Multi-Block Length Hashing using the AES Instruction Set

Joppe W. Bos and Onur Özen

Laboratory for Cryptologic Algorithms, EPFL,
Station 14, CH-1015 Lausanne, Switzerland
{joppe.bos, onur.ozen}@epfl.ch

Abstract. In this work, the most well-known cryptographic hash function designs with provable security reductions to an underlying primitive are approached from a practical point of view. Starting from the early constructions, we consider several blockcipher and permutation based compression and hash function designs and compare their performance on one of the latest architectures supporting the AES instruction set. As far as we are aware, this is the first work to compare and implement various multi-block length hash function constructions in software. We instantiate the primitives of these constructions with AES-128, AES-256 and its larger state variant RIJNDAEL-256, exploiting the performance gain provided by the new AES instruction set. As a result, we obtain a wide range benchmark of the most well known compression function designs on a soon to be mainstream architecture. We conclude that most algorithms we consider outperform the current cryptographic hash standard SHA-256 plus two out of the four AES-inspired second-round SHA-3 candidates.

1 Introduction

This paper is about understanding the performance of the currently available blockcipher and permutation based compression functions, with provable security properties, on architectures supporting the AES instruction set. Our goal is to illustrate what really makes the compression function designs efficient, for some targeted level of security, and to draw conclusions for obtaining better schemes.

For years, most of the well known cryptographic hash function designs have been revolving around a similar design principle [63,52,24]: the Merkle-Damgård paradigm, although modified later under different names to thwart some attacks and increase the performance [48,11,8]. In this cascaded mode of operation, the main focus is to construct a secure[1] and an efficient compression function to deliver these properties to the overall hash function. The compression functions, on the other hand, are built using a primitive which is, or can be regarded as, either a blockcipher or a permutation (e.g. a fixed-key blockcipher). A well-studied class of compression functions are the *single-block length* ones that are based on a blockcipher operating on n-bit blocks with k-bit keys[2] and hence, produce a compression function from $n + k$ bits to n bits. The so-called Preneel, Govaerts and Vandewalle [62] compression functions are the most well-known examples in this category (for $k = n$).

Single-block length hash and compression functions face one major drawback: in order to meet today's one basic security requirement (namely collision resistance), one needs a primitive operating on more than 160 bits [25], thus, ruling out most existing blockciphers, including the current US standard AES [53] operating on 128-bit blocks only. In order to remedy this issue, *double-block length* compression functions and more generally *multi-block length* compression functions were introduced. The latter have tn-bit output (for $t \geq 2$) while being based on a primitive with only n-bit blocks. Thus, one can hope that, for instance, finding collisions requires more than 2^n time (primitive evaluations) using only smaller primitives.

This output expansion is achieved by calling the primitive multiple times and then combining the resulting primitive outputs. In the literature, there has been significant efforts to design and analyze the multi-block length compression functions. Nevertheless, in general, most of the papers in this field are aimed at evaluating the security of the these constructions. The efficiency, on the other hand, is only taken

[1] This assumption can be violated as it is possible to build a secure hash function from a weak compression function [17].

[2] In the sequel, we call the permutation based compression functions single-block length as well that output the same digest-size as the block-length of the permutation. See Section 3 for more on this.

into account retrospectively; actual performance benchmarks, on hard- or software devices, is normally left as a future work[3].

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 or RIJNDAEL-256, and their fixed key versions to build permutations, we obtain hash functions with a fixed 256 digest size with provable security properties[4] (either for collision resistance or for preimage resistance) assuming these underlying primitives are secure. Our choice for the AES and RIJNDAEL-256 [22,53] is two-fold. For security reasons we rely on the AES, this well-studied, and currently world wide deployed blockcipher, has survived many years of cryptanalysis and a practical break of this cipher would have a significant impact on the cryptographic landscape[5]. Secondly, our choice to use the AES as our primitive is from a performance perspective. Although already fast in hard- and software, the AES can be made even more efficient by using the recent AES instruction set (AES-NI) extensions [30,31].

To the best of our knowledge, this is the first work to give an overview of software implementations of the most studied and influential blockcipher and permutation based compression functions. We summarize our results in Table 2 together with the known security results gathered from the literature. This table shows that, when assuming the underlying primitives behave ideally, fast and provably secure hash functions are already available. Moreover, we compare the obtained results with the performance of the four AES-inspired second-round SHA-3 candidates which benefit from the AES-NI as well. Surprisingly, most of the designs outperform two of these candidates; see Section 5 for a discussion of our results.

The rest of this paper is organized as follows. In Section 2, we overview the basic tools required in the paper. Section 3 introduces our target algorithms together with the related literature in the field. In Section 4, we present our benchmark results together with implementation details. We discuss the outcomes of our results and conclude the paper in Section 5. The pictorial illustrations of the target algorithms are given in Appendix A.

2 Preliminaries

2.1 The AES and the AES Instruction Set

The AES is a fixed block-length version of the RIJNDAEL blockcipher [22,53] that was standardized by the US National Institute of Standards and Technology (NIST) after a public competition similar to the one currently ongoing for the SHA-3 [55]. The AES operates on an internal state of 128 bits while supporting 128-, 192-, and 256-bit keys (each version with different key scheduling units). The state of AES 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 `xor`ed into the state, after which the state is modified $N_r - 1$ times according to the round function, followed by the slightly different final round. The round function consists of four steps: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` (except for the final round which omits the `MixColumns` step) each of which operating on the 128-bit state (for the exact details see the AES specification in [22,53]). We note that the larger state variant of AES, RIJNDAEL-256, operates almost in the same way except for the `ShiftRows` operation, a state size of 256 bits, a 256-bit key and $N_r = 14$ rounds.

