Efficient Hashing using the AES Instruction Set

Joppe W. Bos[1] and Onur Özen[1] and Martijn Stam[2]

[1] Laboratory for Cryptologic Algorithms, EPFL, Station 14, CH-1015 Lausanne, Switzerland
{joppe.bos, onur.ozen}@epfl.ch
[2] Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom
stam@cs.bris.ac.uk

Abstract. In this work, we provide a software benchmark for a large range of 256-bit blockcipher-based hash functions. We instantiate the underlying blockcipher with AES, which allows us to exploit the recent AES instruction set (AES-NI). Since AES itself only outputs 128 bits, we consider double-block-length constructions, as well as (single-block-length) constructions based on RIJNDAEL-256. Although we primarily target architectures supporting AES-NI, our framework has much broader applications by estimating the performance of these hash functions on any (micro-)architecture given AES-benchmark results. As far as we are aware, this is the first comprehensive performance comparison of multi-block-length hash functions in software.

1 Introduction

Historically, the most popular way of constructing a hash function is to iterate a compression function that itself is based on a blockcipher (this idea dates back to Rabin [49]). This approach has the practical advantage—especially on resource-constrained devices—that only a single primitive is needed to implement two functionalities (namely encrypting and hashing). Moreover, trust in the blockcipher can be conferred to the corresponding hash function. The wisdom of blockcipher-based hashing is still valid today. Indeed, the current cryptographic hash function standard SHA-2 and some of the SHA-3 candidates are, or can be regarded as, blockcipher-based designs. In the 1980s, several methods were proposed with an eye towards using the then-standard Data Encryption Standard (DES) as the underlying primitive [40,28,14]. At present, the contemporary Advanced Encryption Standard (AES [41]) is a more obvious choice instead.

A well-studied class of blockcipher-based hash functions are the PGV hash functions (after Preneel, Govaerts and Vandewalle [48]), encompassing Davies–Meyer (DM) and Matyas–Meyer–Oseas (MMO) as special cases. When based on a blockcipher operating on n-bit blocks with k-bit keys, these functions compress k bits per blockcipher call and they output an n-bit digest. The PGV hash functions are simple (low overhead) and are provably secure in the ideal-cipher model [10]. Yet they suffer from one major drawback: in order to achieve an acceptable level of collision resistance, one needs a primitive operating on more than 160 bits. This rules out most existing blockciphers, *including* AES (which operates on 128-bit blocks only).

As a remedy, *double-block-length* and more generally *multi-block-length* compression and hash functions were introduced. These are compression functions outputting an rn-bit digest (for an integer $r \geq 2$, $r = 2$ for the double-block-length case), even though they are based on a primitive operating only on n-bit blocks. The longer digest size opens up the possibility of collision resistance of 2^n time (primitive evaluations) even when using a relatively small primitive. Today, there is truly a wealth of suitable blockcipher-based constructions to choose from and Table 1 gives an overview of the constructions we consider. We do not consider *all* possibilities, for instance we omit versions of GRØSTL, JH or SPONGE based on RIJNDAEL-256. As can be seen, we instantiate the underlying blockcipher with either AES-128, AES-256 or RIJNDAEL-256. The latter option allows us to consider single-block-length constructions achieving a 256-bit digest (using an AES-related primitive).

Our choice of constructions includes several different design ideas and paradigms. For years, most cryptographic hash function designs revolved around the same principle [49,40,14]: the Merkle-Damgård paradigm. In this cascaded mode of operation, the main focus is to construct a secure and efficient compression function; these properties are then inherited by the overall hash function. Later constructions started to deviate from this paradigm, for instance by some form of strengthening [35,7] or by only targeting security in the iteration [10,6].

A more fundamental design shift occurred in the way the blockcipher itself is used. A blockcipher, operating on n bits with a k-bit key, can already be regarded as a compressing primitive itself. This facilitates the transformation into a proper compression function, but a disadvantage of using a blockcipher this way is that it requires frequent re-keying, which tends to be expensive (see Section 2 for details). For this reason, there have been substantial efforts in recent years to design permutation-based compression functions. Obviously, given a blockcipher one can construct a permutation by simply fixing the key (we focus on permutations with either $n = 128$ or 256 bits).

While the design and analysis of multi-block-length compression functions have garnered significant attention, the focus in the literature seems squarely at security evaluation and theoretical notions of efficiency (expressed as the ratio of message blocks compressed per blockcipher call). Although the latter is known to give only a coarse indication of real-life efficiency, actual performance benchmarks, in hard- or software, are normally left as future work. (A notable exception is the work by Bogdanov et al. [11], who provide hardware benchmarks for some multi-block-length compression functions in hardware using the lightweight blockcipher PRESENT as the underlying building block.)

Our Contribution. In this work we bring together the mainly theoretical world of compression function designs with the practical demand of fast implementations. Instantiating the blockcipher-based primitives with AES-128, AES-256, respectively RIJNDAEL-256 (and their fixed-key versions to build permutations), we obtain hash functions with a fixed 256-bit digest size. Apart from three constructions (LANE*, LUFFA* and KNUDSEN–PRENEEL) all constructions have known proofs of security in the ideal-cipher model (we refer to the full version of this work [12] for a more detailed discussion on the security of our target constructions). The former SHA-3 candidates LANE* and LUFFA* do not have security proofs, neither for collision resistance nor for preimage resistance. We include them in our benchmark (with different building blocks)

Table 1. A brief taxonomy of the schemes considered. The number of rounds N_r is 10 for AES-128 and 14 for AES-256 and RIJNDAEL-256.

Blockcipher (dimensions)	**Variable-key Constructions**	**Fixed-key Constructions**
AES-128 $(k = 128, n = 128)$	MDC-2, MJH, PEYRIN ET AL.(I)	LP362
AES-256 $(k = 256, n = 128)$	ABREAST-DM, HIROSE-DBL, KNUDSEN–PRENEEL, MJH-DOUBLE, QPB-DBL, PEYRIN ET AL.(II)	n.a.
RIJNDAEL-256 $(k = 256, n = 256)$	DAVIES–MEYER	LANE*, LUFFA*, LP231, SHRIMPTON–STAM

to illustrate their performance capabilities; KNUDSEN–PRENEEL is another exception where a good collision resistance lower bound is still an open problem.

