naturally supports the HMAC construction of FIPS 198-1 [60].
- There are no pre-computation or table-generation start-up costs for ECHO.
- The only S-box in ECHO is the AES S-box from FIPS 197 [58]. There are no constants or other arbitrarily-chosen parameters in ECHO.
- While we believe that the security offered by a double-pipe construction is required for SHA-3, ECHO smoothly supports the single-pipe construction for those that prefer to trade more performance for a little less security.
- The hash output of ECHO is not restricted to the four must-satisfy values of 224, 256, 384, and 512 bits.
- ECHO is a flexible and suitable replacement for SHA-2 in all applications including those covered by NIST FIPS 186-2 [59], FIPS 198-1 [60], SP 800-56A [61], and SP 800-90 [62].

6 Design Rationale

Throughout the design period we have attempted to build on trusted and known components. Compared to our knowledge of block ciphers at the start of the AES process, the research-level understanding of hash functions is much less well-developed today. Over the past years we have seen some radically different new design approaches, some of which feature among submissions to the SHA-3 process.

However, instead of embracing a new and relatively untested approach we decided to return to block ciphers and to use the AES as a foundation for our security. After DES [57] the AES is perhaps the most scrutinised block cipher and it makes an interesting point of departure. In addition, new advances in processor design (see Section 7.3) provide the opportunities for ever faster (and more secure) implementation. We can therefore take this into account and design a hash function with clear performance benefits in the near future. Here we highlight some of the design features.

Simplicity.

> We have attempted to make a simple, aesthetically pleasing, and somewhat provocative design. The main computation in ECHO is round-based and so reduced-round versions that might be of interest to cryptanalysts are easily defined. We encourage independent analysis of our proposal.

Security.

> A simple design encourages analysis. With regards to the anticipated security margins, we have adopted a conservative position. Not only do we reuse trusted components from the AES, but we use a double-length chaining variable to carry additional state through the hash computation. While this has had some impact on the headline performance of ECHO, we believe that the driving factor behind the adoption of SHA-3 should remain security. For those that prefer a little more performance and are happy to move to a single-length chaining variable, ECHO smoothly accommodates this change.

Using the AES.

> We use the 128-bit AES round computation as a building block in ECHO. This allows us to directly exploit any future advances in AES implementation or processor-support. We also replicate the AES "square"-like structure so as to provide a simple framework for analysis. However, we decided not to use the entirety of the AES as the basis of a construction. Analysis [68] has shown that one would need at least five calls to the single-key AES (when using standard design techniques) to achieve a 256-bit hash output and at least eight calls to achieve 512-bit hash outputs. By reusing internal rounds of the AES—rather than the entirety of the AES block cipher—we are able to get much better performance.

Avoiding a Key Schedule.

The classical approach to many hash functions—especially the MD/SHA family—is to base the hash computation on a rather unusual block cipher with a large input/output block and a large key. However, at the same time, the "key schedule" or "message mixing" tends to be relatively simple and this can allow differentials during the hash computation to be tamed and controlled by a sophisticated adversary.

To avoid this some adopt a more complex key schedule. The alternative, however, is to remove the opportunity to manipulate or interfere with a computation once it has begun. This is our approach and as well as removing this opportunity for the cryptanalyst, we avoid the need to design an effective key schedule with the attendant additional cost to run-time performance.

Being Input Neutral.

In the hash function literature—especially in the MD/SHA family—the message and the chaining variable that accumulates the results of the hashing process, are used in different ways. Given the prominence given to academic attacks that exploit manipulations of the chaining variable, we are not sure that such a distinction is entirely appropriate. Instead both inputs are treated in the same way in ECHO and this has the pleasing side-effect of making design and analysis more straightforward.

Smooth Parameter Handling.

The SHA-3 requirements are exacting in several regards, but the requirement to support four distinct hash lengths is one of the more challenging. In ECHO we wanted a design that handled such a requirement as smoothly as possible, and our design effectively allows the same implementation to be reused for all required values.

Performance, Future Performance, and Parallelism.

In any design there are performance trade-offs. Most often it is security levels that are traded against performance and, as noted earlier, we have favoured security. But there can be other compromises, such as how to deal with the massive range of deployed, and future, processors. We decided to look more to the future. Without neglecting the needs of current deployments, we aim to capture the benefits of future design trends since these will define the high-performance environments over the next 30 years. In 2014 or beyond, applications with the highest performance demands will need to benefit from the latest instruction-sets and internal parallelism. This is where we have targeted our design.

7 Performance Overview

7.1 Rule-of-Thumb

Since ECHO builds on the AES it is illustrative to make a *back-of-the-envelope* calculation of their comparative performances. When using COMPRESS_{512} each iteration of BIG.SUBWORDS uses two AES round operations. Over the totality of the compression function this amounts to 256 AES rounds with which we hash 192 bytes. This gives a hashing rate of $\frac{192}{256} = 0.75$ bytes per AES-round. When using COMPRESS_{1024} this drops to $\frac{128}{320} = 0.4$ bytes per AES-round. Turning to the AES we observe that 16 bytes are encrypted over 10 rounds for AES-128 and over 14 rounds for[4] AES-256. The encryption rates are therefore $\frac{16}{10} = 1.6$ or $\frac{16}{14} \approx 1.1$ bytes per AES-round respectively.

Given that we have ignored both the overheads in ECHO and the key schedule demands in the AES, there is a lot of leeway in this guidance. However we feel that a reasonable shorthand to estimate the software performance of ECHO is to assume that it will be half the speed of AES-128 for hash lengths up to 256 bits, and around one quarter the speed of the AES-128 for hash outputs between 257 and 512 bits. Implementation results confirm the validity of this rough measure.

With this kind of performance it is not surprising that ECHO is not the fastest SHA-3 submission on the NIST reference platform. However ECHO has a speed that we are very comfortable with, particularly when we consider the additional positive design factors in ECHO. These include the ability to leverage advances in the implementation of the AES and to deliver a simple design with an advanced security analysis. Moreover, unlike many SHA-3 candidates, ECHO performances are very acceptable on legacy processors and we also prefer to attain a truly dramatic performance on modern processors, as is discussed below (see Section 7.3).

7.2 Software Implementations

Our implementations give the following performance figures under compilers for the NIST reference platform (Core 2 Duo 6600, 2.4 GHz, 2 GB of RAM) running both Vista and Linux Debian etch amd64. All figures are given in cycles per byte and the accuracy of our *rule-of-thumb* that the performance for $257 \leq \text{HSIZE} \leq 512$ is roughly half that obtained for $128 \leq \text{HSIZE} \leq 256$ can be observed.

[4] Arguably AES-256 would offer a more comparable security level.

		128 ≤ HSIZE ≤ 256	
compiler	*operating system*	ANSI C *32-bit*	ANSI C *64-bit*
MS 2005 (cl 14.0, Option /Ox)	Vista	51	38
MS 2008 (cl 15.0, Option /Ox)	Vista	50	37
Intel (icl 10.1, Option /fast)	Vista	45	35
Intel (icc 10.1, Option -fast)	Linux	45	35
		257 ≤ HSIZE ≤ 512	
compiler	*operating system*	ANSI C *32-bit*	ANSI C *64-bit*
MS 2005 (cl 14.0, Option /Ox)	Vista	96	71
MS 2008 (cl 15.0, Option /Ox)	Vista	93	69
Intel (icl 10.1, Option /fast)	Vista	83	66
Intel (icc 10.1, Option -fast)	Linux	83	66

These figures suggest the following software performance in ANSI C on the NIST reference platform:

- $63 - 69$ MBytes/sec. (64-bit implementation) and $47 - 53$ MBytes/sec. (32-bit implementation) for ECHO with hash outputs 224 and 256 bits.
- $34 - 36$ MBytes/sec. (64-bit implementation) and $25 - 29$ MBytes/sec. (32-bit implementation) for ECHO with hash outputs 384 and 512 bits.

No initial computations are required to build up internal tables.

We note that code aimed at any particular platform, *e.g.* the NIST reference platform, will give very different results when run elsewhere and much will depend on the amount of cache memory available. A range of implementations using different techniques is underway, but assembly implementations are more likely to give the best guide to the final performance we might expect. With this in mind, the current results and our anticipated results for the NIST reference platform are given below. All figures are given in cycles per byte. For comparison with SHA-3 submissions that adopt the single-pipe construction, we note that ECHO smoothly supports this option and offers a performance gain. These figures are given below.

	128 ≤ HSIZE ≤ 256		257 ≤ HSIZE ≤ 512	
	64-bit	*32-bit*	*64-bit*	*32-bit*
assembly (ECHO)	28.3	32.5	50.3	59.7
assembly (ECHO-SP)	24.4	26.7	35.3	40.7

A more extensive set of figures for optimized implementations for a wide range of platforms is given in Appendix A.

7.3 Future Directions: AES Accelerators and Intel

The news that Intel would include an AES instructions set in a range of products was warmly welcomed by the cryptographic community. Not only does this

provide additional resistance to a range of side-channel attacks [4, 65], but it also provides direct performance benefits.

ECHO was designed with such progress in mind and our goal was that ECHO should directly benefit from any efforts to accelerate AES operations. Our design uses the AES round untouched and these rounds can be performed directly using the Intel AES instructions set. This gives dramatic performance figures.

The chips with the new instructions set are now available [35]. During the first round of the SHA-3 process this was not the case yet, and we had to use a new methodology to estimate what would be the performance of ECHO with such instructions (see [3] for more details). It turns out that the initial estimates were accurate as our benchmarks on the Intel Core i5 CPU confirm.

Full implementations using the new instructions set are available on the web page of ECHO. We have the following performance measurements in cycles/bytes where 224- and 384-bit outputs are obtained by truncating 256- and 512-bit outputs:

	$128 \leq$ HSIZE ≤ 256		$257 \leq$ HSIZE ≤ 512	
Using AES-NI	*64-bit*	*32-bit*	*64-bit*	*32-bit*
ECHO	6.8	8.3	12.6	15.3
ECHO-SP	5.8	7.1	8.4	10.1

Using the new `aesenc` instruction we can compute a full AES round using the following instruction with syntax:

```
aesenc xmm, xmm/mem128
```

where `xmm` is a 128-bit register and `mem128` is a 128-bit word. The first operand holds the AES state, the second holds the round key. The result of the AES encryption is stored in the first operand. Thanks to the fully pipelined architecture of future AES-based units, it is possible to launch one `aesenc` per cycle provided there is no data-dependency breaking the parallelism. The design of ECHO uses 32 AES encryptions per round of the compression function, and these can be divided into a first set of 16 independent encryptions (with the counter κ as a key), and a second set of 16 independent encryptions with the salt as a key. Incrementing κ after each `aesenc` could introduce unwanted data dependencies, but we can deal with that by having a precomputation stage in the compression function that fills a 2048-byte array. We can then use the 16 `xmm` registers to hold the state and to parallelize 2×16 `aesenc` instructions (which have a dependency after 16 instructions). We have also implemented a 32-bit variant of the code using half of the available `xmm` registers (8 instead of 16 in `amd64` mode). The performance figures remain very competitive as reported on the table.

The advantages of using the AES instructions set—as well as increased resistance to side-channel attacks—are clear.

7.4 Smart Cards and 8-bit Implementations

Given modern trends in smart card design, new devices no longer pose the extreme challenges that were seen even as recently as during the AES process. While there remain smart cards with very limited memory in use, they are not representative of the kind of cards that would be used for new applications. Smart cards that were viewed as high-end ten years ago (during the AES process) are now much cheaper and used routinely. Cards with at least 500 bytes and more usually 1 Kbyte of RAM are common-place and 500 bytes is more than enough for ECHO.