[3] The only exception is the work of Bogdanov et al. [18] where a few multi-block length compression functions are benchmarked in hardware.

[4] Note that two of the SHA-3 candidates we consider, i.e. LANE and LUFFA, do not have security proofs, neither for collision resistance nor for preimage resistance; yet we include those in our benchmark to illustrate their performance capabilities.

[5] Note, however, that the latest attacks on AES-256 [13,14] show that the AES-256 does not behave ideally; yet these attacks can not be exploited, for now, to break the conventional security properties like collision or preimage resistance for the hash function.

Nine years after becoming the symmetric encryption standard the only theoretical attack on the full AES is applicable in the related key scenario to the 192-bit [13] and 256-bit key versions [13,14]. So far no theoretical attacks on all rounds of the AES-128 are known. More cryptanalysis success has been achieved by using the characteristics from the actual implementation of the AES. An instance of such software side channel attacks are cache attacks [57,4], the techniques from [57] allow one to obtain the AES key in only 65 milliseconds (see [76] for a survey covering extensions and variants of such types of attacks on the AES).

Designing fast implementations, which overcome the various software side channel attacks, has been an active research area in the recent years. Recent examples are bitsliced [9] implementations for Intel core i7 architectures [38] and implementations [34,6] which target a variety of common CPU architectures. In the latter an overview of the state-of-the-art fast AES implementations is given. The fastest AES implementations targeting microcontrollers are described in [56].

In the last decade a general trend in computer architecture design is to enhance the speed of software implementations by offloading the computational work to special units which operate on larger data types, improving overall throughput, by using the *single instruction, multiple data* (SIMD) paradigm. 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 [30,31] 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 different people have suggested extensions to current instruction sets to improve the performance of the AES in practice [74,73,7,75]. The AES-NI does not only increase the performance of the 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. The following description of these instructions are from [30,31,32]:

- `AESENC` performs a single round of encryption. The instruction combines the four steps of the AES algorithm `ShiftRows`, `SubBytes`, `MixColumns` and `AddRoundKey` into a single instruction.
- `AESENCLAST` performs the last round of encryption. Combines the `ShiftRows`, `SubBytes` and `AddRoundKey` steps into one instruction.
- `AESDEC` performs a single round of decryption.
- `AESDECLAST` performs the last round of 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.

Performance results of implementations in the C-programming language or assembly can be found in [31,32]. A study of arranging the AES instructions in an optimal order to decrease instruction dependencies is performed in [49].

2.2 Security Considerations

There exist multiple security notions for a cryptographic hash function to satisfy; we only consider the collision- and preimage-resistance and the relevant adversarial models. A *preimage-finding adversary* is an algorithm with access to one or more oracles and whose goal is to find a preimage of some specified compression/hash function output [6]. Similarly, a *collision-finding adversary* is an algorithm whose goal is to find collisions in some specified compression or hash function.

[6] Note that there exist several definitions of preimage resistance [65] and the known results in the literature vary depending on the definition. We state, in Section 4, the known results under the name preimage resistance without differentiating between the different definitions.

Table 1. The number of AES-like operations per b bytes for all AES-inspired candidates. `(R)` : One AES encryption round, `SB`: Substitution operation, `MC`X: Mix-Column operation over X bytes (i.e., X=4 is the one used in AES). The performance in cycles per byte, with and without using the AES instruction set, on an Intel Core i5 650 (3.20GHz), using icc 11.1, is displayed in the second part of the table together with the speedup factor obtained when using the AES instruction set. The implementations are taken from eBASH, the implementation name is given in parentheses.

Hash function	b	(R)	SB	MC4	MC8	MC16	xor (byte)	with	without	speed-up
								using the AES-NI		
ECHO-SP	224	256	-	512	-	-	448	5.37 (aes)	23.57 (core2)	4.39
ECHO-DP	192							6.22 (aes)	25.50 (core2)	4.10
FUGUE	4	-	32	-	-	2	60	15.99 (SSE4.1)	18.02 (SSSE3)	1.13
GRØSTL	64	-	1280	-	160	-	1472	13.63 (aes-ni)	20.31 (asm)	1.49
SHAVITE-3	64	52	-	-	-	-	1280	5.32 (aes-instruct)	18.54 (lower-mem)	3.48

To get a better picture, we consider adversaries in two scenarios: the information- and the complexity-theoretic. For the former, the only resource of interest is the number of queries made to their oracles, the so called query complexity, where the adversaries are considered computationally unbounded. For the latter, on the other hand, we consider the actual runtime of the adversarial algorithm (with respect to a reasonable computational model). By convention, we assume that for optimally secure constructions the only valid attacks are the generic attacks which require $\Theta(2^{s/2})$ and $\Theta(2^s)$ queries/time to find collisions and preimages for a hash/compression function of s-bit digest respectively. From an information-theoretic point of view, the compression functions we consider are not always optimal. Yet, in the complexity-theoretic setting, almost all of them are considered to be optimally secure in the sense that there exist no known algorithm to find collisions and preimages faster than the generic methods.

In the literature, the known results contain both models. For instance, most of the security bounds are given in the information-theoretic setting where the query complexity lower bound is the main source of interest. Similarly, the known attacks state an upper bound both for the query and time complexity. In the sequel, we stick to this convention and whenever we state a lower and an upper bound we mean that there exist a security bound and an attack, respectively, matching the stated bounds (see Table 2). This convention holds both for the security of the compression function and the related mode of operation. Note that for the security of the mode of operation, we assume that the Merkle-Damgård iteration is taking place for most of the compression function designs and the respective security preservations [52,24,65] hold. For the designs with different mode of operations, we simply assume that the original mode is taking place and the corresponding security reductions follow [8,11].