To the best of our knowledge, this is the first overview of software implementations of the most studied and influential blockcipher- and permutation-based compression and hash functions. The target designs (see also Table 1) have been implemented and measured on an Intel Core i5 650 (3.20GHz) using C intrinsics to implement the various SS(S)E{2,3,4} and the recent AES instruction set (AES-NI) extensions [18,19]. Although measured on a single Intel architecture with AES-NI we expect the relative performance obtained to be representative for other Intel architecture families with AES-NI support as well. The Intel compiler version 12.0.0 and GNU Compiler Collection (gcc) version 4.4.3 were used for code compilation. For each design we performed specific optimizations to fully exploit AES-NI. The details are discussed in Section 3, with Table 3 providing a summary of our findings.

The Choice for AES. Our choice for AES (and RIJNDAEL-256) is a natural one: it is the official US and de facto world standard blockcipher. AES' prime position has led to a large body of research on AES, both on its security and implementation. Consequently, AES runs very fast in hard- and software, making AES an obvious choice from a performance perspective. The deal is sweetened further by the recent introduction of AES-NI. Indeed, as reported in [19], one can achieve significant speed using the new instruction set (e.g. up to 1.3 cycles/byte on a single core Intel Core i7-980X for AES-128 in parallel modes). To benefit from synergy with AES and AES-NI in particular, several SHA-3 candidates were instantiated by using some of AES' components as well (e.g. the AES round function), which was later demonstrated to indeed lead to fast hashing [2]. Our goal here is to investigate the potential of AES-NI for fast hashing even further by focusing on well-known blockcipher- and permutation-based (compression function) designs that can be instantiated with AES (or more generally RIJNDAEL).

From a security perspective, AES remains unbroken as a blockcipher in the standard setting. It has survived many years of cryptanalysis and a practical break of this cipher would have a significant impact on the cryptographic landscape. Nonetheless, our choice for AES will not be without detractors as a consequence of recent related-key attacks on AES [8,9]. The theoretical ramifications of a related-key attack to hash-function security are still unclear. Any serious related-key attack undermines the assumption that the blockcipher behaves 'ideally', but this need not lead to any deviant behaviour of the hash function itself (especially if its proof uses the weaker unforgeable-cipher model). Of course, in practice a related-key attack is often underpinned by some other (well-defined) weakness and exploiting this weakness directly (ignoring the derived related-key attack) might be more fruitful when attacking the hash function. For instance, Khovratovich [24, Corollary 2] states unambiguously that "AES-256 in the Davies–Meyer hashing mode leads to an insecure hash function" but later provides solace by remarking that it is not known how the techniques used against AES-256 in Davies–Meyer mode can be modified to attack double-block-length constructions (the focus of this paper).

As a final remark, the timings we obtain evidently depend strongly on the number of rounds used by AES. While one can argue that the number of rounds used should be fine-tuned for each of the hash functions (increasing or decreasing, depending on the perceived security margin), we believe that using AES as is will give the cleanest comparison (and any changes might be considered contentious).

2 Preliminaries

The Blockciphers AES-128, AES-256 and RIJNDAEL-256. AES is a member of the RIJNDAEL blockcipher suite [41]. It was standardized by the US National Institute of Standards and Technology (NIST) after a public competition similar to the one currently ongoing for SHA-3 [43]. AES operates on an internal state of 128 bits while supporting 128-, 192-, and 256-bit keys. The internal state is organized in a 4×4 array of 16 bytes, which is transformed by a round function N_r times. The number of rounds is $N_r = 10$ for the 128-bit key, $N_r = 12$ for the 192-bit key, and $N_r = 14$ for the 256-bit key variants. In order to encrypt, the internal state is initialized, then the first 128-bits of the key are XORed into the state, after which the state is modified $N_r - 1$ times according to the round function, followed by a slightly different final round (for the exact details see the AES specification [41]). The larger state variant of AES, RIJNDAEL-256, operates almost in the same way with a state size of 256 bits, a 256-bit key and $N_r = 14$ rounds.

Nine years after becoming the symmetric encryption standard, the only theoretical attack on the full AES is restricted to the related key scenario and even then applies only to the 192-bit [8] and 256-bit key versions [8,9]. So far no theoretical attacks on all rounds of AES-128 are known. More cryptanalytic success has been achieved by using the characteristics from the actual implementation of AES, e.g. cache attacks [60,3] can recover an AES key in only 65 milliseconds (Tromer et al. [60] give a more detailed survey of side-channel attacks against AES). However, side-channel analysis is far less of a concern for hash functions (except for MACs based on hash functions, such as HMAC) and we will blithely ignore the issue in this paper.

The AES Instruction Set (AES-NI). In the last decade, use of the *single instruction, multiple data* (SIMD) paradigm has become a general trend in computer architecture design. It enhances the speed of software implementations by offloading the computational work to special units which operate on larger data types, improving overall throughput. In 1999, Intel introduced the streaming SIMD extensions (SSE), a SIMD instruction set extension to the x86 architecture. One of the latest additions to these extensions is the AES instruction set [18,19] available in the 2010 Intel Core processor family based on the 32nm Intel micro-architecture named Westmere. This instruction set will also be supported by AMD in their next-generation CPU "Bulldozer". (Note that previously several instruction set extensions have been suggested towards improving the performance of AES [58,5,59].) AES-NI does not only increase the performance of AES (as well as any version of RIJNDAEL) but also runs in data-independent time and by avoiding the use of any table lookups the aforementioned cache attacks are avoided. This instruction set consists of six new instructions. At the same time, a new instruction for performing carry-less multiplication is released in the `CLMUL` instruction set extension. We can summarize the new instructions as follows [18,19,20]:

- `AESENC` and `AESDEC` perform a single round of encryption, resp. decryption.
- `AESENCLAST` and `AESDECLAST` perform the last round of encryption resp. decryption.
- `AESKEYGENASSIST` is used for generating the round keys used for encryption.
- `AESIMC` is used for converting the encryption round keys to a form usable for decryption using the Equivalent Inverse Cipher.
- `PCLMULQDQ` performs carry-less multiplication of two 64-bit operands to an 128-bit output.