For estimates of the 8-bit performance of ECHO one can turn to figures for the implementation of the AES. Since the fundamental component of ECHO is the AES round ECHO inherits the same byte-oriented architecture. Over the course of one iteration of the compression function COMPRESS_{512} we compute 256 AES rounds. Since the AES is typically implemented in smart cards with *on-the-fly* key generation, we might reuse exactly the same number of cycles per round that are estimated for the AES, and assume that this covers the actions of the SALT and κ in ECHO. Thus, we can start our estimates by multiplying the cycles per round for an AES implementation by 256. The most significant additional cost in ECHO would be the 64 additional AES `MixColumns` operation that figure in BIG.MIXCOLUMNS and we estimate this by $20 \times 64 = 1280$ cycles per round. Taken together these lead to the following indicative figures for ECHO with $128 \leq \text{HSIZE} \leq 256$. For ECHO with $257 \leq \text{HSIZE} \leq 512$ the performance can be expected to drop by a factor of $\frac{8}{15}$.

Extrapolating from ...	AES *cycles/round*	AES *cycles/byte*	ECHO *cycles/byte*
8051 AES estimates [17]	254	159	595
68HC705 AES estimates [40]	946	591	1314
ARM AES estimates [30]	289	181	439

A first-cut implementation of ECHO with $128 \leq \text{HSIZE} \leq 256$ on the Atmel AT90SC (3.68 MHz) runs at roughly 360 cycles/byte. This is slightly better than the extrapolations given above.

7.5 Hardware Implementations

Over the years much work has been done on FPGA and ASIC implementations of the AES. As is well-known, the range of possible performance trade-offs for hardware implementations is significant, depending on whether cost, throughput, or area/energy consumption is a priority. It is therefore impossible to capture the performance profile of any algorithm in just a few figures.

AES implementation work has often focused at the two extremes of the hardware profile; implementations that attempt to maximise throughput and implementations that attempt to minimise any area/cost/energy demands. All of this work can be applied to ECHO and so, for instance, techniques that provide

some of the smallest AES implementations [25] can be extended to implementations of ECHO. However, we question the usefulness of making such low-cost estimates for SHA-3 candidates since the must-satisfy parameter choices and security properties [63] are not particularly relevant to resource-constrained environments.[5] Our focus at this stage, therefore, will be on estimating the throughput of ECHO using existing AES implementations for which throughput is a priority.

The performance of a hardware implementation will depend on two main factors; the size of the architecture (*i.e.* 32-bit or 128-bit) and the use of loop unrolling and pipelining. The fastest AES implementations will be 128-bit architectures while the value of loop unrolling and pipelining will depend on the mode of operation of the AES. For ECHO the issue is somewhat simplified since we do not need to support different modes of operation. Instead, when hashing a single message we can only compute the second iteration of a compression function when the first one is finished, which makes it similar to what are termed the feedback modes of operation for the AES. For a single message there is probably little additional benefit in considering a fully-unrolled implementation, though it is true that independent hash inputs could be interleaved to ensure that any fully-enrolled circuit always remains busy. In general terms a hardware implementation of ECHO will inevitably be significantly larger than one for the AES. However these additional costs should be mitigated a little since:

- we do not need to support decryption,
- we replace the key schedule with a simple counter, and
- there is a two-fold opportunity for parallelism, both within the AES round and across the state[6] table S.

Estimates for the hardware performance profile of ECHO might go as follows. Here we consider the two extremes of (i) maximum throughput but large area, and (ii) minimum area with potentially limited throughput.

(i) For the highest throughput it is possible to fully unroll and pipeline all the stages of the ECHO compression function. The state S is made of 16 cells, each one being processed independently during the two AES rounds of BIG.SubWords. The output is then available for the BIG.MixColumns operation, which can also be performed in parallel using 64 dedicated circuits. The whole combination of operations would be repeated 8 or 10 times depending on the ECHO variant. Pipelining allows to process the core operations of the compression function in 8 (*resp.* 10) cycles, plus one cycle for the feedforward step. This strategy has been implemented in [50, 51] where a front-end serial to parallel module is used to feed the state with chunks of incoming message, and a Finite State Machine is used

[5] In many low-cost applications there is no need for collision-resistance and an 80-bit security level is often viewed as adequate, particularly if there is a need to minimize cost.

[6] By construction, actions on S replicate the AES structure at a higher level.

to control the data path inside the rounds. For ASIC implementations, VHDL simulations for 0.13 μm CMOS technology indicate a throughput of 14.85 Gbps at 87.1 MHz for $128 \leq$ HSIZE ≤ 256, and a throughput of 7.75 Gbps at 83.3 MHz for $257 \leq$ HSIZE ≤ 512. Similar results were obtained for FPGA implementations (Xilinx Virtex 5 technology). Using some optimization tricks on the AES modules, even a better throughput of **26.4 Gbps** has been achieved on Virtex 5 (**30 Gbps** on Virtex 6) with a reasonable impact on the area for $128 \leq$ HSIZE ≤ 256. This implementation will soon be available on the ECHO website.

(ii) For a reduced area implementation of ECHO, it would be possible to use only one copy of the iterated one-round AES block (with 16 SBoxes) BIG.SUBWORDS throughout (in 32 cycles), and a small circuit that mixes columns in 128-bits chunks. Some 2048-bit registers are necessary to store the data waiting to be processed between the input, the BIG.SUBWORDS, the BIG.MIXCOLUMNS and the final stage that produces the hash output. Such a design, implemented in [50, 51], shows a throughput of 0.37 Gbps at 66.6 MHz for 82.8 KGates on ASIC technology. One can imagine better results would be obtained by using a single-round AES block with only 1 SBox unit and only 1 MixColumns unit for BIG.SUBWORDS and BIG.MIXCOLUMNS: the MixColumns unit will be shared throughout the whole Compress. Such a strategy has been implemented on Virtex 5 FPGA in [6], where ECHO takes only 127 slices and one memory block on the area while running at 72 Mbps (with a 352 MHz frequency).

Between these two extreme designs there are many different variants using different levels of parallelism to give different throughputs and area requirements. A simple approach would be to find a suitable compromise between fully parallelized and fully iterated ECHO. For instance, one can use 8 AES blocks instead of 16, making two iterations for each half of the state, and then using only 32 cells for the BIG.MIXCOLUMNS stage. While the number of cycles for one iteration of the compression function would be roughly doubled, this does allow us to get a range of results. Extrapolating from previous published results on ASIC [50], a very crude estimate would give the following approximations for a $128 \leq$ HSIZE ≤ 256 hash output (supposing that the area requirement of an AES two-round block is around 30 KGates):

- 16 AES blocks — 500 KGates @ 14.85 Gbps
- 8 AES blocks — 290 KGates @ 7 Gbps
- 4 AES blocks — 170 KGates @ 3 Gbps
- 2 AES blocks — 120 KGates @ 1 Gbps

These figures don't take into account bottlenecks and constant-cost effects on the area and the throughput, but they do reflect the interesting range of trade-offs. As already stated, more variants could be designed using a one-round block AES

as the starting point, or even using different basic AES-block implementations depending on the target architecture. And since surveys such as [32] show that the AES can attain encryption speeds close to 70 Gbps in a high-throughput implementation, improving the hashing rates over 30 Gbps for ECHO is entirely plausible.

In short, the design space for implementing ECHO (and of course ECHO-SP) is vast and a more complete evaluation of the possible trade-offs will gradually take shape. Nevertheless, since ECHO builds directly on the AES we believe this evaluation will be quickly achieved and, instead of reinventing the implementation wheel, existing work on the AES, with regards to both performance and resistance to side-channel attack, can be leveraged directly.

Part III
Security Claims and Analysis

In this final part of the document we provide a security assessment of ECHO.

Summary

- When ECHO is used to generate a hash output of n bits, where n takes the values 224, 256, 384, and 512 bits, we claim that
  - the work effort to compromise collision resistance is $2^{\frac{n}{2}}$ operations,
  - the work effort to compromise preimage resistance is 2^n operations,
  - there are no efficient distinguishing attacks against the HMAC construction using ECHO, and that
  - full security is provided against the message/salt attack described by NIST in [63].

  In short, the best attacks on ECHO are generic attacks.
- The number of rounds in ECHO can be viewed as tunable parameter. We do not envisage any attack on six or more iterations of BIG.ROUND that would compromise the above security claim for $128 \leq$ HSIZE ≤ 256. Nor do we envisage any attack on eight or more iterations of BIG.ROUND that would compromise the above security claim for $257 \leq$ HSIZE ≤ 512.
- The expected probability of the best characteristic over four BIG.ROUND operations in ECHO, averaged over salt and counter, is upper-bounded by 2^{-750}.

  UPDATE: *A recent improvement of the proof shows that the expected probability of the best characteristic over four* BIG.ROUND *operations in* ECHO*, averaged over salt and counter, is actually upper-bounded by* 2^{-1200}*.*
- The expected probability of the best differential over four BIG.ROUND operations in ECHO, averaged over salt and counter, is upper-bounded by 1.1×2^{-452}.
- In a direct parallel to the AES, structural distinguishers can be applied to reduced-round ECHO. These reflect the AES-structure underlying ECHO and help to set the security bounds.
- The internal counter κ is vital to the security of ECHO. Without this counter, symmetries leading to trivial weaknesses can be identified.

- ECHO uses the double-pipe strategy [52] which eliminates generic attacks, such as multi-collision and herding attacks, on the Merkle-Damgård construction. ECHO is also resistant to length-extension attacks.

8 Security Claims

In an iterated hash function, the relationship between the length of the hash output and the length of the chaining variable has an impact on the security offered. In the table below, we give the expected security levels against different generic attacks when the chaining variable and the hash output are both n bits in length and different iterated constructions are used.

Attack	Ideal Hash	Merkle Damgård	HAIFA
Preimage	2^n	2^n	2^n
2^{nd}-preimage (k blocks)	2^n	$\frac{2^n}{k}$	2^n
Collision	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$
k-multi-collision	$2^{\frac{n(k-1)}{k}}$	$\lceil \log_2 k \rceil 2^{\frac{n}{2}}$	$\lceil \log_2 k \rceil 2^{\frac{n}{2}}$

Details on these generic attacks (and extensions such as herding attacks) and the security offered using different techniques is given in Section 10. However, in ECHO we use a chaining variable that is twice the size of the hash output n (the double-pipe strategy). This results in the following security claims for a hash output of n bits.

Attack	Double Pipe	ECHO	ECHO-SP
Preimage	2^n	2^n	2^n
2^{nd}-preimage (k blocks)	2^n	2^n	2^n
Collision	$2^{n/2}$	$2^{n/2}$	$2^{n/2}$
k-multi-collision	$2^{\frac{n(k-1)}{k}}$	$2^{\frac{n(k-1)}{k}}$	$\lceil \log_2 k \rceil 2^{\frac{n}{2}}$

9 The Compression Function

The compression function of ECHO operates on a large internal state of 2048 bits. In some sense this was a natural consequence of our design philosophy since the most effective way to guarantee a simple analysis was to directly replicate the AES structure. We had tried smaller states before, but we were dissatisfied with the strength of the security arguments we could make.