2.3 SHA-3 competition

Due to the recent developments in the cryptanalysis of well-known hash functions MD5 [78] and SHA-1 [77,10], a public competition is announced by the NIST [55] to develop a new cryptographic hash algorithm intended to replace the current standard SHA-2 [54]. The new hash algorithm will be called SHA-3 and subject to a Federal Information Processing Standard (FIPS) as done for the AES. The competition officially started in late 2008 with several submissions from all over the world. As a result, 64 proposals were received, of which 51 met the minimum submission requirements and became the first round candidates. In summer 2009, the number of candidates for the second round was further cut down to a more manageable size of 14 by eliminating the ones having major security or performance flaws. In this work, we consider the second round SHA-3 candidates which benefit from the AES instruction set; this has been studied in [3]. From the initial 51 candidates 10 have components which are substantially RIJNDAEL-based and 8 out of these 10 can potentially benefit from the AES instruction [3]. The current list of semi-finalists still contains 4 AES-inspired candidates: ECHO [2], FUGUE [33], GRØSTL [29] and SHAVITE-3 [12].

Table 1 shows what components of the AES these SHA-3 candidates are using. We use the notation for the MixColumn operations over X bytes as in [19], see this paper for details on the costs when implementing $\texttt{MC}X$ on platforms without the AES instruction set. The second part of Table 1 states the performance, expressed in cycles per byte, of implementation with and without the use of the AES-NI. These implementations are taken from the eBASH [5] benchmarking suite. While FUGUE and GRØSTL obtain a moderate speedup by using the AES-NI, a factor between 1.0 and 1.5, the performance of ECHO and SHAVITE-3 is increased by a factor of at least three.

3 Target Algorithms

In this section, we go over the designs that we consider in our framework and state the related literature. Throughout, we consider the compression functions rather than the full blown hash functions to evaluate the performance of the target algorithms. By doing so, we assume that for long messages the performance of the compression function is the dominating factor in evaluating the efficiency of the hash function, regardless of which mode of operation is used. By convention, we focus on the compression functions that might possibly benefit from the AES instruction set. Hence, we consider only the constructions which are either blockcipher or permutation based such that the flavors of AES or RIJNDAEL-256 (see Table 2 how the target algorithms are instantiated) can be used as underlying primitive (with or without fixed-key).

We would like to remark that the security of the constructions we consider goes through for any idealized blockcipher or permutation. Analogously, in the sequel, we separate our analysis into two, depending if the compression function is blockcipher or permutation based. Note that some of the compression function designs we consider here are based on the SHA-3 candidates. For those, we consider their compression function instantiated by one of the primitives stated above, rather than the original submitted versions. Finally, we remark that some of the compression functions support more than 256-bit output; we assume that for those compression functions either Merkle-Damgård paradigm or a similar mode of operation is taking place with an output transformation to reduce the output to 256-bit.

3.1 Blockcipher Based Constructions

Historically, the most popular way of constructing a compression function is to use a blockcipher as the underlying component (the idea of which dates back to Rabin [63]). The main motivation for using this approach is the fact that a blockcipher, operating on n bits with an k-bit key, $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ is already a compressing primitive. So, although tricky, it would be convenient to transfer the trust in the blockcipher to the corresponding compression or even hash function. Moreover, as was a common practice in the early days of modern cryptography, when the hardware platforms were the main targets, it is more practical to use a single blockcipher which implements both an encryption and a hashing primitive. For this purpose, several methods were proposed using the Data Encryption Standard as an underlying primitive [52,42,24] which is nowadays replaced by the AES. The wisdom of blockcipher based hashing is still valid; we go over the most well known techniques proposed in the literature and see the practical consequences of the AES instruction set on these designs. For the algorithms investigated here, we instantiate the underlying blockcipher with either AES-128 or RIJNDAEL-256 when a single-length key blockcipher (where $k = n$) is used (for $n = 128$ and $n = 256$ respectively). For the double-length key (for $k = 2n$) scenario we assume that AES-256 is the main component (with $n = 128$).

DM. The Davies-Meyer (DM) [51] is a single-block length compression function design which is the most popular way of using a blockcipher as an underlying main component to create a secure hash function (see Fig. 1). Most of the cryptographic hash functions, including MD5 [64] (for $n = 128$, $k = 512$) and SHA-256 [54] (for $n = 256$, $k = 512$), follow the DM design philosophy. The first extensive security analysis of DM was performed by Preneel, Govaerts and Vandewalle [62] whose

main approach was to attack a class of single-block length hash/compression functions (including DM for $n = k$) which turned out to successfully resist their efforts. The first security *proof* for DM, on the other hand, was given by Black, Rogaway and Shrimpton [16] (whose technique is later simplified by Stam [71] and jointly published in [17]) where they showed that the DM indeed enjoys optimal preimage and collision resistance. We remark that 12 similar compression functions were also proved to be optimally collision and preimage resistant; we refrain ourselves from re-implementing all of these variants and focus only on one famous *representative*. We expect them to have similar performance as DM. Note that due to the security level we are aiming, the only relevant primitive for DM among our choices is RIJNDAEL-256.