Many of the constructions targeted in this paper require the computation of more than one call to a blockcipher (with or without a fixed-key). If these two or more calls can be run concurrently (while possibly sharing the key expansion), a performance gain can be expected as AES round instructions are pipelined and can be dispatched theoretically every 1-2 CPU clock cycles, provided that all data is available on time and there is no dependency between such subsequent calls [18]. Since the latency of a single round instruction is 5 cycles [17], running multiple independent blockciphers increases the overall throughput. The same reasoning holds when implementing a single RIJNDAEL-256 component. This sibling of AES works on an internal state of 256 bits and it is implemented using two data-independent calls to `AESENC`.

In the context of encryption, several performance results of AES exploiting AES-NI have been presented [19,20,37]. These works show that using AES-NI tends to give very fast implementations when multiple blockcipher calls can be made in parallel (incidentally, they also show that the optimal way to interleave the instructions is hard to pin down). However, they are of limited use to predict the runtimes of AES-based hash functions as re-keying tends to be far more frequent in the hashing scenario than in the encryption one. Indeed, for blockcipher-based compression functions considered in this paper, the key-scheduling needs to be performed for every compression function evaluation and that results in a significant overhead. For this reason, we start with a detailed performance overview of AES and RIJNDAEL-256 that takes re-keying into account. Table 2 contains performance details when running multiple key expansions, encryptions or a combination of the two. In order to conduct these experiments we created a code generator which, when given a number of x key expansions and y encryptions, tries different strategies to implement these functionalities. The performance numbers presented in Table 2 are an average over millions of runs. For comparison, we also included timings from Gueron's hand-crafted assembly code [19,20] as used in the Intel AES-NI sample library. (Note that, roughly speaking, our measure **1E** coincides

Table 2. Our experimental results on the encryption and key expansion routines for AES-128 (A128), AES-256 (A256) and RIJNDAEL-256 (R256). The entries show the results in cycles per operation together with the compiler, icc (i) or gcc (g), resulting in the fastest code. In the table **K** and **E** denote the key expansion and the encryption respectively. The upper part of the table shows the results of several independent key expansions and encryption operations that are called in parallel. In the lower part, **xKyE** denotes x independent key schedules followed by y independent encryptions. If $x = 1$ all encryptions use the same expanded key, if $x = y$ all encryptions use a different expanded key. For comparison, the performance details of the Intel AES-NI sample library on our platform are stated as well.

	Operation							
	1K	**2K**	**3K**	**4K**	**1E**	**2E**	**3E**	**4E**
A128	97.7 (g)	126.1 (g)	163.4 (g)	226.7 (i)	60.2 (i)	60.6 (i)	67.7 (i)	84.7 (i)
A256	125.5 (g)	147.2 (g)	202.6 (i)	287.2 (i)	82.0 (i)	83.0 (i)	93.6 (i)	113.9 (i)
R256	291.6 (g)	316.6 (g)	412.6 (g)	570.3 (i)	182.9 (i)	219.2 (g)	281.4 (i)	352.6 (g)
	1K1E	**2K2E**	**3K3E**	**4K4E**		**1K2E**	**1K3E**	**1K4E**
A128	107.4 (g)	149.2 (g)	200.0 (g)	269.9 (g)		120.1 (g)	135.3 (g)	137.8 (g)
A256	152.8 (g)	178.1 (g)	249.7 (g)	337.9 (g)		154.0 (g)	158.4 (g)	164.9 (g)
R256	285.3 (i)	407.5 (i)	620.5 (i)	867.3 (i)		312.0 (g)	373.3 (i)	463.7 (g)

	Intel AES-NI Sample Library						
	1K	**1E**	**4E**		**1K**	**1E**	**4E**
A128	98.8	62.1	79.6	A256	124.4	84.6	108.8

with AES run in a chaining mode such as CBC or CFB, whereas AES run in a parallel mode such as CTR or ECB is closer to the best time we get for **xE**, see Table 2 for the performance details).

Finite Field Arithmetic ($\mathbb{F}_{2^m}$ Full/Scalar Multiplication). Some of the compression function designs we consider require finite field multiplication, in particular in $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$. There is some freedom in how to represent the fields—the security proofs for the hash functions are independent of this choice—so we opt for the usual representation of elements in $\mathbb{F}_{2^m}$ as polynomials over $\mathbb{F}_2$ reduced modulo an irreducible polynomial of degree m. We use $x^{128} + x^7 + x^2 + x + 1$ as irreducible polynomial for $m = 128$ and $x^{256} + x^{10} + x^5 + x^2 + 1$ for $m = 256$.

Multiplication in $\mathbb{F}_{2^{128}}$ is implemented using the code examples as described in [20] in the setting of implementing the Galois counter mode. This is realized by using the new instruction `PCLMULQDQ` to implement the multiplication; this instruction calculates the carry-less product of the two 64-bit input to an 128-bit output. Note that this instruction has a latency of 12 cycles and can be dispatched every 8 cycles [17]. Hence, compared to other SSE instructions, some of which can be dispatched in pairs of three every clock cycle, this instruction might not always be the optimal choice from a performance perspective. An example where the usage of the `PCLMULQDQ` instruction might not lead to a speed-up is in the case of (field) multiplication by x. This can be computed by shifting the input one position to the left (the polynomial multiplication by x) and performing a conditional `XOR` with the reduction polynomial (depending on the bit shifted out). Unfortunately, the SSE instruction set has no bit shift operation shifting the full 128-bit vector. Shifting the two 64-bit, four 32-bit or eight 16-bit in SIMD fashion is possible but the bits shifted out locally are lost. We outline a novel approach (with the SSE instruction in parentheses) to obtain the desired result in the setting of $\mathbb{F}_{2^{128}}$ where we exploit the fact that the second largest exponent of the reduction polynomial is < 32 (which also holds in the setting of $\mathbb{F}_{2^{256}}$). Given an input A we

1. swap the two 64 bit halves of A to t (PSHUFD),
2. create a mask m (either all ones or zeros in each 64-bit half) depending if bits 63 and 127 of t are set (PCMPGTQ),
3. use m to extract the correct 64-bit parts of a precomputed constant $[1, R]$ in t (PAND),
4. shift both 64-bit parts of A left by one bit and store this in s (PSLLQ),
5. perform the reduction plus restoring the local carry bit by combining s and t (PXOR).