However once we have arrived at a larger state, then there are various advantages to be had. A large state allows us to carry a larger chaining variable, which has obvious advantages against domain extension algorithm attacks and for our security claims (see Section 8). And provided the large state has good cryptographic mixing, *i.e.* good confusion and diffusion, we can drive down the success probabilities of an attacker attempting to manipulate a differential path. We now explore this issue in more detail.

9.1 Confusion and Diffusion

Shannon's famous terms *confusion* and *diffusion* [71] often feature in block cipher design and analysis. These notions can be loosely transferred to hash functions even though there is now no secret key. The concepts are somewhat imprecise and have been interpreted by commentators in different ways, but *confusion* is often used to capture the idea of making the relationship between input bits complicated, while *diffusion* is often used to describe the distribution of change or influence.

It is not always easy to identify exactly how each component contributes to the net gain in confusion or diffusion and, as a result, such analysis should not be taken too seriously. But we know from results on the AES that the S-box is a good source of confusion and makes an ideal contribution to the resistance of the cipher to differential and linear cryptanalysis. Since we build on the AES, this is a property that we inherit. Added to this, we can stress the importance of the internal counter κ in the compression function of ECHO. The counter κ determines the subkeys that are used in BIG.SUBWORDS, and since the counter changes from cell to cell each set of two AES-rounds embodies a different transformation. The bit counter $\mathcal{C}_i$ that determines the initial value of κ never takes the same value during the hashing of a message and this property is inherited by the initial values of the internal counter κ. For very long messages it is possible for κ to return to zero during a compression function computation but this is of no consequence. In a similar vein, it is possible for the same values of κ to appear in two successive calls to the compression function, for instance in the last two calls to the compression function for a message of a suitable length. But it is not possible for κ to take the same values at the same corresponding positions of the compression function computation. Even if this were not the case, having distinct values of κ between compression function calls was not a design goal and their potential recurrence is of no consequence.

The actions of the S-boxes and κ are integrated into a very successful diffusion mechanism that is directly inherited from the AES. To analyse diffusion

in the BIG.ROUND function, we will appeal to the classical result that over two rounds of the AES a change to a single byte in the input to the AES influences all 16 state bytes after two encryption rounds. A similar result can be claimed for any two BIG.ROUND operations, if we change our reasoning from bytes to words, *i.e.* that a change to a single 128-bit word in the input to BIG.ROUND has an influence on all 16 words after two BIG.ROUND operations. In fact, a closer analysis shows that the modification of any internal state byte in COMPRESS_{512} or COMPRESS_{1024} potentially impacts every internal state byte after two iterations of BIG.ROUND. While we have these useful results on diffusion, there are still some decisions to make on the placement of the inputs to COMPRESS_{512} and COMPRESS_{1024} in the state S. We elected to pack the chaining variable and message words in such a way as to ensure that the input chaining variable potentially impacts the whole internal state with the first BIG.MIXCOLUMNS transformation since BIG.SHIFTROWS is applied to the internal state before the BIG.MIXCOLUMNS transformation.

9.2 Differential Cryptanalysis

Differential cryptanalysis [10], while primarily viewed as a tool for block cipher cryptanalysis, has also been one of the most successful ways to attack hash functions. Indeed the *collision* is, in effect, a very carefully controlled differential path and many of the considerations that apply to the differential cryptanalysis of block ciphers apply equally well to hash functions. Interestingly, many recent attacks exploit the relatively weak "key schedule" that is found in many contemporary hash function designs, and this has given many differential attacks a new flavour. In this section, therefore, we concentrate on establishing the level of resistance of ECHO to the many differential techniques that are available to the cryptanalyst.

9.2.1 Notation

For ease of exposition we will need to consider the action of the compression functions COMPRESS_{512} and COMPRESS_{1024} without the BIG.FINAL and feedforward operations. What remains after removing these operations is, in fact, a block cipher and it is the block cipher that naturally arises from our extension of the AES. We will therefore call this derived cipher ECHO.AES. It has a 2048-bit block size and consists of the composition of r round functions ECHO.ROUND that each take as input the state of the block cipher S and a 4096-bit round key K. The round key can be viewed as a sequence of 128-bit words and it consists of the sixteen (k_1, k_2) pairs for each of the cells in S:

$$K = k_1^0 \parallel k_2^0 \parallel k_1^1 \parallel k_2^1 \parallel \cdots \parallel k_1^{15} \parallel k_2^{15} \ .$$

This round function is almost identical to BIG.ROUND though it takes as input (S, K) instead of (S, SALT, κ). The round function ECHO.ROUND is therefore defined as the sequential application of the three functions:

ECHO.SUBWORDS(S, K)	
ECHO.SHIFTROWS(S)	(identical to BIG.SHIFTROWS)
ECHO.MIXCOLUMNS(S)	(identical to BIG.MIXCOLUMNS)

The function ECHO.SUBWORDS(S, K) is very closely related to the function BIG.SUBWORDS and is defined as:

$$\begin{aligned} w'_0 &= \text{AES}(\text{AES}(w_0, k_1^0), k_2^0) \\ w'_1 &= \text{AES}(\text{AES}(w_1, k_1^1), k_2^1) \\ &\vdots \\ w'_{15} &= \text{AES}(\text{AES}(w_{15}, k_1^{15}), k_2^{15}) \end{aligned}$$

For analysis purposes we will use ECHO.SBOX to denote the AES-based permutation on 128-bit words w parametrized by two 128-bit keys k_1 and k_2:

$$\text{ECHO.SBOX}(w, k_1, k_2) = \text{AES}(\text{AES}(w, k_1), k_2) \ .$$

We emphasize that the cipher ECHO.AES is only a tool to allow the analysis of the ECHO compression function. This helps us to derive our security results. However ECHO.AES should not, in itself, be viewed as a secure block cipher proposal.

9.2.2 Basic concepts

The differential properties of the AES have been extensively studied. Before applying them to our construction let us recall a few definitions. For 128-bit input and output differences Δw and $\Delta w'$, and two fixed keys k_1 and k_2, we will use $\mathtt{dp}_{2\text{A}}[k_1, k_2](\Delta w, \Delta w')$ to denote the *differential probability* of the ECHO.SBOX transformation.[7] This is classically defined as

$$\mathtt{dp}_{2\text{A}}[k_1, k_2](\Delta w, \Delta w') = \frac{\#\{x \in \{0,1\}^{128} | \text{ECHO.SBOX}(x \oplus \Delta w, k_1, k_2) \oplus \text{ECHO.SBOX}(x, k_1, k_2) = \Delta w'\}}{2^{128}}$$

Note that this value does not depend on k_2 since k_2 is exclusive-ored at the end of the second AES round. With this definition to hand we can now define:

$\mathtt{edp}_{2\text{A}}(\Delta w, \Delta w')$: the *expected differential probability*, the average of the differential probability over all keys, and

$\mathtt{medp}_{2\text{A}}$: the *maximum expected differential probability*, the maximal value of $\mathtt{edp}_{2\text{A}}(\Delta w, \Delta w')$ for all non-zero Δw and $\Delta w'$.

[7] We use the subscript 2A to emphasize its composition as two AES rounds.

Similar values can be defined for the composition of r rounds of the transformation ECHO.ROUND(S, K) and we will denote the corresponding values

$$\mathrm{DP}_r[K_1, \ldots, K_r](\Delta S_0, \Delta S_r)$$
$$\mathrm{EDP}_r(\Delta S_0, \Delta S_r)$$
$$\mathrm{MEDP}_r$$

respectively, where ΔS_0 is the input difference to the first round and ΔS_r is the output difference from the r^{th} round.

So far we have considered what is, in essence, a *differential* behavior since we have paid no attention to the intermediate difference values generated at each AES round. At times, however, we will need to define a characteristic, and so we will adopt the following notation:

$$\Omega = (\Delta S_0, \Delta S'_0, \Delta S_1, \Delta S'_1, \ldots, \Delta S_{r-1}, \Delta S'_{r-1}, \Delta S_r) \ ,$$

where ΔS_i denotes the input to the $(i+1)^{\text{st}}$ round and $\Delta S'_i$ denotes the intermediate difference after the first AES round in the $(i+1)^{\text{st}}$ ECHO.SUBWORDS operation. The probability of the characteristic Ω is defined as the expected probability of the characteristic over all keys and will be denoted $\mathrm{ECP}_r(\Omega)$. The maximal expected characteristic probability, $\max_\Omega \mathrm{ECP}_r(\Omega)$, will be denoted MECP_r.

As is usual in the literature on differential cryptanalysis, for a characteristic Ω we say that any ECHO.SBOX with a non-zero input/output is *active*. Similarly, in an active ECHO.SBOX, any specific AES S-box with a non-zero input/output will be called an *active AES S-box*.

9.2.3 Characteristics and differentials

We are now ready to state our results on the differential properties of the cipher ECHO.AES. This is the underlying component that is derived from the compression function of ECHO, and our results will carry over to the compression function, and then on to ECHO itself.

The case of *characteristics* is easily handled using existing results on the AES. Indeed one can show, as for the AES [15], that characteristics over four rounds of ECHO.AES require at least 25 active ECHO.SBOX. In turn, each active ECHO.SBOX implies at least five active AES S-boxes, and so any characteristic over four rounds of ECHO.AES will have at least 125 active AES S-boxes. Since the maximal differential probability of the AES S-box is 2^{-6} this gives the following result:

Theorem 1. *The maximal expected probability for a characteristic over four rounds of* ECHO.AES *is upper-bounded by*

$$\mathrm{MECP}_4 \leq 2^{-750} \ .$$

This result can be improved by using the Super-Sbox view [16, 19, 18] of an AES-like permutation. One can write four rounds of ECHO.AES as

$$BSW \circ BSR \circ BMC \circ BSW \circ BSR \circ BMC \circ \\ BSW \circ BSR \circ BMC \circ BSW \circ BSR \circ BMC. \quad (1)$$

It is obvious that the linear part $BSR \circ BMC$ at the end of the fourth round has no impact on the number of active AES S-boxes. Moreover, we can commute the layers BSW and BSR and we are left with

$$BSR \circ BSW \circ BMC \circ BSW \circ BSR \circ BMC \circ BSR \circ BSW \circ BMC \circ BSW.$$

Again, the first BSR layer has no impact on the number of active AES S-boxes and we can see the 512-bit Super-Sboxes $BSW \circ BMC \circ BSW$ appearing. We denote $SBSW = BSW \circ BMC \circ BSW$ and we have to analyse

$$SBSW \circ (BSR \circ BMC \circ BSR) \circ SBSW.$$

It is clear that the $SBSW$ layer updates all the 512-bit columns independently and we denote by min_{SBSW} the minimal number of active AES S-boxes through $SBSW$. We know that through BMC an active 512-bit column will contain at least 5 active words on its input and output. Therefore, we can directly conclude that at least five columns will be active on the input and output of the middle layer ($BSR \circ BMC \circ BSR$). Thus, the minimal number of active AES S-boxes in a characteristic over four rounds of ECHO is $5 \times min_{SBSW}$.

Since BIG.SUBWORDS is composed of two AES rounds, we can write one 512-bit Super-Sbox

$$SB \circ ShR \circ MC \circ SB \circ ShR \circ MC \circ BMC \circ SB \circ ShR \circ MC \circ SB \circ ShR \circ MC.$$

Using the same commutation technique as before, we deduce that we have to analyse

$$SB \circ MC \circ SB \circ ShR \circ MC \circ BMC \circ ShR \circ SB \circ MC \circ SB.$$

We can see the 32-bit Super-Sboxes $SB \circ MC \circ SB$ appearing. We denote $SSB = SB \circ MC \circ SB$ and we are left with

$$SSB \circ (ShR \circ MC \circ BMC \circ ShR) \circ SSB.$$

The middle layer $ShR \circ MC \circ BMC \circ ShR$ forces that at least 8 columns of 32 bits will be active on its input and output (because the byte branching number of $MC \circ BMC$ is equal to 8). Moreover, it is well known that the minimal number of active AES S-boxes through SSB is 5. Finally, we can conclude that $min_{SBSW} = 8 \times 5 = 40$ and the minimal number of active AES S-boxes through four rounds of ECHO is equal to 200.