MDC-2. The MDC-2 [20] is one of the classical examples of double-block length hash functions which is also specified in the ANSI X9.31 and ISO/IEC 10118-2 standards [37,28] (see Fig. 2). Although in its first proposal it was presented to be used with DES; hence with different parameters, we consider here a slightly different version where one can use two blockciphers with n-bit block and n-bit key (e.g. with AES-128). In spite of its early appearance in the literature, the first security proof for MDC-2 was recently given by Steinberger [72] where it was shown that any collision finding adversary (to the iterated hash function[7]) asking $2^{3n/5-\epsilon}$ (for any $\epsilon > 0$) queries to the underlying blockciphers has a negligible chance of completing a collision. This lower bound then was complemented by the attacks of Knudsen et al. [39] by showing collision and preimage attacks requiring time complexities of $O(2^n/n)$ and $O(2^n)$ respectively. To the best of our knowledge, reducing the gap between the query complexity lower bound and the time complexity upper bound is still an open problem.

ABREAST-DM. After the design of MDC-2, there had been various double-block length compression/hash function designs in the early 90's that aimed to solve the problem of outputting larger digest sizes while using a primitive with a smaller block size. Unfortunately, most of the proposed constructions were shown to suffer from weak security properties. Nevertheless, there are some early designs like ABREAST-DM [42] and its sister design TANDEM-DM [42] offering almost optimal collision resistance [26,43,46] where a double-length key blockcipher is the main primitive (AES-256 in our case). In this work, we only consider ABREAST-DM (see Fig. 3) for our benchmark and we expect that TANDEM-DM has a slightly worse performance compared to ABREAST-DM due to its iterative structure.

KNUDSEN-PRENEEL. The goal for almost all constructions in the 90's has been optimal collision-resistance: a target output size is fixed and the compression function is designed to be collision resistant up to the birthday bound for that digest size. Contrary to this, Knudsen and Preneel [40] adopted a different approach by a priori fixing a particular security target and letting the output size vary in order to guarantee a security level *without* imposing optimal security. To this end, they proposed several constructions using the generator matrices of various linear error correcting codes. Although it was shown [58,60] that the compression functions do not deliver the security level[8] they were designed for, still there exist some constructions satisfying a desirable level of security (when used in wide-pipe mode along with Merkle-Damgård iteration). We consider one of their proposals that is based on a $[4, 2, 3]$ linear code over $\mathbb{F}_{2^3}$ to show its performance capabilities with AES-NI[9]. The security of Knudsen-Preneel hash functions in the Merkle-Damgård iteration is still an open problem.

HIROSE-DBL. The multi-block length compression functions MDC-2, ABREAST-DM and KNUDSEN-PRENEEL suffer from a performance drawback that, although run in parallel, the underlying blockciphers require separate key scheduling routines. In order to solve this performance issue, Hirose [35] presented

[7] The compression function of MDC-2 was already known to be weak, i.e. collisions and preimages can be found with $O(2^{n/2})$ and $O(2^n)$ queries respectively for $2n$-bit output.

[8] In [58,60], the underlying primitives are modeled as independent random functions whereas in this work we instantiate them with blockciphers (with input separation, like in HIROSE-DBL) running in DM mode.

[9] We note however that the compression function was not defined explicitly in [40]; we derive the generator matrix based on the works [40,58].

a blockcipher based double-block length compression function (see Fig. 4) that requires one shared key scheduling algorithm for two blockcipher calls. Moreover, he was also able to prove that his design enjoys almost optimal collision resistance (recently, it has been proved that the construction has preimage resistance $\Omega(2^{1.5n})$ [41]). By the time of its proposal, it was the first blockcipher based compression function design enjoying both provable security and higher performance characteristics (compared to earlier design in the same category like ABREAST-DM). We note that a similar compression function without feed-forward is shown to be almost as collision resistant as HIROSE-DBL [59]. Although we expect a higher efficiency in terms of hardware cost (i.e. area) for the construction without feed-forward, we believe it achieves almost the same speed as HIROSE-DBL in software.

PEYRIN ET AL.-DBL. All the multi-block length compression function designs considered so far follow a very similar design approach: there exist linear pre- and post-processing functions that operate on the blocks of data, interacting with the underlying primitives. The pre-processing function takes the input to the compression function, parses it as blocks and determines (block-wise linearly) the input of the underlying primitives. Similarly, the post-processing function takes the outputs of the underlying primitives, together with the input to the compression function in case there is a feed-forward, and outputs the digest (that is again based on a linear transformation). Based on this general model, Peyrin et al. [61] studied under a very general attack-based approach (like done by Preneel, Govaerts and Vandewalle for the single length compression functions [62]) that determines the necessary conditions to have a secure compression function (where they used smaller ideal compression functions as underlying primitives which are again replaced by blockciphers in DM mode in our framework). They conclude that one needs at least five calls to the blockciphers in order to thwart some generic attacks and they proposed some constructions satisfying their criteria. For our purposes, we consider two of their proposals (see Fig. 6) to investigate the performance. Note that in a later work Peyrin and Seurin [68] followed a more proof centric approach and derived some security lower and upper bounds for their proposals[10]. Improving the bounds given in [68] and showing the security in the MD-iteration are still open problems.

MJH. Recently, a faster alternative double-block length compression function to MDC-2 (see Fig. 7) was proposed by Lee and Stam [45] inspired by the compression function of JH [79], one of the second round SHA-3 candidates. As in the case of MDC-2, the compression function itself does not provide a security beyond a single-length compression function can offer; yet Lee and Stam showed that it enjoys a collision resistance bound of $2^{2n/3-\log n}$ in the MD-iteration; whereas preimages can be found with 2^n queries and in as much time. It is worth noting that the security of the construction is claimed to still hold once the message block to the compression function is doubled (this is what we call the MJH-DOUBLE). This leads to a significantly more efficient scheme, although the cost of key set-up increases.