Here R denotes the hexadecimal representation of the reduction polynomial, excluding the term with the highest exponent, stored in a 64-bit word. Note that this computation might be sped up, depending on the setting, in the following way. Replace step 1 by a byte shuffle (PSHUFD) which moves bits 63 and 127 to bit position 95 and 31 respectively and set the other 14 bytes to zero. The resulting vector, viewed as four 32-bit signed integers, contains two 32-bit words where only the sign bit may be set. Now step 2 can be replaced by using an arithmetic right shift of 31 positions (PSRAD) creating the mask by using the fact that this instruction shifts in the sign bit. In order to overcome this instruction set limitation (no 128-bit single-bit shift instruction) we tried if field multiplication by x^8 is faster. Now the input needs to be shifted eight bits, which can be performed using a single byte-shuffle instruction. The reduction, a subtraction by $i \cdot R$, where $0 \leq i < 2^8$, depends on the eight bits shifted out. Since the reduction polynomial is constant we can precompute the 256 multiples and use the shifted-out byte as in index for this look-up table. We found that, using our implementation of both approaches, the performance of both field multiplications, by x and x^8, are comparable with a slight advantage when multiplying by x.

3 Implementations of the Target Algorithms

Table 3 contains an overview of the benchmarks we obtained. The measurements have been carried out analogously to [19]; i.e. with the help of the time stamp counter which is read using the RDTSC instruction. The presented performance results are an average over thousands of times compressing a random 4KB message. In the sequel, we provide separate treatments for constructions based on a (variable-key) blockcipher versus a permutation (in which case we fix the key of the blockcipher). Due to space limitations, we refer to the original works for exact specifications of the various algorithms (references, including those relevant for security results, are given in Table 3; see also the full version [12] for a more detailed analysis and illustrated specifications).

Two of the designs considered are based on past SHA-3 candidates. For those, we instantiate the underlying permutation by (fixed-key) RIJNDAEL-256, rather than the originally submitted permutation. For compression functions supporting more than

Table 3. The achieved speeds (in cycles per byte) using the AES-NI for the designs considered in this work. Also mentioned are the number of b bytes which are absorbed per compression function call and how many unique key scheduling calls are made (see Table 1 for the primitives employed). Predicted speed estimates are based on the results from Table 2. The last column provides additional references.

Algorithm	b	Key Scheduling	Predicted Speed Range	Achieved Speed	Security Reference
ABREAST-DM [28]	16	two	$11.1 + \epsilon$	11.21	[16,29,33]
DM [39]	32	one	$[6.8, 10.2]$	8.69	[48,10]
HIROSE-DBL [21]	16	one, shared	9.6	9.82	[21,27]
KNUDSEN–PRENEEL [26]	32	four	10.6	10.58	[44,46]
LANE* (Sec. 3)	64	fixed	11.7	11.71	[22]
LP231 [51,52]	32	fixed	$12.6 + \epsilon$	13.04	[51,52,30]
LP362 [51,52]	16	fixed	$11.8 + \epsilon$	12.09	[51,52,31]
LUFFA* (Sec. 3)	32	fixed	$8.8 + \epsilon$	10.22	[15]
MDC-2 [13]	16	two	$[9.3, 11.7] + \epsilon$	10.00	[57,25]
MJH [32]	16	one, shared	$6.6 + \epsilon$	7.45	[32]
MJH-DOUBLE [32]	32	one, shared	$4.1 + \epsilon$	4.82	[32]
QPB-DBL [55]	16	one	$9.5 + \epsilon$	14.12	[55]
PEYRIN ET AL.(I) [47]	16	three, shared	$[12.5, 16.3]$	15.09	[53]
PEYRIN ET AL.(II) [47]	32	three, shared	$[7.8, 10.7]$	8.75	[53]
SHRIMPTON–STAM [54]	32	fixed	12.6	12.39	[54]

256-bit output (e.g. KNUDSEN–PRENEEL and LUFFA*) an output transformation (after MD-iteration) can be used to reduce the final output to 256-bit, however we neither implemented nor timed this.

3.1 Blockcipher-Based Constructions

Davies–Meyer (DM). Davies–Meyer (DM) [39] is a single-block-length compression function design. It is one of the most popular ways of creating a secure hash function using a blockcipher: many cryptographic hash functions, including MD5 [50] (for $n = 128$, $k = 512$) and SHA-256 [42] (for $n = 256$, $k = 512$), follow the DM design philosophy. DM is one of the most efficient PGV-type compression functions as it allows to run several key schedules independently in the MD-iteration. In our implementations, we exploit this feature; yet we also study other possible optimizations. Namely, these are the three flavors of DM that we have considered in our benchmark:

1. Standard iterative approach: compression function calls are made sequentially for each step in the MD-iteration. The compression function evaluation starts with the key schedule and continues with the encryption call. Independent key schedule and encryption rounds are interleaved to get more efficient results.
2. Partially pipelined: the encryption call of the current round and the key schedule of the next round are being processed concurrently.
3. Fully pipelined: j key schedules are called in parallel for some (integer) $j > 1$ followed by j iterative encryption calls . Several experiments were run for varying j

and the best result is obtained for $j = 4$. Note that this approach allows to interleave the first encryption round calls with the key scheduling stage to hide latencies and obtain faster results.

Among the three approaches the fully pipelined version gives the best result and is the one reported in Table 3. We included a prediction of the performance of DM based on the vanilla timings of RIJNDAEL-256 provided in Section 2. Here the timing for **4K4E** serves as a lower bound, as it makes the encryption calls in parallel. The timing for **4K** plus four times **1E** serves as an upper bound for DM because the first encryption can be scheduled during the four key scheduling stages hiding the instruction dependencies in the encryption improving the overall throughput. (Similar strategies are used for the constructions discussed subsequently. If the predictions in Table 3 include an ϵ, this indicates that certain computations, for instance finite field multiplications, are not considered in the prediction.)