Theorem 2. *The maximal expected probability for a characteristic over four rounds of* ECHO.AES *is upper-bounded by*

$$\mathrm{MECP}_4 \leq 2^{-1200} \ .$$

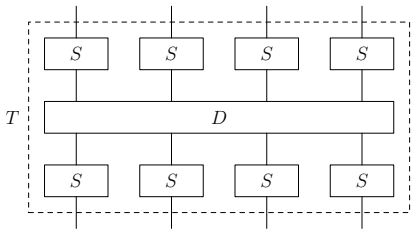


Figure 1: The SDS transformation, where S denotes a keyed m-bit to m-bit bijection, and D denotes a bijective GF(2)-linear mapping over the set of n-tuples of m-bit words, with $n = 4$.

Generally speaking, it is not as straightforward to derive upper bounds for differentials as it is for characteristics. However, due to our design, we are able to give a strong upper bound for the probability of any differential over four rounds of ECHO.AES. To start, we will use the following result on the exact value of the MEDP of two rounds of AES that is due to Keliher and Sui [41]. Note that this value is exactly the MEDP of ECHO.SBOX.

Lemma 1. *The exact value of the MEDP for two rounds of AES is*

$$\mathtt{medp}_{2\mathrm{A}} = \frac{53}{2^{34}} \simeq 1.7 \times 2^{-29} \ .$$

To use this result we will need a generalisation, to be given below, of a result of Hong *et al.* [33]. This result is concerned with the MEDP of what is termed a *substitution-diffusion-substitution (SDS) transformation*. Let S denote a keyed m-bit to m-bit bijection, and let D denote a bijective GF(2)-linear mapping over the set of n-tuples of m-bit words. The SDS transformation associated with D and S is illustrated in the keyed transformation T that appears in Figure 1. The keys of the $2n$ mappings S involved in T are assumed to be independent and uniformly distributed. The linear mapping D is said to be *Maximal Distance Separable* (MDS) if its branch number is equal to $n + 1$; that is each non-zero input-output pair contains at least $n + 1$ non-zero m-bit words.

Let Δx and Δy denote the input and output differences to the SDS transformation and Δz denote the difference after the first layer of S permutations. The SDS transformation is said to be Markovian [48] if, for all input and output difference values Δx and Δy, the following equality holds:

$$\mathrm{EDP}(\Delta x, \Delta y) = \sum_{\Delta z} \mathrm{EDP}(\Delta x, \Delta z) \cdot \mathrm{EDP}(D(\Delta z), \Delta y) \ .$$

Let $\mathrm{MEDP}(S)$ (*resp.* $\mathrm{MEDP}(T)$) denote the maximal expected differential probability of S (*resp.* T). Then one can prove the following lemma:

Lemma 2. *If D is* GF(2)*-linear and MDS and the SDS transformation is Markovian, then*

$$\mathrm{MEDP}(T) \leq \mathrm{MEDP}(S)^n \ .$$

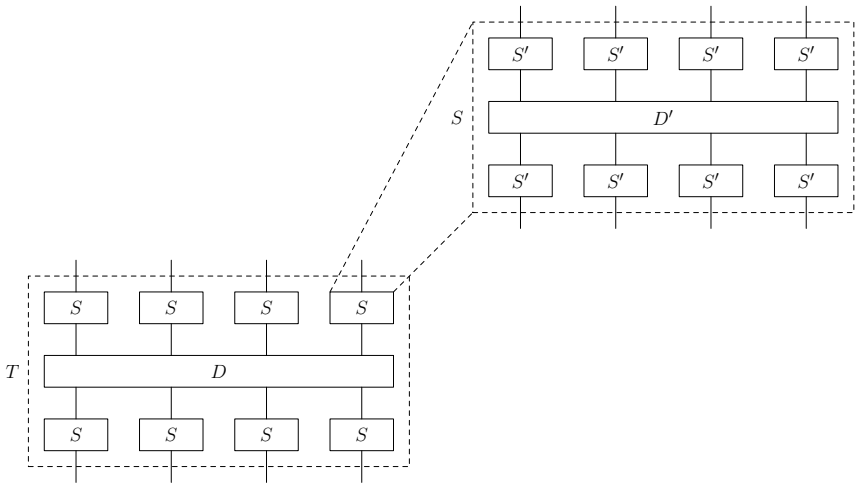


Figure 2: An example of a nested SDS transformation.

Proof. The proof is similar to the one in [33] which assumes that the keyed permutation S consists of the exclusive-or of a round key to the state followed by the application of a fixed S-box, which yields a special case of Markovian SDS transformation. We also observe that while the proof of [33] uses the assumption that D is GF(2^m)-linear, it can be easily extended to the more general case where D is GF(2)-linear. The remainder of the proof is unchanged. □

We can now apply this result in a recursive manner to ECHO.AES using an idea due to Rijmen [69] and also used by Ohkuma *et al.* [64]. Consider the entirety of the nested construction that is depicted in Figure 2, and where we now observe that the keyed permutation S is, itself, another SDS transformation. Provided that the linear mappings D and D' are MDS and that the Markovian property holds for both SDS transformations S and T, then we can apply Lemma 2 to both levels of the nested construction T to give the upper bound:

$$\mathrm{MEDP}(T) \leq \mathrm{MEDP}(S')^{n^2} \ ,$$

where n is the number of parallel permutations S' in each substitution layer of the SDS transformation S.

But how do we apply this nested SDS to ECHO.AES? This next step is due to an observation of Rijmen [69] and Ohkuma *et al.* [64] who showed that four-round AES can be viewed as an instance of just this kind of nested construction in Figure 2. There is a difference in the fixed initial and final linear mappings, but these have no effect on the MEDP value that is derived. In this way, an upper bound of $2^{-6\times 16} = 2^{-96}$ was derived for the MEDP on four-round AES, though the tighter upper bound of $\left(\frac{53}{2^{34}}\right)^4 \approx 1.881 \times 2^{-114}$ was later established in [41].

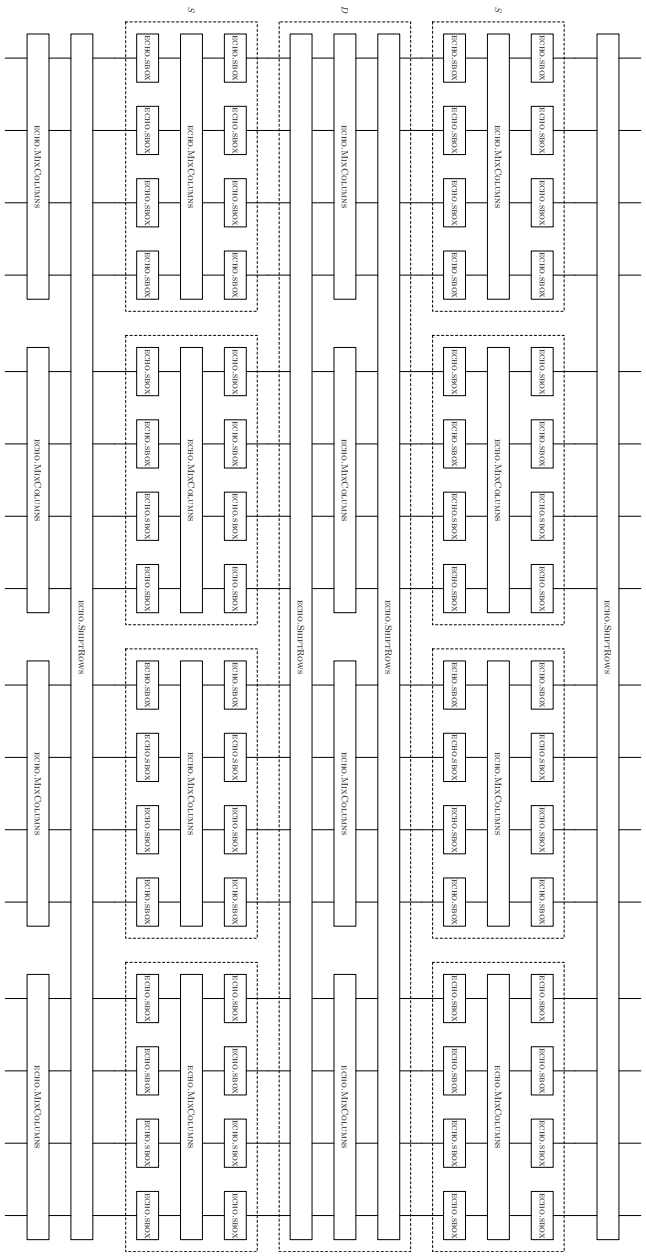


Figure 3: An equivalent representation of four-round ECHO.AES using the fact that ECHO.SUBWORDS and ECHO.SHIFTROWS commute. Recall that ECHO.SBOX is keyed (the input keys are not represented). In a slight abuse of notation the ECHO.MIXCOLUMNS denoted here represents the restriction of the full ECHO.MIXCOLUMNS to a single column of the state.

Since ECHO directly replicates the AES structure, the observation of Ohkuma *et al.* can be easily transposed to the four-round version of ECHO.AES. The rewriting of four rounds of ECHO.AES as a nested SDS transformation is shown in Figure 3. Recall that each round of ECHO.AES consists of ECHO.SUBWORDS, ECHO.SHIFTROWS, and ECHO.MIXCOLUMNS and also that ECHO.SUBWORDS and ECHO.SHIFTROWS commute. We can therefore represent four rounds of ECHO.AES as in Figure 3. It is now easy to see that if we ignore the initial ECHO.SHIFTROWS and the final ECHO.MIXCOLUMNS and ECHO.SHIFTROWS transformations, then we get an instance of the nested SDS construction of Figure 2. The parameters are $m = 128$ and $n = 4$ and dotted lines are used to group the second-level internal SDS transformations as well as the top-level linear mapping D. Then, to complete the derivation of our result, we note the following.

1. The linear mapping D' applied to each column of ECHO.MIXCOLUMNS is MDS with respect to 128-bit words.

2. The linear mapping

   $$D = \text{ECHO.SHIFTROWS} \circ \text{ECHO.MIXCOLUMNS} \circ \text{ECHO.SHIFTROWS}$$

   is MDS with respect to 512-bit words. This easily follows since a non-zero input to D requires that at least one of the four instances D' in ECHO.MIXCOLUMNS is "active" which means that at least five of the four 128-bit input and four output words of D' are non-zero. These four 128-bit inputs (*resp.* output) words belong to four distinct 512-bit groups at the input (*resp.* output) of D.

3. The inner and outer SDS transformations in Figure 3 are Markovian. This follows since the second subkey k_2 is exclusive-ored at the end of the ECHO.SBOX transformation.