QPB-DBL. The double-block length compression functions considered so far make use of $r > 1$ calls to an underlying blockcipher and outputs $2n$ bit digest where n is the blocksize of the underlying blockcipher. Another interesting scenario is to construct a $2n$ bit digest while making a single call to the blockcipher. To this end, Stam [70] provided the first construction of this type in the public random function model using a quadratic-polynomial based (hence the name QPB-DBL) design. What he suggested was to use a linear preprocessing function together with a nonlinear post-processing function that uses a quadratic polynomial to evaluate the digest. This construction was then generalized by Lee and Steinberger [47] to the ideal cipher model where the random function used by Stam was replaced by a double-length key blockcipher running in DM mode (see Fig. 8). In the QPB-DBL compression function, the main overhead is the costly finite field multiplications which we try to minimize by using the features of the new `CLMUL` instruction set. The security of QPB-DBL is extensively analyzed in [47].

[10] Note that as in the KNUDSEN-PRENEEL case all the security results are given under the assumption that the underlying primitives are random functions, here we instantiate these primitives with blockciphers running in DM mode, although this case was not studied throughly.

3.2 Constructions Based on Non-Compressing Primitives

The compression functions considered so far are all based on blockciphers which require both the key set-up and the data processing costs. Constructing compression functions from permutations, which can be regarded as fixed-keyed blockciphers, has some definite advantages. Firstly, blockciphers tend to have significant key scheduling costs (as pointed out in Section 4). Secondly, the recent related-key attacks [13,14] reveal that although secure in the standard model, even the AES is vulnerable against related-key attacks which are of definite interest in the hashing context. Finally, a fixed-key design leads to a more compact primitive $\pi : \{0,1\}^n \rightarrow \{0,1\}^n$ comparing to the map $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ that avoids expensive implementations. In recent years, there has been substantial efforts to design permutation based compression functions; we focus our attention on the ones that can be instantiated with one of our primitives[11].

We remark that in the blockcipher based compression functions, it is intuitively clear that the compression is indeed taking place, due to a blockcipher's already being a compressing primitive. In the permutation based setting, however, one would need a compressing primitive inside a compression function. This can be done in several ways; (i) a compressing pre- or post-processing function can be used and/or (ii) compression might take place somewhere in between the pre- or post-processing. The targets we consider in this work do benefit from either approach, sometimes even both.

ROGAWAY-STEINBERGER'S LP. In [15], Black, Cochran and Shrimpton showed an impossibility result for efficient (that is single-block length and single call to a primitive) permutation based compression functions. The investigation of the security once the non-compressing primitives are called several times is left as an open problem. As a follow-up work, Rogaway and Steinberger [66,67] studied this problem in a general setting and obtained several positive results (see Fig. 10 for the construction they consider for three calls to the permutations). They concluded that in order to have at least $2^{n/2}$ level of collision and preimage resistance, one needs at least three and five calls to the primitives for the single-block length and double-block length compression functions respectively. They also proposed several compression function designs obeying their constraints and left it as an open problem to investigate the practical consequences of LP designs. In this work, we consider two of their suggestions: LP231 and LP362 (for $n = 256$ and $n = 128$ respectively). Both of the constructions enjoy a collision resistance bound beyond $2^{n/2}$ and have varying preimage resistance (that is between $2^{n/2}$ and 2^n).

The ROGAWAY-STEINBERGER'S LP construction is a $\{0,1\}^{mn} \rightarrow \{0,1\}^{rn}$ compression function making k calls to the (either different or identical) permutations π_i for $i \in \{1,\ldots,k\}$ (hence the notation LPmkr throughout). We denote (x_i, y_i) the input-output pair corresponding to the permutation π_i. The main ingredient of ROGAWAY-STEINBERGER'S LP design is the use of a $(k+r) \times (k+m)$ matrix A (satisfying an independence criterion [66,67]) over $\mathbb{F}_{2^n}$ with entries $a_{i,j}$ for $1 \leq i \leq k+r$ and $1 \leq j \leq k+m$. This matrix is used to determine the block-wise interaction between the 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, the inputs to the underlying permutations are determined by the inner product $x_i = a_i \cdot (V_1, \ldots, V_r, M_1, \ldots, M_{m-r}, y_1, \ldots, y_{i-1})$ whereas the output Z (which is treated as a concatenation of r n-bit blocks Z_i) is computed by $Z_i = a_{k+i} \cdot (V_1, \ldots, V_r, M_1, \ldots, M_{m-r}, y_1, \ldots, y_k)$. For LP231, we use the following matrix A' to define the compression function (which is the one sug-

[11] Note that the compression functions suggested in the sponge framework are also considered to be in the permutation-based setting; yet we mainly refrain ourselves from considering those since they mainly require larger states which can not be instantiated with our primitives.

gested in [67]). For LP362 we use the matrix A'' given in [44]. These matrices are defined as

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

The entries of A' (A'') represent the elements of $\mathbb{F}_{2^{256}}$ ($\mathbb{F}_{2^{128}}$) defined by the irreducible polynomial $x^{256}+x^{10}+x^5+x^2+1$ ($x^{128}+x^7+x^2+x+1$) over $\mathbb{F}_2$. That is, $0{\times}1 \equiv 1$, $0{\times}2 \equiv x$ etc. Note that the compression function suggested by Shrimpton and Stam [69] (SS) falls also in this general framework although the matrix

$$\tilde{A} = \begin{pmatrix} 1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0 \\ 1\,1\,1\,1\,0 \\ 1\,0\,1\,0\,1 \end{pmatrix}$$

(over $\mathbb{F}_{2^{256}}$) does not satisfy the independence criterion imposed by Rogaway and Steinberger. We remark here that the compression function of Shrimpton and Stam also enjoys almost optimal collision resistance (yet suboptimal preimage resistance). In our benchmark, we include all the three proposed schemes in this category. The security of LP-schemes has been extensively studied in [72,66,44].