ABREAST-DM. ABREAST-DM and its sister design TANDEM-DM, both proposed in the early 90s [28], are two of the classical examples of double-block-length compression/hash function designs. We only consider ABREAST-DM instantiated with AES-256 for our benchmark. We expect that TANDEM-DM has a slightly worse performance compared to ABREAST-DM due to its sequential structure. In our implementations, we make extensive use of the parallelism inside the ABREAST-DM compression function by calling two key schedules in parallel followed by two concurrent encryption calls (where the 'follow' is on a fine-grained per AES-round basis). Hence, the prediction for ABREAST-DM is based on the performance numbers for AES-256 in the **2K2E** setting (see also [12] for the discussion on another alternative yet slower method to implement ABREAST-DM).

HIROSE-DBL. ABREAST-DM suffers from a performance drawback that, although run in parallel, the underlying blockciphers require separate key schedule routines. Hirose's construction [21] overcomes this problem by sharing the key scheduling for the two blockcipher calls. In our implementations, we apply the same approach as for ABREAST-DM to HIROSE-DBL and our results are in accordance with the predicted speed based on the **1K2E** setting for AES-256. Our timings also demonstrate that Hirose's scheme is indeed faster than ABREAST-DM.

MDC-2. MDC-2 [13] is one of the oldest double-block-length hash functions available and it has been specified in the ANSI X9.31 and ISO/IEC 10118-2 standards [1,23]. Although originally designed for use with DES, we consider the obvious generalization where one can use two calls to a single-key blockcipher (where $k = n$ with AES-128). Since MDC-2 is based on MMO, it is difficult to pipeline multiple MDC-2 compression function calls in the MD-iteration (as we did for DM). Yet, one can benefit from the parallelism naturally present within a single compression function evaluation by making the two blockcipher calls concurrently (corresponding to **2K2E**). This is indeed how we have achieved our best result, matching the predicted speed.

MJH. Recently, an alternative construction called MJH was proposed by Lee and Stam [32]. It is inspired by the compression function of JH [61] (one of the SHA-3 finalists). The main design rationale behind MJH is to reduce the number of key-schedules required in a single compression function evaluation—as in HIROSE-DBL—and call several key schedules in parallel in multiple iterations—as in DM. Obviously, this results in an efficient design. More interestingly, the security of the construction still holds once the message block (size) to the compression function is doubled (this is what we call MJH-DOUBLE). This leads to a significantly more efficient scheme, although the cost of key set-up increases. We investigate the performance of MJH in accordance with our optimizations on DM and HIROSE-DBL. Based on our results, we note that MJH-DOUBLE has achieved the best cycle count in our benchmark. We implemented different strategies when interleaving $1 \leq i \leq 8$ iterations of the compression function, the best results are obtained with $i = 2$. Hence, the predictions are based on the setting **2K2E+2E**, ignoring the cost of the finite field (scalar) multiplications.

KNUDSEN–PRENEEL. One of the classical examples of multi-block-length compression functions is provided by Knudsen and Preneel [26] who proposed several constructions with multiple blockcipher calls in parallel using the generator matrices of various linear error correcting codes. We consider one of their proposals, which is based on a $[4, 2, 3]$ linear code over $\mathbb{F}_{2^3}$, to show its performance capabilities with AES-NI. When based on AES-256, this gives rise to a $6n \rightarrow 4n$ bit compression function with security expected to be at least that of a $2n$-bit compression function. We base our exact specification on the later analysis by Özen et al. [44]. One of the nice features of this construction is that one can call four independent key schedules followed by four independent encryptions where one can interleave the rounds of both operations to hide latencies. This makes it much easier to give an accurate performance estimate since this scenario is exactly the **4K4E** case for AES-256.

PEYRIN ET AL.-DBL. All the designs considered so far follow a very similar approach: there exist linear pre- and post-processing functions that operate on the blocks of data, interacting with the underlying primitives. Based on this general model, Peyrin et al. [47] determined, under a very general attack-based approach (i.e. only considering time-complexity upper bounds), necessary conditions to have a secure compression function (where they used smaller ideal $2n \rightarrow n$ and $3n \rightarrow n$ bits compression functions as underlying primitives which are replaced by single-key, resp. double-key, blockciphers in DM mode in our framework). To investigate the performance using AES-NI, we consider their two concrete proposals: one uses five AES-128 calls and leads to a $3n \rightarrow 2n$ bit compression function, the other uses five AES-256 calls for a $4n \rightarrow 2n$ bit compression function. In our implementations, we make use of the high parallelism inside a single compression function evaluation by calling several shared key-schedules. In both scenarios the predicted time corresponds to **3K5E**, since among the five encryptions two keys are used twice. This case is not considered in Table 2 and we estimate the performance by considering the performance interval [**3K3E**, **3K3E+2E**] for AES-128 and AES-256 instead.

QPB-DBL. We finish this section with the interesting scenario of constructing a $2n$-bit digest while making only a single call to the blockcipher (theoretically, this would provide optimal efficiency). Lucks [36] provided the first construction of this type, although it is secure only in the iteration (see [45] for a detailed discussion of the security of Lucks' construction). The main practical overhead in Lucks' construction are the costly finite field multiplications that are bound to be performed sequentially. Later, Stam [55] gave another, more practical, construction in the public random function model using a quadratic-polynomial based design (hence the name QPB-DBL). This construction was generalized [56,34] to the ideal cipher model by replacing the random function with a double-length-key blockcipher running in DM mode. For our benchmarks, we use a slightly modified compression function, in that we shuffle the inputs slightly. This allows us to benefit from increased parallelism in the iteration, without violating the security proof. As already argued, in the QPB-DBL compression function the main overhead consists of costly finite field multiplications (which we try to minimize by using the features of the new PCLMULQDQ instruction). Our tweak allows us to interleave the key-scheduling of round $i + 1$ with the two (sequential) finite field multiplications of round i. The predicted performance of QPB-DBL is based on the **1K1E** setting for AES-256 and ignores the relatively high cost of the two (full) finite field multiplications.