Using Lemma 1 we can now state the following result:

Theorem 3. *The maximal expected differential probability for four rounds of* ECHO.AES *is upper bounded by* $(\texttt{medp}_{2\text{A}})^{16}$*, that is*

$$\text{MEDP}_4 \leq \left(\frac{53}{2^{34}}\right)^{16} \approx 1.055 \times 2^{-452} \quad .$$

While the bounds in Theorems 1 and 3 are good enough for our purposes, we strongly believe that both bounds are not tight. In Section 9.2.6 we will bring these results together and make our security claims for ECHO.

9.2.4 Fixed-key characteristics

Moving from bounds on the expected probability of a characteristic over all keys, to an estimate with a particular key is not so easy. With this in mind, some authors have considered the case of characteristics for fixed keys. Daemen

and Rijmen have introduced the notion of *plateau characteristics* [16]. Given a characteristic Ω related to a keyed mapping, Ω is called a plateau characteristic if the characteristic probability $\mathrm{CP}[K](\Omega)$ associated with a key K can only take the values 0 and $p_\Omega \neq 0$. It is interesting to note [16] that all the two-round characteristics of AES are plateau characteristics and that the maximum value of $\mathrm{CP}[K](\Omega)$ over all keys and non-trivial 2-round characteristics is equal to $\frac{2^5}{2^{32}} = 2^{-27}$. This analysis on two rounds of the AES, and the accompanying preliminary analysis of its extension to four rounds, indicates that even over more AES rounds, the ratio between the maximum value of $\mathrm{CP}[K](\Omega)$ over all keys and non-trivial characteristics and the MECP remains moderate no matter which key is chosen. Any significant divergence from the expected behavior of a characteristic occurs with a very low probability and this suggests that the maximum fixed-key differential probabilities encountered in an AES-based algorithm, such as the compression function of ECHO, are not expected to vary dramatically from the average case when we fix the SALT and counter.

9.2.5 Truncated differentials

Truncated differentials are a special class of differential attacks. Instead of considering characteristics for which the difference value at each round is entirely specified, we might represent the difference as an n-tuple

$$(\gamma_1, \cdots, \gamma_n)$$

where the entirety of a w-bit block is viewed as, say, n blocks of m bits (with $w = n \times m$) and γ_i takes a one-bit value depending on whether a given m-bit sub-block is active or not. The probabilities associated with truncated differentials are often higher than for regular differentials since less detail needs to be specified. However they are typically most applicable to the analysis of highly-structured primitives. One such example is given by the analysis [66] of Grindahl [45] which also reused AES components. It is therefore worth considering the vulnerability of ECHO to truncated differentials.

While there is no obvious characterization of truncated differential attacks on ECHO.AES, particularly ones that would allow an adversary to construct a collision or pseudo-collision for the compression function, it is natural to assume that the difficulty of a truncated differential attack on $\mathrm{COMPRESS}_{512}$ or $\mathrm{COMPRESS}_{1024}$ would be related to the difficulty of finding a high-probability truncated differential for ECHO.AES. In the notation above, this will amount to finding a byte-wise output difference vector $(\gamma_1, \cdots, \gamma_{256})$ with an exceptionally low Hamming weight $t \ll 256$ indicating that not many of the 256 possible byte positions are active. Strong evidence against this eventuality is given by considering the two most natural attack strategies:

1. Start from a high-weight input difference pattern (or let the weight of a difference pattern grow freely in the first rounds of ECHO.AES) and gradually reduce the weight of the difference pattern during later rounds.

2. Find a truncated characteristic in which the initial and all intermediate difference patterns have an exceptionally low weight.

For the first approach, there appear to be no truncated differentials of non-negligible probability that would allow a high Hamming weight truncated differential, with say close to 256 active bytes, to be reduced to a low Hamming weight truncated differential with, say, 128 active bytes. To see this we note that among the elementary byte-oriented transformations inside ECHO.AES only the `MixColumns` operation can effect the number of active bytes. This operation was analysed in [66] and approximate values for the base two logarithm of the probabilities associated with any four-byte input difference, represented by a binary quartet of Hamming weight ω_i, being mapped to any four-byte output difference, represented by a binary quartet of Hamming weight ω_o, are given below.

$\omega_i \backslash \omega_o$	0	1	2	3	4
0	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0
2	$-\infty$	$-\infty$	$-\infty$	-8	0
3	$-\infty$	$-\infty$	-16	-8	0
4	$-\infty$	-24	-16	-8	0

So if $\omega_i > \omega_o$ are the input and output Hamming weights of any binary quartet that represents a four-byte difference pattern, then the associated approximate probability is upper bounded by $2^{-8(\omega_i - \omega_o)}$. If we now consider any truncated characteristic over several rounds for which the input difference pattern has weight H and the output difference pattern has weight L, then the probability is upper-bounded by a quantity close to $2^{-8(H-L)}$. Thus if $H \geq 228$ and $L \leq 128$, say, then the probability of any truncated differential characteristic with these input and output weights is upper-bounded by about 2^{-800}.

The success of the second approach depends on finding a very sparse truncated differential that holds over sufficiently many rounds of ECHO.AES. To see that this is very unlikely, we consider an analysis of 128-bit words, rather than the byte-level analysis we used before. Any ECHO.AES state difference can be represented by a 16-tuple $(\Delta w_0, \cdots, \Delta w_{15})$ of 128-bit words and an associated truncated difference pattern $\gamma = (\gamma_0, \cdots, \gamma_{15})$ of sixteen binary values. Now consider any non-trivial truncated differential characteristic over four consecutive ECHO.AES rounds. Denote by γ^1, γ^2, γ^3, and γ^4 the difference patterns at the inputs to the first through fourth rounds, where none of the γ^i is trivial. Then at least one of the four patterns from γ^1 through to γ^4 has a Hamming weight at least eight. So for any message pair that satisfies this truncated differential, at least one half of all sixteen 128-bit difference values in at least one round are active, and the excellent diffusion properties of the AES that we have inherited in ECHO make it very difficult to maintain a lightweight truncated differential over many rounds.

Summary: Truncated differentials.

We know of no cryptographic primitive that affords a formal proof of resistance to truncated differential cryptanalysis. In this regard ECHO.AES and ECHO are no different. That said, the inherent structure within the design of ECHO is very strong, and this allows us to give very good analytical evidence that truncated differential cryptanalysis is highly unlikely to be successful, particularly in producing pseudo-collisions or collisions in ECHO.

9.2.6 Differential cryptanalysis and ECHO

Taken together, our results show that the different flavours of differential cryptanalysis are very unlikely to be applicable to ECHO. Our results on characteristics and differentials in ECHO.AES carry over to the compression functions of ECHO. It is important to note that the attacker fully controls the compression function inputs and can increase the potential for a differential attack by, for example, choosing the input words adaptively, by using neutral bits technique [7], by message modification [74, 75, 76, 77], and by using boomerang techniques [38]. We conservatively assume, therefore, that an attacker is only unable to control four rounds of the compression function (half of the scheme for $128 \leq \text{HSIZE} \leq 256$ and less than half the scheme for $257 \leq \text{HSIZE} \leq 512$). Even then, our best differential attacks gives a complexity that far exceeds the difficulty of compromising ECHO by brute-force techniques.

9.3 Resistance to Other AES Attack Methods

Since the compression function of ECHO is so closely related to the AES, we can investigate its resistance to various dedicated attack methods that have been proposed against the AES. Taken together with our differential cryptanalysis, these have helped us set the appropriate number of rounds for COMPRESS_{512} and COMPRESS_{1024}.

9.3.1 Structural attacks

Some of the most efficient attacks against reduced-round versions of AES are based upon efficient distinguishers. Two of the most efficient distinguishers are the three-round integral distinguisher of the Square attack [15] and the four-round integral distinguisher used in the improved integral cryptanalysis of round-reduced AES [26].

Given the design of ECHO the transposition of these distinguishers to ECHO is straightforward. If we apply a three-round ECHO.AES encryption to a set of 2^{128} 2048-bit blocks which take all possible values on one of the sixteen 128-bit word positions and some constant value in all the others, then the sum of all the 2048-bit output blocks is equal to zero. Similarly, if we apply a four-round ECHO.AES encryption to a suitably chosen set of 2^{512} 2048-bit blocks, then the sum of the obtained output blocks will be equal to zero.

The efficient four-round distinguisher used in the so-called *bottleneck attack* on round-reduced AES [27] can also be transposed to ECHO.AES. One would obtain a four-round distinguisher requiring about 2^{256} partial computations to distinguish the four-round ECHO.AES instance associated with a random unknown key from a random permutation. This however is unrelated—even if we were to ignore the considerable gap in the number of rounds—to the attack properties that would be required after eight or ten rounds in trying to compromise ECHO, particularly if we take into account BIG.FINAL.

While improved distinguishers against ECHO.AES and ECHO can never be ruled out, the close relation of our design to the AES, and the amount of study that has taken place on the AES, lends considerable strength to our believe that integral or collision-based distinguishers represent little risk to ECHO.

9.3.2 Known-key distinguishers

In [46] Knudsen and Rijmen introduced a novel security requirement on block ciphers, namely their resistance to so-called *known-key* distinguishers. This was motivated by the observation that block ciphers are sometimes used in settings where the algorithm inputs, outputs and keys are known of an adversary, and that the security requirements in such settings, *e.g.* hashing, are not sufficiently captured by the usual security definitions.

While the authors of [46] recognise that finding a rigorous and practical definition of known-key distinguishers remains an open problem, they describe a reasonably efficient known-key distinguisher on seven-round AES where the `MixColumns` operation is omitted from the seventh round. This distinguisher consists of generating, given any known key, a set of 2^{56} plaintext/ciphertext pairs such that in each of the 32 input or output byte positions, each byte value occurs exactly 2^{48} times. It can be reasonably conjectured that generating such a set in the case of a random permutation over 128 bits would be computationally intractable. The known-key distinguisher of [46] exploits the existence of efficient integral distinguishers for AES encryption over the last four rounds and for decryption over the first three rounds. The adversary starts from an appropriately chosen structure of 2^{56} middle blocks and exploits these distinguishers to build the plaintext/ciphertext pairs.

The transposition of this distinguisher to ECHO.AES is straightforward. It would allow us to distinguish seven-round ECHO.AES without the transformation BIG.MIXCOLUMNS in the seventh round, using a structure of $2^{7\times128} = 2^{896}$ 2048-bit blocks, and its complexity would be close to 2^{896} seven-round ECHO.AES computations. However, due to the huge complexity and the number of input blocks it involves, this distinguisher clearly does not threaten the security of the ECHO compression function. Moreover, even if we assume that an improved known-key distinguisher might be found in the future, it would be unlikely to result in a distinguishing property for the ECHO compression function since the BIG.FINAL transformation has a convolution effect on the ECHO.AES output distribution.

9.3.3 Algebraic attacks

It is well-known that the AES has a relatively simple algebraic description [56]: it can be described as a sparse and overdetermined multivariate quadratic system of 5248 equations over GF(2^8) with 2560 state variables and 1408 key variables. This observation can likely be extended to ECHO.AES, and in turn to ECHO, but resulting in a much larger algebraic system. Given the lack of progress in algebraic cryptanalysis on these kind of systems, we would be surprised if algebraic cryptanalytic techniques were applicable to any great extent on ECHO. We also note that since the path of each input bit can be tracked through 16 to 20 AES S-boxes, the total degree of the equations describing the compression function of ECHO will have a very high degree. We believe this will provide resistance to the recent cube attack of Dinur and Shamir [23].