LANE*. The LANE [36] is a permutation based hash function design (see Fig. 11) submitted to the SHA-3 competition by Indesteege and Preneel. Although some weaknesses have been exploited [50] for the proposed version (that led to the elimination from the selection process) its generic compression function is worth reconsidering due to its capabilities for high parallelism and suitability for the AES instruction set. Indeed, LANE-256, the submitted version using AES-based permutations, was recently shown to be one of the fastest AES-based SHA-3 candidates when using the AES instruction set [3]. For our purposes, we consider its 256-bit digest version which is instantiated by fixed-key RIJNDAEL-256 and denoted by LANE*. The security of the LANE* is known to be sub-optimal: a yield-based adversary can be shown to find collisions and preimages with $O(2^{n/6})$ and $O(2^{n/3})$ queries respectively for n-bit digest. Yet, there is still no known algorithm to find collisions and preimages with time complexity less than the generic attacks.

LUFFA*. LUFFA [21] is a second round SHA-3 candidate designed by De Cannière and Watanabe. We consider the LUFFA compression function as another permutation based function that might possibly benefit from AES-instruction set once its underlying permutations are modified accordingly (see Fig. 9). To this end, we instantiate the three underlying permutations of LUFFA with fixed-key RIJNDAEL-256 and denote this version by LUFFA*. The security analysis of LUFFA borrows characteristics from the sponge framework [8,1] assuming that the underlying permutations are ideal. We note that the compression function of LUFFA is not ideal and the hash function is secure only in the iteration. Moreover, the compression function of LUFFA* outputs 768 bits and one requires an output transformation to reduce the digest to 256-bit.

4 Implementation Considerations and Benchmark Results

The constructions from Section 3 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 AES instructions. The measurements have been carried out analogously to [31]; 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 many millions data-dependent calls to the compression function measured.

Table 2 presents the performance numbers, expressed in cycles per byte, and the security levels for the considered constructions, the relevant SHA-3 candidates and SHA-256. Security claims are given in accordance with the conventions introduced in Section 2 and the results from Section 3. Briefly, lower bounds correspond to the proven query complexity lower bounds and query/time complexity upper bounds correspond to the existing attacks matching these bounds. Although we took care to schedule the instructions in our C-implementation, the presented performance numbers can certainly be optimized when carefully implementing them directly in assembly. In order to reproduce our results we plan to make all our source code publicly available.

The SHA-256 results have been obtained with the implementation from the crypto++ library [23]. As far as we are aware, this implementation results in the best performance for this platform. The performance numbers for the second round SHA-3 candidates are obtained with the submitted codes to eBACS [5] run on our target platform. We note that we include the compression functions of LUFFA* and LANE* to our benchmark instantiated by fixed-key RIJNDAEL-256. The original version of LUFFA has almost identical performance (10.49 cycles per byte) using the fastest implementation submitted to eBASH (SSSE3-PS-2). The original version of LANE, on the other hand, performs significantly faster (4.3 cycles per byte) due to the relatively light permutations in the submitted version.

Throughput Considerations. Many of the constructions described in Section 3 require to compute more than one call to a blockcipher operating on n bits (for $n \in \{128, 256\}$). If these two or more blockciphers can run independent from each other, while possibly sharing the key expansion, a performance gain can be expected in a software implementation (similar holds for the permutation-based setting). The AES round instructions are pipelined and can be dispatched theoretically every 1-2 CPU clock cycles when there is no dependency between such subsequent calls and provided that all data is available [30]. The latency of these instructions is 5 cycles [27]. Hence, running multiple independent blockciphers reduces data dependencies between the subsequent AES round function calls and increases the overall throughput. The same reasoning holds when implementing a single RIJNDAEL-256 component. This variant of the AES works on an internal state of 256 bits which is implemented using two data-independent calls to AESENC increasing the overall throughput.

Polynomial Multiplication. In some of the compression function designs we consider, one of the crucial components is the polynomial multiplication over a finite field, in particular over $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$. Multiplication in $\mathbb{F}_{2^{128}}$ is implemented using the code examples as described in [32] 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 [27]. 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 speedup is in the case of polynomial multiplication by x, represented as the hexadecimal number $0\text{x}2$. Essentially the result can be obtained by shifting the input one position to the left (the multiplication by x) and perform 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. Given an input A we

1. swap the two 64 bit halves of A to t (PSHUFD),

Table 2. The comparison of all the relevant multi-block length designs. The first column shows the corresponding algorithm where the digest size is 256-bit. The number of b bytes which are absorbed per compression function call is shown together with the primitive employed, the achieved speed using the AES instructions (with or without fixed-key) and how many unique key scheduling calls are made. The remainder of the table shows the known results in terms of the collision and the preimage resistance. The relevant entries illustrate the bit security. Note that the ECHO compression function can be regarded as a blockcipher based compression function; it nevertheless employ a fixed-key schedule hence it is treated as a permutation based compression function.