3.2 Permutation-Based Constructions

Rogaway and Steinberger's LP and SHRIMPTON–STAM. Rogaway and Steinberger introduced a class of linearly-determined, permutation-based compression functions $\{0,1\}^{mn} \rightarrow \{0,1\}^{rn}$ making k calls to the different permutations π_i for $i \in \{1,\ldots,k\}$ (hence the notation LPmkr throughout). Let (x_i, y_i) denote the input-output pair corresponding to the permutation π_i. The main ingredient of Rogaway and Steinberger's LP design is a $(k + r) \times (k + m)$ matrix A over $\mathbb{F}_{2^n}$. This matrix determines the block-wise interaction between the inputs to the compression function (V, M), (x_i, y_i) pairs and the output Z of the compression function in the following way: for the row vector a_i (of A), the inputs to the underlying permutations are determined by the scalar product $x_i = a_i \cdot (V_1, \ldots, V_r, M_1, \ldots, M_{m-r}, y_1, \ldots, y_{i-1}, 0^{k-i})$ whereas the output Z (which is treated as a concatenation of r n-bit blocks Z_i) is computed by $Z_i = a_{k+i} \cdot (V_1, \ldots, V_r, M_1, \ldots, M_{m-r}, y_1, \ldots, y_k)$. There is considerable choice for the matrices A, as long as a certain independence criterion is satisfied [51,52]). For optimal performance, we use the matrices A' over $\mathbb{F}_{2^{256}}$ (suggested in [30]), and A'' over $\mathbb{F}_{2^{128}}$ (taken from [31]) as given in Fig. 1 for LP231, respectively LP362.

In independent work, Shrimpton and Stam [54] proved security for a compression function (SS) that can be regarded as an LP231 scheme (based on matrix $\tilde{A}$), even though their matrix does not satisfy the independence criterion imposed by Rogaway and Steinberger.

There are multiple ways how one can implement these constructions in practice. We chose to implement LP231 in three stages where we first run two permutations and a field multiplication by x in parallel, followed by one permutation and field multiplication by x and x^2 and finally the remaining multiplication by x^2. This corresponds to the setting **2E+1E** $+ \epsilon$ for RIJNDAEL-256 on which we base our performance prediction. After experimenting with different strategies we settled on the following regarding LP362. Again three stages are used where we do three, two and one permutation in parallel in every stage. The multiplications are calculated in the last two stages in order to hide the relatively high latencies of especially the single permutation. Hence, the predicted performance is based on the **3E+2E+1E**$+\epsilon$ setting for AES-128. The implementation of SS is straightforward corresponding to the setting **2E+1E** for RIJNDAEL-256,

$$A' = \begin{pmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 1&2&1&1&0 \\ 1&1&2&4&2 \end{pmatrix}, \quad \tilde{A} = \begin{pmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 1&1&1&1&0 \\ 1&0&1&0&1 \end{pmatrix} \quad \text{and} \quad A'' = \begin{pmatrix} 1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0 \\ 0&0&1&0&0&1&1&0&0 \\ 1&1&1&1&1&1&1&0&1 \\ 1&2&4&1&2&4&0&1&0 \end{pmatrix}$$

Fig. 1. The matrices used for LP231, LP362 and SS. The field elements denoted $1, 2$ and 4 correspond to polynomials $1, x$ and x^2, respectively (see Section 2).

note that this is the only case where the actual construction (slightly) outperforms the predicted speed. This anomaly might be explained by the fact that SS has to load (store) the input (output) only once for both operations while in the performance benchmark setting this has to be done twice.

LANE*. LANE [22] is a permutation-based hash function design submitted to the SHA-3 competition by Indesteege (supported by the COSIC research group). For our purposes, we consider the LANE compression function with 256-bit digest size which is instantiated by eight calls to the fixed-key RIJNDAEL-256 and denoted by LANE*. Although some weaknesses have been exploited [38] for the original proposal, it is not immediate that the attacks carry over to LANE* as the attacks exploit weaknesses in the original permutations (in particular the relatively low number of rounds). In our implementations, we exploit the high parallelism inside a single compression function evaluation by running several permutation calls in parallel. Although possible, we did not investigate further pipelining options along the MD-iteration due to sufficient number of independent permutation calls in a single compression function evaluation. The predicted speed for LANE* is based on the setting of **6E+2E** for RIJNDAEL-256. Note that the original version of LANE, performs significantly faster (4.3 cycles per byte) on our platform due to the relatively light permutations given in the submitted version.

LUFFA*. LUFFA [15] is a second round permutation-based SHA-3 candidate designed by De Cannière and Watanabe which can possibly benefit from the AES-NI once the underlying permutations are modified accordingly. To this end, we instantiate the three underlying permutations of LUFFA-256 with fixed-key RIJNDAEL-256 and denote this version by LUFFA*. In the implementation of LUFFA*, we follow a standard approach: first the multiplications required in the message injection step are computed (see [15] for a description of how to implement these efficiently), followed by the computation of the three independent permutations. The predicted performance results (**3E** $+\,\epsilon$ using RIJNDAEL-256) is too optimistic, the ϵ incorporates the cost of the multiple polynomial multiplications. Note that our implementation is slightly faster then the original version of LUFFA (which runs at 10.49 cycles per byte) using the fastest implementation (called `SSSE3-PS-2`) submitted to eBASH [4].

4 Discussion and Conclusion

In this work, we presented the first comprehensive performance comparison of many multi-block-length hash functions (old and new alike) in software on a modern architecture supporting AES-NI. Our results are summarized in Table 3 in conjunction with speed predictions based on the vanilla AES timings from Table 2. Based on these results, we can draw the following conclusions:

1. Our major conclusion is that, when assuming that the underlying primitives behave ideally, one can obtain fast and provably secure blockcipher-based hash functions on soon to be mainstream architectures supporting AES-NI. Indeed, the algorithms studied provide reasonable collision and preimage resistance and require between 4 and 15 cycles per byte on our target platform, so in this sense almost *all* of them outperform SHA-256 while several of them are faster than SHA-512.[1] As discussed in the introductions, our results are obtained with the original number of rounds for AES and RIJNDAEL-256. Relative performance results follow by increasing or decreasing the number of rounds, depending on the security margin.
2. Among the blockcipher-based compression functions, DM is the fastest algorithm when optimal security (in terms of proven collision resistance lower bound) is desired. For practical security levels, MJH-DOUBLE significantly outperforms the others (including the permutation-based designs). Note that both constructions require only one key schedule call inside a single compression function evaluation.
3. In the permutation-based setting, the LUFFA* compression function is the fastest, but it is being outperformed by many blockcipher-based constructions. This is partly due to the higher number of primitive calls, but one can argue that our methodology (use AES as is) results in a relatively more conservative security margin for fixed-key constructions. Among the provably secure constructions LP362 performs the best, showing the possibility of achieving higher speed despite the increased number of primitive calls.