9.3.4 Related-key attacks

The design of ECHO is such that we do not have a conventional key schedule; the attacker is not able to control the chaining variable and the message inputs in such a way as to generate differences that would be input part-way through a computation. Once a difference in these quantities is input to the compression function, they are not used elsewhere. The only additional inputs are the counter κ and SALT. These certainly are not available to the kind of manipulation that we might see applied to the chaining variable and message; for instance SALT might not even be used and might be set to zero in most applications. However some particularly sophisticated attacks might be worth considering.

Related counters.

In ECHO the value of the internal counter at the start of compression is assigned by the value of $\mathcal{C}_i$ and so it can only be controlled in a very crude way by an attacker. Moving blocks of message would allow differences to be invoked in the two parallel instantiations of κ, but it would at the same time mean that the attacker would have to control, or at least account for, the induced difference in the chaining variable. Much of Section 9.2 was devoted to demonstrating how hard this would be. If one were to consider a weakened version of ECHO where an attacker could control the counter values absolutely, then we can demonstrate an interesting interaction with the SALT. However the probability of the resultant differential is beyond our security bounds, and there are numerous reasons why even this simplest approach does not apply to either the weakened or the full version of ECHO.

Assume that at one iteration of the compression function there is no difference in the chaining variable or message input. Then suppose that a low-weight difference is applied to κ so that the resultant differences in κ throughout the compression function computation remain in a single byte. (This is possible since κ increases by less than 256 during compression.) Then these one-byte differences will be active during every BIG.SUBWORDS call of the compression function. More precisely, differences in κ will be added after the first AES round

since κ is the first of two subkeys. The second AES round will then diffuse the one-byte differences to four bytes using `MixColumns` in the AES round. The second subkey is SALT and we might assume that the attacker can choose the appropriate difference to the SALT so that differences created by κ are systematically erased by SALT. This might be viewed as *local collisions* [14] in every BIG.SUBWORDS call, and the hope would be to obtain a collision at the end of the compression function computation. However, for each local collision, under the best conditions for the attacker only one AES S-box is active. Thus we have 16 local collisions per round, which gives a success probability upper bounded by $2^{-96\times r}$ over r rounds. Already this is enough to see that such an attack is very unlikely and this is before we have even considered whether such differentials are structurally feasible given that SALT is added in the same way across all table entries across all rounds.

Related salts.

Since the counter κ is difficult to control and since it increments during compression, the attacker might be more interested in trying to set up differences between parallel hashes using values to SALT that are related in some way. Here we compute a bound on the probability of success for any differential path that is generated in this way, whatever the input or output differences to the compression function.

For this we introduce the following notation. Let n_j^{in} be the number of active 128-bit words at the input to BIG.ROUND j of the compression function, and let n_j^{out} be the number of active 128-bit words after the BIG.SHIFTROWS function of BIG.ROUND j but before BIG.MIXCOLUMNS. This means that n_j^{out} and n_{j+1}^{in} are separated by the MDS-based BIG.MIXCOLUMNS operation and it can be seen that:

$$4n_{j+1}^{\text{in}} \geq n_j^{\text{out}} \ .$$

At every round, and in every 128-bit word of the state S, the application of SALT will introduce a non-zero difference during BIG.SUBWORDS where SALT is used as the subkey to the second AES round. Therefore, for any round j, the number of active words before BIG.SUBWORDS plus the number of active words after BIG.SUBWORDS is at least 16. Since BIG.SHIFTROWS does not affect the number of active words, we have that $n_j^{\text{in}} + n_j^{\text{out}} \geq 16$ and over r rounds

$$\sum_{j=1}^{r}(n_j^{\text{in}} + n_j^{\text{out}}) \geq 16r \ .$$

Whatever the input and output differences, since n_j^{in} also represents the number of active words before applying the BIG.SUBWORDS function in round j, for an r-round differential path the number of active ECHO.SBOX is at least $\min \sum_{j=1}^{r} n_j^{\text{in}}$, where the minimum is taken over all valid values for n_j^{in} and

n_j^{out}. Putting this together we have[8]:

$$\begin{aligned}
\sum_{j=1}^{r}(n_j^{\text{in}} + n_j^{\text{out}}) &\geq 16r \\
\sum_{j=1}^{r} n_j^{\text{in}} + 4\sum_{j=2}^{r} n_j^{\text{in}} + n_r^{\text{out}} &\geq 16r \\
\sum_{j=1}^{r} n_j^{\text{in}} + 4\sum_{j=1}^{r} n_j^{\text{in}} + n_r^{\text{out}} - n_1^{\text{in}} &\geq 16r \\
5\sum_{j=1}^{r} n_j^{\text{in}} &\geq 16r - n_r^{\text{out}} + n_1^{\text{in}} \\
5\sum_{j=1}^{r} n_j^{\text{in}} &\geq 16(r-1) \\
\sum_{j=1}^{r} n_j^{\text{in}} &\geq \left\lceil \frac{16(r-1)}{5} \right\rceil \quad .
\end{aligned}$$

This analysis shows that any differential path using related salts will contain at least ten active ECHO.SBOX for four rounds and sixteen for six rounds. Since the probability for an active ECHO.SBOX is upper-bounded by 2^{-30} the probability of a four-round differential path is upper bounded by 2^{-300} and a six-round differential path by 2^{-480}.

We stress that it is very unlikely that a differential path exists that meets these bounds. To be valid such a differential path imposes numerous constraints on SALT, of which many are likely to be contradictory. Also our bound was derived by considering any differential path; the goal of achieving a collision in the compression function requires us to consider BIG.FINAL which adds even more constraints and further lowers the probability. In short we do not believe that differential attacks on ECHO exploiting the relationship between salts are viable.

9.4 New Analysis of AES-Based Hash Functions

Reflecting the simplicity and clarity of AES-like design techniques, some new cryptanalysis based on truncated differentials [44] has recently been published. These are particularly relevant to hash functions that use AES-like techniques and it is interesting to note that results can be divided into five different types:

- *simple truncated differential attacks* [66],
- *rebound attacks* [54, 53],

[8]For the penultimate inequality $n_1^{\text{in}} = 0$ and $n_r^{\text{out}} = 16$ covers the best candidate among all differential paths.

- *start-from-the-middle attacks* [53, 67],
- *super-Sbox attacks* [28, 49, 67, 70], and
- *multiple inbound attack* [73].

The first type of attack, using regular truncated differential attacks, was already considered during the design phase of ECHO. It was already known that the very substantial use of the `MixColumns` transformation in ECHO effectively eliminates this threat.

The second and third types of analysis, rebound attacks and start-from-the-middle attacks, were applied on ECHO in [53]. It was claimed that the *internal permutation* that lies within the ECHO compression function can be distinguished from a random 2048-bit permutation with 2^{384} computations and 2^{64} memory when reduced to 7 rounds. However, an inconsistency was pointed out in regards to the utilization of this technique for ECHO [34].

The fourth type of analysis is, perhaps, currently the most effective for collision search on ECHO compression function. The latest such analysis [67, 70] show that one can derive a distinguishing attack against 4.5 rounds of the 256-bit version of the ECHO compression function with an effort of 2^{96} computations and 2^{32} memory and against 6.5 rounds of the 512-bit version with an effort of 2^{96} computations and 2^{32} memory. Regarding free-start collisions for the compression function, 3 rounds of the 256-bit version can be attacked with 2^{64} computations and 2^{32} memory while 3 rounds of the 512-bit version can be attacked with 2^{96} computations and 2^{32} memory.

Moreover, the full 8-round internal permutation can be distinguished from a random 2048-bit permutation with 2^{512} computations and 2^{512} memory while the 7-round version requires 2^{128} computations and 2^{32} memory [67]. With sophisticated improvements concerning the Super-Sbox method, the complexity has been reduced down to 2^{182} computations and 2^{37} memory for the 8-round version and down to 2^{118} computations and 2^{38} memory for the 7-round version [70].

It is interesting to note that the full number of rounds in the internal permutation can be "reached" with the Super-Sbox analysis, as is the case with other SHA-3 candidates such as Grøstl [67]. Such analysis reflects the simplicity of design and, depending on the details of the analysis, it helps to measure the inherent differential properties of these algorithms. However, when designing ECHO the goal was not to build a seemingly-ideal 2048-bit permutation, but rather to build a secure 256-bit or 512-bit hash function. The large internal permutation is an important component of the compression function, but then so is the final convolution that provides a major contribution to the diffusion in ECHO. As a consequence, it is important to analyse the compression function in its entirety rather than to consider the internal permutation in isolation. Indeed, this is the appropriate position since the attendant proofs of security for the operational mode of a hash function require indistinguishability of the compression function.

Finally, the fifth type of analysis has proved to be efficient when studying the hash function security, or distinguishers for the compression function. In [73], the author describes a distinguisher for the 256-bit version of ECHO reduced to 5 rounds with 2^{96} computations and 2^{64} memory, while a collision attack on 4 rounds requiring 2^{64} computations and memory is also given. Interestingly, these attacks are not directly applicable to the corresponding ECHO compression function because they exploit the final truncation phase at the end of the hash function process. Therefore, these attacks are not showing weaknesses on reduced versions of ECHO, but rather that the incorporation of an output function before the final truncation phase would be an interesting feature.

In [73] is also described a distinguishing attack against the 7-round reduced 256-bit version of ECHO compression function with 2^{152} computations and 2^{64} memory. The same attack applies for the 512-bit version with a complexity of 2^{162} computations and 2^{64} memory.

All the results are summarized in Table 1 with ongoing work showing similar complexities for ECHO-SP. Note that in general the single-pipe versions of ECHO compression functions are harder to distinguish because the attack complexity can not be compared with the generic complexity for the double-pipe security anymore. In conclusion, there have been a lot of cryptanalysis conducted on ECHO thanks to the simplicity of its design that positively encourages analysis. The ECHO security margin is still perfectly fine, even after the application of very sophisticated methods that broke many SHA-3 candidates in the past years. Moreover, unlike other AES-based SHA-3 candidates, both ECHO compression and hash functions remain secure. Finally, even if aiming at double-pipe security is much harder, our design philosophy has always been one of *security first*; just as we resisted the temptation to compromise security with a single-pipe design, we would prefer a solid margin for security in the compression function of ECHO.

9.5 New Analysis on the AES Itself

During 2009 there was much excitement about some recent advances in the analysis of the AES. In several pieces of work [13, 11] attacks on different configurations of the AES were highlighted. On closer examination, however, it is clear that this analysis focuses on the key schedule of the AES and not on the properties of the encryption routine. Indeed, we see very similar results beginning to appear as was the case with SHA-1; the very light AES-256 key schedule allows the attacker to build what are, in effect, local collisions during encryption as a way to control and limit the propagation and spread of a difference.

Rather than being a threat to ECHO, this work actually gives very strong confirmation for our design philosophy; in ECHO there is no key schedule and once the compression function computation has begun there is no opportunity for the attacker to control or limit any avalanche of change. Instead, this work on the key schedule of the AES provides yet more evidence for the difficulty of designing a good key schedule. Instead it is the SHA-3 submissions that use a

Table 1: A summary of best cryptanalysis results for ECHO.

target	rounds	comp. req.	mem. req.	type	ref.
ECHO internal	7	2^{118}	2^{38}	distinguisher	[70]
permutation	8	2^{182}	2^{37}	distinguisher	[70]
ECHO (256-bit)	3/8	2^{64}	2^{32}	free-start collision[1]	[67]
comp. function	7/8	2^{152}	2^{64}	distinguisher[2]	[73]
ECHO-SP (256-bit)	3/8	2^{64}	2^{32}	semi-free-start collision	[67]
comp. function	6/8	2^{152}	2^{64}	distinguisher	[73]
ECHO (512-bit)	2.5/10	2^{32}	2^{32}	free-start collision[1]	[67]
comp. function	7/10	2^{162}	2^{64}	distinguisher	[73]
ECHO-SP (512-bit)	3/10	2^{64}	2^{32}	free-start collision[1]	[67]
comp. function	7/10	2^{152}	2^{64}	distinguisher[2]	[73]
ECHO (256-bit)	4/8	2^{64}	2^{64}	collision	[73]
hash function	5/8	2^{96}	2^{64}	distinguisher	[73]
ECHO-SP (512-bit)	4/10	2^{64}	2^{64}	collision	[73]
hash function	5/10	2^{96}	2^{64}	distinguisher	[73]

key schedule that are more threatened by this kind of advance.