Algorithm	b	**Primitive**	**Key Scheduling**	**Speed**	**Collision Resistance**						**Preimage Resistance**					
					Compression Function			Mode of Operation			Compression Function			Mode of Operation		
					Query		Time	Query		Time	Query		Time	Query		Time
				Cycles per byte	Lower Bound	Upper Bound		Lower Bound	Upper Bound		Lower Bound	Upper Bound		Lower Bound	Upper Bound	
SHA-256	64	blockcipher	one	15.74	127	128	128	127	128	128	255	256	256	255	256	256
ECHO-SP	224	permutation	fixed	5.37	127	128	128	127	128	128	255	256	256	255	256	256
ECHO-DP	192	permutation	fixed	6.22	255	256	256	128	128	128	511	512	512	256	256	256
FUGUE	4	permutation	fixed	15.99	1	1	1	-	128	128	0	0	0	-	256	256
GRØSTL	64	permutation	fixed	13.63	127	128	170	127	128	128	255	256	256	255	256	256
SHAVITE-3	64	blockcipher	one	5.32	127	128	128	127	128	128	255	256	256	255	256	256
ABREAST-DM	16	AES-256	two	14.04	125	128	128	125	128	128	127	256	256	127	256	256
DM	32	RIJNDAEL-256	one	14.17	127	128	128	127	128	128	255	256	256	255	256	256
HIROSE-DBL	16	AES-256	one, shared	13.06	126	128	128	126	128	128	171	256	256	172	256	256
KNUDSEN-PRENEEL	32	AES-256	four	11.94	-	128	128	-	128	128	255	256	256	255	256	256
LANE*	64	RIJNDAEL-256	fixed	12.03	-	43	128	-	128	128	-	86	256	-	256	256
LP231	32	RIJNDAEL-256	fixed	18.89	127	128	128	127	128	128	172	172	256	172	181	256
LP362	16	AES-128	fixed	12.11	81	85	128	81	110	128	103	107	256	103	116	256
LUFFA*	32	RIJNDAEL-256	fixed	10.40	1	2	2	-	128	128	0	0	1	-	256	256
MDC-2	16	AES-128	two	11.54	63	64	64	76	120	120	-	128	128	-	128	128
MJH	16	AES-128	one, shared	10.66	63	64	64	80	128	128	-	128	128	-	128	128
MJH-DOUBLE	32	AES-256	one, shared	6.79	63	64	64	80	128	128	-	128	128	-	128	128
QPB-DBL	16	AES-256	one	19.57	120	128	128	120	128	128	-	256	256	-	256	256
PEYRIN ET AL.(I)	16	AES-128	three, shared	15.92	86	86	128	86	128	128	172	172	256	172	172	256
PEYRIN ET AL.(II)	32	AES-256	three, shared	9.83	86	96	128	86	128	128	172	172	256	172	172	256
SS	32	RIJNDAEL-256	fixed	12.96	127	128	128	127	128	128	127	172	256	127	181	256

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

Here R denotes the hexadecimal representation of the reduction polynomial stored in a 64-bit word. Note that this computation might be speed 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 and 3 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. Shifting a 256-bit vector is done in a similar fashion, the total required number of instructions in this case is ten.

Key Expansion Cost. When using the AES-NI, the cost of the AES round function is highly reduced but the cost of the key-expansion remains significant. In most circumstances in practice, where AES is used, this is not an issue since the key expansion is typically done once and many block encryptions/decryptions are performed with this same key. For blockcipher based compression functions considered in this paper, the key expansion needs to be performed for every compression function evaluation. To illustrate the relatively expansive cost of the key expansion we show the performance numbers obtained by Intel using their assembly implementation [31]. These numbers have been obtained when running on a Westmere-based processor running at 2.67GHz. The performance numbers, expressed in cycles per byte, are obtained when running AES in ECB mode and processing a 1KB buffer.

	key expansion	encryption
AES-128	6.75	1.28
AES-256	8.50	1.76

This shows that the constructions which require multiple calls to the key scheduling routines per compression function invocation pay a significant performance price. In order to avoid such a performance penalty, constructions might use a single key expansion shared among multiple invocations or use a fixed key to completely eliminate this cost, e.g. all permutation based compression functions (see Table 2 for a more elaborate comparison of key scheduling needs for the compression functions we consider).

5 Discussion and Conclusion

We briefly discuss the results provided in Table 2 and conclude the paper. The conclusions of our work are several fold:

1. Using the current compression function designs it is possible to construct *fast* cryptographic hash functions with provable security reductions to an underlying primitive. Indeed, most of the target algorithms outperform the current cryptographic hash standard SHA-256 on our target platform. Moreover, many of the considered compression functions are faster compared to the second-round AES-inspired SHA-3 candidates GRØSTL and FUGUE whereas ECHO and SHAVITE-3 outperform all the constructions we considered. The latter result shows that ECHO and SHAVITE-3 do benefit effectively from the AES-NI.
2. Many of the multi-block length blockcipher based compression functions outperform the DM (when used with a single-key). This shows that on modern architectures, supporting high parallelism, one can achieve better performance, without loosing too much security, by using smaller blockciphers that are run concurrently. We expect that the situation is analogous for hardware implementations as

well. This result further motivates research on the design on multi block-length compression/hash functions.