Finally, we remark that all the constructions we consider are generic in the sense that they can be instantiated with any secure blockcipher (or permutation, where relevant). Hence, it is well possible that one can achieve better performance with different blockciphers or permutations. In particular, any AES-inspired yet more efficient primitive, for instance a round-reduced version or a tweaked version with more secure and efficient key-scheduling, would result in a faster scheme on our target platform. We believe that our benchmark provides a valuable toolbox to see the relative performance figures for a majority of blockcipher- and permutation-based compression and hash functions.

Acknowledgements. This work was supported by the Swiss National Science Foundation under grant numbers 200020-132160, 200021-119776, and 200021-122162 and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. We gratefully acknowledge Çağdaş Çalık and Institute of Applied Mathematics at Middle East Technical University for granting us access to the Intel i5

[1] Compared SHA-256 and SHA-512 speeds (13.90 and 10.47 respectively) are based on the fastest publicly available implementation on eBACS [4] run on Intel Core i5 M 520 (2.4 GHz with AES-NI).

with AES-NI to benchmark our programs and Thorsten Kleinjung for useful discussions on how to optimize the SSE field multiplication by x. We would like to thank the anonymous reviewers for their useful comments and suggestions.

References

1. American National Standards Institute: Public key cryptography using reversible algorithms for the financial services industry. American National Standards Institute (1998)
2. Benadjila, R., Billet, O., Gueron, S., Robshaw, M.J.B.: The Intel AES instructions set and the SHA-3 candidates. In: Matsui, M. (ed.) Asiacrypt 2009. LNCS, vol. 5912, pp. 162–178. Springer, Heidelberg (2009)
3. Bernstein, D.J.: Cache-timing attacks on AES (2005), `http://cr.yp.to/papers.html#cachetiming`
4. Bernstein, D.J., Lange, (editors), T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to` (2010)
5. Bertoni, G., Breveglieri, L., Farina, R., Regazzoni, F.: Speeding up AES by extending a 32 bit processor instruction set. In: Application-specific Systems, Architectures and Processors. pp. 275–282. IEEE Computer Society (2006)
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N. (ed.) Eurocrypt 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
7. Biham, E., Dunkelman, O.: A framework for iterative hash functions – HAIFA. Presented at *Second NIST Cryptographic Hash Workshop, 2006, Santa Barbara, USA.*
8. Biryukov, A., Khovratovich, D.: Related-key cryptanalysis of the full AES-192 and AES-256. In: Matsui, M. (ed.) Asiacrypt 2009. LNCS, vol. 5912, pp. 1–18. Springer, Heidelberg (2009)
9. Biryukov, A., Khovratovich, D., Nikolic, I.: Distinguisher and related-key attack on the full AES-256. In: Halevi, S. (ed.) Crypto 2009. LNCS, vol. 5677, pp. 231–249. Springer, Heidelberg (2009)
10. Black, J., Rogaway, P., Shrimpton, T., Stam, M.: An analysis of the blockcipher-based hash functions from PGV. Journal of Cryptology 23(4), 519–545 (2010)
11. Bogdanov, A., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y.: Hash functions and RFID tags: Mind the gap. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 283–299. Springer, Heidelberg (2008)
12. Bos, J.W., Özen, O., Stam, M.: Efficient hashing using the AES instruction set. Cryptology ePrint Archive, Report 2010/576 (2010)
13. Brachtl, B., Coppersmith, D., Hyden, M., Matyas, S., Jr., Meyer, C., Oseas, J., Pilpel, S., Schilling, M.: Data authentication using modification detection codes based on a public one-way encryption function. U.S. Patent No 4,908,861 (1990)
14. Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) Crypto 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
15. De Cannière, C., Sato, H., Watanabe, D.: Hash function Luffa: Supporting document. Submission to NIST (Round 2) (2009), `http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_SupportingDocument_20090915.pdf`
16. Fleischmann, E., Gorski, M., Lucks, S.: Security of cyclic double block length hash functions. In: Parker, M. (ed.) Cryptography and Coding 2009. LNCS, vol. 5921, pp. 153–175. Springer, Heidelberg (2009)
17. Fog, A.: Instruction tables, lists of instruction latencies, throughputs and microoperation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/` (2010)