9.6 Further Notes

In this section, we briefly outline some other scenarios that we have considered in our analysis of ECHO.

First one can remark that the main internal primitive used in ECHO is an S-box composed of two AES rounds. An attacker could pre-compute and store the S-box outputs for any input and for any counter or salt value. This would cost him $2^{128\times 3} = 2^{384}$ operations and memory. Then, when the attacker tries to mount an attack, he might use these huge tables to accelerate his basic operation cost, such as finding an input that maps to a particular output after application of the function BIG.SubWords. While we have observed no attack that can be directly derived from this observation, it may remain a useful tool for future analysis of the security of ECHO.

Secondly we have considered the inherent symmetry of the AES cipher. When an AES internal state is filled with identical column values then it is well-known that an application of the AES round will maintain this property. In the case of the AES block cipher, this is prevented by the asymmetry in

[1] In a chosen-salt setting, this can be improved to a 3-round reduced semi-free-start collision attack requiring 2^{96} computations and 2^{32} memory [67].

[2] In a chosen-salt setting, this can be improved to a 7-round reduced distinguishing attack requiring 2^{107} computations and 2^{64} memory [73].

the key schedule. However, in our case the key schedule has been removed and there might be the concern that such a vulnerability exists for ECHO. In fact, this highlights the primary purpose of the internal counter κ which is different for every BIG.SUBWORDS call. This helps resist efforts to maintain symmetry within both the inner- and the outer-AES components of ECHO.

10 The Domain Extension Algorithm

Our choice for the domain extension algorithm in ECHO may be a little surprising. The HAIFA framework [8, 9] is often mentioned as an alternative to the so-called *double-pipe* strategy [52], in which the chaining variable is twice the size of the hash output HSIZE. However we have opted to combine the two. We see numerous advantages in doing this, though the increased assurance we get is somewhat offset by the performance implications.

The HAIFA framework, its security, and its practical advantages are already covered elsewhere [8, 9]. Here we merely highlight some of the advantages that made this approach an attractive choice for us.

Apart from its simplicity, the HAIFA framework has many nice features. A variable output hash size can be supported in a secure way since both initialisation and padding depend on the output length. Varying initialization helps to avoid hash outputs being related when truncation is used to get different hash output lengths. Length-dependent padding also helps avoid birthday paradox-based techniques being applied to the internal state to get related hash outputs. In addition HAIFA has built-in support, via a SALT, for a family of hash functions that allows us to place the hash function within certain theoretical models. We can also provide *randomized hashing* and, when the salt is unknown in advance, pre-computation attacks on the compression function or hash function are avoided.

The HAIFA framework also has several practical and security advantages. First, we inherit the advantages of Merkle-Damgård such as the *one-pass online* hashing property, which means that we are only required to keep a fixed amount of memory for hashing each message block. On the security side, HAIFA allows us to maintain the Merkle-Damgård proof regarding collision-resistance, namely, that if the compression function is collision-resistant then the hash function is also collision-resistant.

As well as a salt, the HAIFA framework includes a bit counter as an input to the compression function. This is a useful counter-measure and hinders an attacker from exploiting any fixed points that might be found in the compression function. When using Merkle-Damgård with a traditional compression function, then once a fixed point is found, it is straightforward to concatenate message blocks so as to keep the output of the compression functions looping as often as required. Dean [21] showed that this could be used to devise a second-preimage attack and while this kind of attack might have little practical impact because of the long messages required, many commentators would prefer to avoid this possibility. In the case of HAIFA, the counter prevents an attacker

from exploiting such a fixed point since the bit counter that is input to the compression function changes at each iteration thereby disrupting the loop in the computation. Note however that Davies-Meyer-like fixed points are avoided in the compression function of ECHO.

The counter is also useful in avoiding message extension attacks, which is sometimes viewed as a threat to certain *message authentication code* (MAC) constructions. For instance, the construction $\text{MAC}_k(m) = \text{HASH}(k\|m)$ is known to be secure for an ideal hash function, but not with a Merkle-Damgård construction, even when instantiated with an ideal compression function. In HAIFA, however, the number of bits hashed so far is input to the compression function and this helps prevent extension attacks on the highlighted MAC construction.

One final advantage of the HAIFA framework is the improved resistance to a spate of generic attacks that have been applied to the Merkle-Damgård construction; these include multi-collisions [37], long second preimage attacks [43], and herding attacks [42]. On this issue, it is worth paying a bit more attention to exactly what HAIFA provides. Regarding multi-collisions attacks, for instance, the HAIFA framework only prevents pre-computation by an adversary thanks to the SALT. Therefore if the effort to find an internal collision is, say, 2^a operations, then a 2^k-multi-collision can be found with $k \times 2^a$ operations. There is therefore no real gain over Merkle-Damgård, and this is exploited in applications of the herding attack [42]. And while herding attacks are made more complex by an unknown salt, if the salt is chosen by the attacker then the attacks will still apply.

It is for this reason that we decided to complement the HAIFA approach with the double-pipe proposal [52]. We feel that while a salt has a very useful role it is only a partial solution to structural problems with vanilla Merkle-Damgård. Further, when a hash function requires n-bit security the salt should be at least $\frac{n}{2}$ bits long without the double-pipe. For hash outputs such as 512 bits, as required by NIST, this begins to have an impact on the simplicity of the underlying design.

Thus the SALT in ECHO is primarily used to support randomized hashing. We then complement the added functionality of HAIFA with a double-length chaining variable. So for a HSIZE-bit hash output, the chaining variable in ECHO is at least $2 \times$ HSIZE bits long. In our view it is the double-sized chaining variable that protects us from recent structural attacks on Merkle-Damgård and we get immediate resistance to herding attacks and to multi-collision attacks. If no collision attack on the compression function with complexity less than 2^{HSIZE} operations can be found, then no multi-collision attack applies to ECHO. However, even if a collision attack on the compression function in 2^{HSIZE} operations were to be found, while our resistance to multi-collisions would default to that offered by the HAIFA construction, collision-resistance of ECHO would still be maintained, as is shown by the proof to the Merkle-Damgård construction. We also get increased resistance to attacks that might seek to exploit any fixed points in the compression function. And, since only half the information in the final chaining variable is present in the hash value, we obtain a natural resistance to extension attacks.

For more background and details the interested reader is referred to Section 4 of [9] for more study of the HAIFA framework, and to [52] for a security analysis of the double-length chaining value.

Acknowledgments

We would like to thank Côme Berbain for fruitful discussions at the early stage of this work, Jonathan Etrog for his support during the design of ECHO and Eric Brier for his improvements of the security bounds. We are very grateful to Doug Whiting for his thoughtful comments, and for having brought to our attention some minor issues in the way the internal counter is described in earlier versions of this documentation.

Note

Olivier now works elsewhere.

References

[1] M. Bellare. New Proofs for NMAC and HMAC: Security Without Collision Resistance. In C. Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer-Verlag, 2006.

[2] M. Bellare, R. Canetti and H. Krawczyk. Keying Hash Functions for Message Authentication. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

[3] R. Benadjila, O. Billet, S. Gueron, and M.J.B. Robshaw. The Intel AES Instructions Set and the SHA-3 Candidates. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 162–178. Springer-Verlag, 2009. Available at `http://crypto.rd.francetelecom.com/sha3/AES/paper/`.

[4] D. Bernstein. Cache Timing Attacks on AES. Available at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[5] D. Bernstein and P. Schwabe. New AES Software Speed Records. In D.R. Chowdhury, V. Rijmen, and A. Das, editors, *Progress in Cryptology – INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer-Verlag, 2008. Available at `http://cr.yp.to/papers.html#aesspeed`.

[6] J-L. Beuchat, E. Okamoto, and T. Yamazaki. A Compact FPGA Implementation of the SHA-3 Candidate ECHO. Cryptology ePrint Archive, Report 2010/364. Available at `http://eprint.iacr.org/2010/364.pdf`.

[7] E. Biham and R. Chan. Near-Collisions of SHA-0. In M. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer-Verlag, 2004.

[8] E. Biham and O. Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Presented at Second NIST Cryptographic Hash Workshop, August 24-25, 2006. Available at `http://csrc.nist.gov/groups/ST/hash/documents/DUNKELMAN_NIST3.pdf`.

[9] E. Biham and O. Dunkelman. A Framework for Iterative Hash Functions: HAIFA. ePrint archive, 2007. Available at `http://eprint.iacr.org/2007/278.pdf`.

[10] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer Verlag, 1993.

[11] A. Biryukov, O. Dunkelman, N. Keller, D. Khovratovich, and A. Shamir. Key Recovery Attacks of Practical Complexity on AES-256 Variants with

up to 10 Rounds. In H. Gilbert, editor, *Advances in Cryptology – Eurocrypt 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 299–319. Springer-Verlag, 2010.

[12] A. Biryukov and D. Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2009.

[13] A. Biryukov, D. Khovratovich, and I. Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In S. Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 231–249. Springer-Verlag, 2009.

[14] F. Chabaud and A. Joux. Differential Collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.

[15] J. Daemen and V. Rijmen. The Design of Rijndael: AES—The Advanced Encryption Standard. Springer, 2002.

[16] J. Daemen and V. Rijmen. Plateau characteristics. In *Information Security*, IET , vol. 1, no. 1, pages 11-17, March 2007.

[17] J. Daemen and V. Rijmen. AES Proposal: Rijndael, version 2, September 3, 1999.

[18] J. Daemen and V. Rijmen. Two-Round AES Differentials. Cryptology ePrint Archive, Report 2006/039, 2006. `http://eprint.iacr.org/`.

[19] J. Daemen and V. Rijmen. Understanding Two-Round Differentials in AES. In Roberto De Prisco and Moti Yung, editors, *SCN*, volume 4116 of *Lecture Notes in Computer Science*, pages 78–94. Springer, 2006.

[20] I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1989.

[21] R.D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[22] H. Demirci and A.A. Selçuk. A Meet-in-the-Middle Attack on 8-Round AES. In K. Nyberg, editor, *Fast Software Encryption - FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 116-126. Springer-Verlag, 2008.

[23] I. Dinur and A. Shamir. Cube Attacks on Tweakable Black Box Polynomials. In A. Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 278–299. Springer-Verlag, 2009. Available at `http://eprint.iacr.org/2008/385`.

[24] ECRYPT. State of the Art in Hardware Architectures (D.VAM.2). September 5, 2005. Available at `http://www.ecrypt.eu.org/documents/D.VAM.2-1.0.pdf`.

[25] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong Authentication for RFID Systems Using the AES Algorithm. In M. Joye and J.-J. Quisquater, editors, *CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 357–370. Springer-Verlag, 2004.

[26] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. In B. Schneier, editor, *Fast Software Encryption FSE 00*, volume 1978 of *Lecture Notes in Computer Science*, pages 213230. Springer-Verlag, 2000.