18. Gueron, S.: Intel's new AES instructions for enhanced performance and security. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 51–66. Springer, Heidelberg (2009)
19. Gueron, S.: Intel advanced encryption standard (AES) instructions set. Tech. rep., Intel (2010), `http://software.intel.com/file/24917`
20. Gueron, S., Kounavis, M.E.: Intel carry-less multiplication instruction and its usage for computing the GCM mode. Tech. rep., Intel (2010), `http://software.intel.com/file/24918`
21. Hirose, S.: Some plausible constructions of double-block-length hash functions. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, pp. 210–225. Springer, Heidelberg (2006)
22. Indesteege, S.: The LANE hash function. Submission to NIST (2008), `http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf`
23. International Organization for Standardization: ISO/IEC 10118-2: hash functions using an n-bit block cipher (2010)
24. Khovratovich, D.: New Approaches to the Cryptanalysis of Symmetric Primitives. Ph.D. thesis, University of Luxembourg (2010)
25. Knudsen, L.R., Mendel, F., Rechberger, C., Thomsen, S.S.: Cryptanalysis of MDC-2. In: Joux, A. (ed.) Eurocrypt 2009. LNCS, vol. 5479, pp. 106–120. Springer, Heidelberg (2009)
26. Knudsen, L.R., Preneel, B.: Construction of secure and fast hash functions using nonbinary error-correcting codes. IEEE Transactions on Information Theory 48(9), 2524–2539 (2002)
27. Krause, M., Armknecht, F., Fleischmann, E.: Preimage resistance beyond the birthday barrier – the case of blockcipher based hashing. Cryptology ePrint Archive, Report 2010/519 (2010)
28. Lai, X., Massey, J.L.: Hash functions based on block ciphers. In: Rueppel, R. (ed.) Eurocrypt 1992. LNCS, vol. 658, pp. 55–70. Springer, Heidelberg (1993)
29. Lee, J., Kwon, D.: The security of Abreast-DM in the ideal cipher model. Cryptology ePrint Archive, Report 2009/225 (2009)
30. Lee, J., Park, J.H.: Adaptive preimage resistance and permutation-based hash functions. Cryptology ePrint Archive, Report 2009/066 (2009)
31. Lee, J., Park, J.H.: Preimage resistance of LPmkr with $r = m - 1$. Information Processing Letters 110(14-15), 602–608 (2010)
32. Lee, J., Stam, M.: MJH: a faster alternative to MDC-2. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 213–236. Springer, Heidelberg (2011)
33. Lee, J., Stam, M., Steinberger, J.: The collision security of Tandem-DM in the ideal cipher model (2011)
34. Lee, J., Steinberger, J.P.: Multi-property-preserving domain extension using polynomial-based modes of operation. In: Gilbert, H. (ed.) Eurocrypt 2010. LNCS, vol. 6110, pp. 573–596. Springer, Heidelberg (2010)
35. Lucks, S.: A failure-friendly design principle for hash functions. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474–494. Springer, Heidelberg (2005)
36. Lucks, S.: A collision-resistant rate-1 double-block-length hash function. In: Symmetric Cryptography. No. 07021 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
37. Manley, R., Magrath, P., Gregg, D.: Code generation for hardware accelerated AES. In: Application-specific Systems Architectures and Processors (ASAP), 21st IEEE International Conference on. pp. 345–348 (2010)
38. Matusiewicz, K., Naya-Plasencia, M., Nikolic, I., Sasaki, Y., Schläffer, M.: Rebound attack on the full Lane compression function. In: Matsui, M. (ed.) Asiacrypt 2009. LNCS, vol. 5912, pp. 106–125. Springer, Heidelberg (2009)
39. Menezes, A.J., van Oorschot, P.C., Vanstone, S.: CRC-Handbook of Applied Cryptography. CRC Press (1996)
40. Merkle, R.C.: One way hash functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)

41. NIST: FIPS-197: Advanced encryption standard (AES) (2001), http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf
42. NIST: Secure hash standard. FIPS 180-2, NIST, http://www.itl.nist.gov/fipspubs/fip180-2.htm (August 2002)
43. NIST: Cryptographic hash algorithm competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html (2008)
44. Özen, O., Shrimpton, T., Stam, M.: Attacking the Knudsen-Preneel compression functions. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 94–115. Springer, Heidelberg (2010)
45. Özen, O., Stam, M.: Another glance at double-length hashing. In: Parker, M. (ed.) Cryptography and Coding 2009. LNCS, vol. 5921, pp. 176–201. Springer, Heidelberg (2009)
46. Özen, O., Stam, M.: Collision attacks against the Knudsen-Preneel compression functions. In: Abe, M. (ed.) Asiacrypt 2010. LNCS, vol. 6477, pp. 76–93. Springer, Heidelberg (2010)
47. Peyrin, T., Gilbert, H., Muller, F., Robshaw, M.J.B.: Combining compression functions and block cipher-based hash functions. In: Lai, X., Chen, K. (eds.) Asiacrypt 2006. LNCS, vol. 4284, pp. 315–331. Springer, Heidelberg (2006)
48. Preneel, B., Govaerts, R., Vandewalle, J.: Hash functions based on block ciphers: A synthetic approach. In: Stinson, D. (ed.) Crypto 1993. LNCS, vol. 773, pp. 368–378. Springer, Heidelberg (1994)
49. Rabin, M.O.: Digitalized signatures. In: Foundations of Secure Computations. pp. 155–166. Academic Press (1978)
50. Rivest, R.: The MD5 message-digest algorithm, request for comments (RFC) 1320. Tech. rep., Internet Activities Board, Internet Privacy Task Force (1992)
51. Rogaway, P., Steinberger, J.: Security/efficiency tradeoffs for permutation-based hashing. In: Smart, N. (ed.) Eurocrypt 2008. LNCS, vol. 4965, pp. 220–236. Springer, Heidelberg (2008)
52. Rogaway, P., Steinberger, J.P.: Constructing cryptographic hash functions from fixed-key blockciphers. In: Wagner, D. (ed.) Crypto 2008. LNCS, vol. 5157, pp. 433–450. Springer, Heidelberg (2008)
53. Seurin, Y., Peyrin, T.: Security analysis of constructions combining FIL random oracles. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 119–136. Springer, Heidelberg (2007)
54. Shrimpton, T., Stam, M.: Building a collision-resistant compression function from non-compressing primitives. In: Aceto, L., Damgård, I., Goldberg, L., Halldórsson, M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) International Colloquium on Automata, Languages and Programming 2008. LNCS, vol. 5126, pp. 643–654. Springer, Heidelberg (2008)
55. Stam, M.: Beyond uniformity: Better security/efficiency tradeoffs for compression functions. In: Wagner, D. (ed.) Crypto 2008. LNCS, vol. 5157, pp. 397–412. Springer, Heidelberg (2008)
56. Stam, M.: Blockcipher-based hashing revisited. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 67–83. Springer, Heidelberg (2009)
57. Steinberger, J.P.: The collision intractability of MDC-2 in the ideal-cipher model. In: Naor, M. (ed.) Eurocrypt 2007. LNCS, vol. 4515, pp. 34–51. Springer, Heidelberg (2007)
58. Tillich, S., Großschädl, J.: Instruction set extensions for efficient AES implementation on 32-bit processors. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 270–284. Springer, Heidelberg (2006)
59. Tillich, S., Herbst, C.: Boosting AES performance on a tiny processor core. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 170–186 (2008)
60. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. Journal of Cryptology 23, 37–71 (2010)
61. Wu, H.: The hash function JH. Submission to NIST (updated) (2009), http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh_round2.pdf