[27] H. Gilbert and M. Minier. A Collision Attack on 7 Rounds of Rijndael. In *Proceedings of the third AES Candidate Conference*, pages 230–241, 2000.

[28] H. Gilbert and T. Peyrin. Super S-box cryptanalysis: Improved attacks for AES-like permutations. In S. Hong and T. Iwata, editors, *Fast Software Encryption FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 365-383. Springer-Verlag, 2010.

[29] S. Gueron. Advanced Encryption Standard (AES) Instructions Set. Intel Corporation White Paper, April 2008. Available at `http://software.intel.com`.

[30] G. Hachez, F. Koeune, and J.-J. Quisquater. cAESar Results: Implementation of Four AES Candidates on Two Smart Cards. In *Proceedings of the second AES Candidate Conference*, pages 95–108, 1999.

[31] A. Hodjat, D. Hwang, B.-C. Lai, K. Tiri, and I. Verbauwhede. A 3.84 Gbits/s AES crypto coprocessor with modes of operation in a 0.18μm CMOS technology. In *Proceedings of the 15th ACM Great Lakes Symposium on VLSI 2005*, pages 60–63. ACM, 2005.

[32] A. Hodjat and I. Verbauwhede. Area-Throughput Trade-Offs for Fully Pipelined 30 to 70 Gbits/s AES Processors. In *IEEE Trans. Computers*, vol. 55, no. 4, pages 366–372, 2006.

[33] S. Hong, S. Lee, J. Lim, J. Sung, D.H. Cheon, and I. Cho. Provable Security against Differential and Linear Cryptanalysis for the SPN Structure. In B. Schneier, editor, *Fast Software Encryption FSE 00*, volume 1978 of em Lecture Notes in Computer Science, pages 273-283. Springer-Verlag, 2000.

[34] K. Ideguchi and E. Tischhauser. Private communication, June 2010.

[35] Intel Corporation. Intel Advanced Encryption Standard (AES) Instructions Set. White paper available at `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set`.

[36] Intel Corporation. Intel Software Development Emulator. Freely available for download at `http://software.intel.com/en-us/articles/intel-software-development-emulator`.

[37] A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M.K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer-Verlag, 2004.

[38] A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 2007.

[39] C. Jutla. Private communication, April 2010.

[40] G. Keating. Performance Analysis of AES Candidates on the 6805 CPU Core. In *Proceedings of the second AES Candidate Conference*, pages 109–114, 1999.

[41] L. Keliher and J. Sui. Exact Maximum Expected Differential and Linear Cryptanalysis for Two-Round Advanced Encryption Standard. In *IET Information Security*, vol. 1, no. 2, pages 53-57, 2007.

[42] J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer-Verlag, 2006.

[43] J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer-Verlag, 2005.

[44] L.R. Knudsen. Truncated and Higher Order Differentials. In B. Preneel, editor, *Fast Software Encryption – FSE '94*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, 1994.

[45] L.R. Knudsen, C. Rechberger, and S.S. Thomsen. The Grindahl Hash Functions. In A. Biryukov, editor, *Fast Software Encryption – FSE '06*, volume 4593 of *Lecture Notes in Computer Science*, pages 39–57. Springer-Verlag, 2006.

[46] L.R. Knudsen and V. Rijmen. Known-Key Distinguishers for Some Block Ciphers. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer-Verlag, 2007.

[47] M. Kounavis and S. Gueron Vortex: A New Family of One Way Hash Functions based on Rijndael Rounds and Carry-less Multiplication. Submission to NIST. 2008. Available at `http://eprint.iacr.org/2008/464.pdf`.

[48] X. Lai, J.L. Massey, and S. Murphy. Markov Ciphers and Differential Cryptanalysis. In D. Davies, editor, *Proceedings of EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 1991.

[49] M. Lamberger, F. Mendel, C. Rechberger, V. Rijmen, and M. Schläffer. Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 126–143. Springer-Verlag, 2009.

[50] L. Lu, M. O'Neill, and E. Swartzlander. ASIC Evaluation of ECHO Hash Function. *IEEE International System-on-Chip Conference – SOCC Conference 2009.*

[51] L. Lu, M. O'Neill, and E. Swartzlander. Hardware Evaluation of SHA-3 hash Function Candidate ECHO. *The Claude Shannon Workshop on Coding and Cryptography.* May 2009.

[52] S. Lucks. A Failure-Friendly Design Principle for Hash Functions. In B.K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer-Verlag, 2005.

[53] F. Mendel, T. Peyrin, C. Rechberger, and M. Schläffer. Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In M.J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography – SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 2009.

[54] F. Mendel, C. Rechberger, M. Schläffer, and S.S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In O. Dunkelman, editor, *Fast Software Encryption – FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer-Verlag, 2009.

[55] R.C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer-Verlag, 1989.

[56] S. Murphy and M.J.B. Robshaw. Essential Algebraic Structure within the AES. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2002.

[57] National Institute of Standards and Technology. FIPS 46-3: Data Encryption Standard, October 1993. Available at `csrc.nist.gov`.

[58] National Institute of Standards and Technology. FIPS 197: Advanced Encryption Standard, November 2001. Available at `csrc.nist.gov`.

[59] National Institute of Standards and Technology. FIPS 186-2: Digital Signature Standard, January 2000. Available at `csrc.nist.gov`.

[60] National Institute of Standards and Technology. FIPS 198-1: The Keyed-Hash Message Authentication Code (HMAC), July 2008. Available at `csrc.nist.gov`.

[61] National Institute of Standards and Technology. SP 800-56A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), March 2007. Available at `csrc.nist.gov`.

[62] National Institute of Standards and Technology. SP 800-90: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised), March 2007. Available at `csrc.nist.gov`.

[63] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. Available at `csrc.nist.gov`.

[64] K. Ohkuma, H. Shimizu, F. Sano, and S. Kawamura. Security Assessment of Hierocrypt and Rijndael against the Differential and Linear Cryptanalysis. In *Proceedings of the 2nd NESSIE workshop*, 2001. Available at `http://eprint.iacr.org/2001/070`.

[65] D.A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2006.

[66] T. Peyrin. Cryptanalysis of Grindahl. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer-Verlag, 2007.

[67] T. Peyrin. Improved Differential Attacks for ECHO and Grostl. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 370–392. Springer-Verlag, 2010. Available at `http://eprint.iacr.org/2010/223.pdf`.

[68] T. Peyrin, H. Gilbert, F. Muller and M.J.B. Robshaw. Combining Compression Functions and Block Cipher-based Hash Functions. In X. Lai, editor, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 315–331. Springer-Verlag, 2006.

[69] V. Rijmen. Cryptanalysis and design of iterated block ciphers. Doctoral Dissertation, K.U Leuven. October 1997.

[70] Y. Sasaki, Y. Li, L. Wang, K. Sakiyama and K. Ohta. Low Complexity Distinguisher for ECHO-Permutation. Second ECRYPT Hash Function Retreat. May 2010.

[71] C. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28:656–715, 1949.

[72] T. Smalley. Westmere is Nehalem's Successor. News article on `http://www.bit-tech.net`, September 19, 2007.

[73] M. Schläffer. Subspace Distinguisher for 5/8 Rounds of the ECHO-256 Hash Function. Cryptology ePrint Archive, Report 2010/321. Available at `http://eprint.iacr.org/2010/321.pdf`.

[74] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 1-18. Springer-Verlag, 2005.

[75] X. Wang, Y.L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17-36. Springer-Verlag, 2005.

[76] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19-35. Springer-Verlag, 2005.

[77] X. Wang, H. Yu, and Y.L. Yin. Efficient Collision Search Attacks on SHA-0. In V. Shoup, editor *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2005.

[78] J.W. Bos and D. Stefan Performance Analysis of the SHA-3 Candidates on Exotic Multi-Core Architectures. *CHES 2010*.

[79] eBASH. ECRYPT Benchmarking of All Submitted Hashes. Available at `http://bench.cr.yp.to/ebash.html`.

A Performance Figures

The following table gives the latest implementation results (cycles/byte) for ECHO on a wide range of platforms. For ECHO-SP the anticipated performance can be obtained by multiplying figures in the first column by 0.86 and those in the second by 0.67.

Platform	$128 \leq$ HSIZE ≤ 256	$257 \leq$ HSIZE ≤ 512
Core i5 (`x86`)*	8.3	15.3
Core i5 (`amd64`)*	6.8	12.6
Core i7 (`x86`)	30.6	56.6
Core i7 (`amd64`)	25.7	47.5
Core 2 (`x86`)	32.5	59.7
Core 2 (`amd64`)	28.3	50.3
Phenom I/II (`x86`)	34.9	64.6
Phenom I/II (`amd64`)	28.4	52.5
Opteron K8 (`x86`)	32.3	59.0
Opteron K8 (`amd64`)	28.5	51.7
Athlon K8 (`x86`)	35.3	65.3
Athlon K8 (`amd64`)	28.4	52.5
Athlon K7 XP (`x86`)	40.7	75.3
Athlon ThunderBird (`x86`)	41.6	76.0
Pentium 4 (`x86`)	42.5	78.6
Pentium III (`x86`)	45.5	84.2
Pentium II (`x86`)	47	87
Pentium M (`x86`)	40.6	75.1
Atom (`x86`)	84.1	158.4
PowerPC G3†	66	122.1
IBM Power4†	59.0	101.0
Itanium II$^\alpha$	33.9	60.9.0
SPARC‡	95	175.8
ARM$^\alpha$	147	272
Cell Broadband (PlayStation 3)¶	29.6	-
Nvidia GPU (GTX 295)¶	0.85	-

* Using AES-NI instructions set.
¶ Implementations described in [78].

All figures correspond to optimized assembly, except:

† optimized C code, further improvements are possible with assembly, especially on G4 and G5 series where Altivec vector instructions set is enabled (a 30% performance increase is expected).

‡ non optimized assembly, some improvements might be possible.

$^{\alpha}$ non optimized C, significant improvements are possible with C and assembly.

Test configurations were as follows. The compilers were gcc 4.4 or 4.3 and the OS were Debian GNU/Linux lenny or Ubuntu karmic koala –`i386` and `amd64`– except for ARM which was running Linux 2.6.15 embedded with gcc 3.4.3.

- Pentium 4: Pentium 4 Northwood @ 2 GHz and Xeon Prestonia @ 2 GHz
- Pentium III: Pentium III Coppermine @ 960 MHz
- Pentium II: Pentium II Klamath @ 300 MHz
- Pentium M: Pentium M @ 1.6 GHz
- Core 2: Core 2 Duo 6600 @ 2.4 GHz and Xeon E5420 @ 2.5 GHz
- Core i7: Core i7 920 @ 2.67 GHz
- Core i5: Core i5 M540 @ 2.53 GHz
- Opteron: AMD Opteron 850 @ 2.2 GHz
- Athlon K8: Athlon 64 3200+ @ 2 GHz
- Athlon K7: Athlon XP 2800+ @ 2 GHz
- Atom: Intel Atom N270 @ 1.6 GHz
- PowerPC G3: PowerPC 750FX @ 800 MHz
- SPARC: SPARC IIi Sabre @ 267 MHz
- ARM: ARM926EJ-Sid @ 500 MHz

The figures for the Athlon ThunderBird, the Phenom I/II, the IBM Power4 and the Itanium II have been extracted from the eBASH [79] results. The figures for the multi-threaded Cell and GPU platforms are reported in [78].