3. The cost of key-scheduling might affect the performance of the algorithms considerably[12]. For instance, consider the performance of ABREAST-DM and HIROSE-DBL (with the same underlying primitive and almost the same security level), the performance gain is seven percent in HIROSE-DBL. Note however that there might be a possibility for blockcipher based compression functions to increase their performance. Namely, one might increase the number of message bits compressed per compression function evaluation by increasing the key size of the underlying blockcipher *without* violating the security proof. Obviously, this can affect the performance significantly assuming that the key scheduling cost and the number of rounds used in the blockcipher do not increase too much. The compression functions MJH-DOUBLE and the double-key version of DM fall in this category[13] by using AES-256 instead of AES-128. On the other hand, removing the key-schedule altogether, by moving to the permutation based setting, results in more calls to the underlying primitives in order to guarantee the same security level. Still, one can generally achieve comparable performance, in particular when the permutations can be run independently. Nevertheless, we believe that using larger key blockciphers in DM or MJH-DOUBLE would result in a better performance (although the design of key-scheduling algorithms are not well understood and generally leads to weaker schemes).
4. Among the blockcipher-based compression functions, HIROSE-DBL is the fastest algorithm when optimal security (in terms of proven lower bound) is desired. For the practical security level (considering the time complexity upper bound) MJH-DOUBLE and PEYRIN ET AL.(II) outperform the others including the permutation based compression functions. Here, the former makes use of the fact that it compresses more message bits by increasing the key size (see the conclusion stated above), whereas the latter benefits from high parallelism by compressing a high number of message bytes using multiple independent primitives.
5. In the permutation based setting, the LUFFA* compression function is the fastest. Note that it is one of the rare compression functions outputting more than 256 bits yet having a considerable speed. When comparing the performance between the different constructions in the ROGAWAY-STEINBERGER'S LP framework it is interesting to note that SS outperforms LP231. This is only partially because the former requires no polynomial multiplication by 0×2. The main speed-up is due to the fact that the first two fixed-key RIJNDAEL-256 permutations are independent and can be run concurrently while this is not possible in LP231. The same argument holds in the setting of LP362 which, despite requiring some polynomial multiplications, allows to run multiple faster (fixed-key AES-128) permutations in parallel. It is an interesting open problem to find more efficient ROGAWAY-STEINBERGER'S LP constructions that allows to run many permutations in parallel.
6. As briefly mentioned above, one of the key components to achieve high performance is to increase the number of message bits compressed per compression function evaluation, even if the number of (parallel) primitive calls have to be increased to assure a certain security level. Indeed, there are already some algorithms among our targets that benefit from this idea: LUFFA*, LANE*, PEYRIN ET AL.(II) and KNUDSEN-PRENEEL perform quite well on our benchmark platform. We believe that it is worth further studying compression functions of this form with possibly more parallel calls to the underlying primitives and outputting variable digest sizes in order to achieve better performance by compressing more bits.
7. 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. 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

[12] As shown in Section 4, AES has a similar problem; we believe that modifying the AES key scheduling in a secure and efficient way is a challenging open problem.

[13] The compression functions of ECHO and SHAVITE-3 also benefit from this feature.

that our benchmark provides a valuable toolbox to see the relative performance figures for almost all blockcipher and permutation based compression functions. So, one can use the results from Table 2 to evaluate both security and performance of any construction of this form and derive clear conclusions once a primitive with certain properties is designed.

Acknowledgements. This work was supported by the Swiss National Science Foundation under grant numbers 200020-132160, 200021-119776, and 200021-122162. We gratefully acknowledge Çağdaş Çalık, for granting us access to the Intel i5 with AES-NI to benchmark our programs, Thorsten Kleinjung, for useful discussions on how to optimize the SSE polynomial multiplication by $0\text{x}2$, and Martijn Stam, for useful discussions regarding MJH and MJH-DOUBLE.

References

1. E. Andreeva, B. Mennink, and B. Preneel. Security reductions of the second round SHA-3 candidates. Cryptology ePrint Archive, Report 2010/381, 2010. `http://eprint.iacr.org/`.
2. R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 proposal: ECHO, 2009.
3. R. Benadjila, O. Billet, S. Gueron, and M. J. B. Robshaw. The Intel AES instructions set and the SHA-3 candidates. In *Asiacrypt 2009*, volume 5912 of *LNCS*, pages 162–178. Springer, 2009.
4. D. J. Bernstein. Cache-timing attacks on AES, 2005. `http://cr.yp.to/papers.html#cachetiming`.
5. D. J. Bernstein and T. Lange, (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, 2010.
6. D. J. Bernstein and P. Schwabe. New AES software speed records. In *Indocrypt 2008*, volume 5365 of *LNCS*, pages 322–336. Springer, 2008.
7. G. Bertoni, L. Breveglieri, R. Farina, and F. Regazzoni. Speeding up AES by extending a 32 bit processor instruction set. In *Application-specific Systems, Architectures and Processors*, pages 275–282. IEEE Computer Society, 2006.
8. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the indifferentiability of the sponge construction. In *Eurocrypt 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, 2008.
9. E. Biham. A fast new DES implementation in software. In *FSE 1997*, volume 1267 of *LNCS*, pages 260–272, 1997.
10. E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of SHA-0 and reduced SHA-1. In *Eurocrypt 2005*, volume 3494 of *LNCS*, pages 36–57. Springer, 2005.
11. E. Biham and O. Dunkelman. A framework for iterative hash functions – HAIFA. Presented at *Second NIST Cryptographic Hash Workshop, August 24–25, 2006, Santa Barbara, California, USA*.
12. E. Biham and O. Dunkelman. The SHAvite-3 hash function, 2009.
13. A. Biryukov and D. Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In *Asiacrypt 2009*, volume 5912 of *LNCS*, pages 1–18. Springer, 2009.
14. A. Biryukov, D. Khovratovich, and I. Nikolic. Distinguisher and related-key attack on the full AES-256. In *Crypto 2009*, volume 5677 of *LNCS*, pages 231–249, 2009.
15. J. Black, M. Cochran, and T. Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. *Journal of Cryptology*, 22(3):311–329, 2009.
16. J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *Crypto 2002*, volume 2442 of *LNCS*, pages 320–335. Springer, 2002.
17. J. Black, P. Rogaway, T. Shrimpton, and M. Stam. An analysis of the blockcipher-based hash functions from PGV. *Journal of Cryptology*, 23(4):519–545, 2010.
18. A. Bogdanov, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, and Y. Seurin. Hash functions and RFID tags: Mind the gap. In E. Oswald and P. Rohatgi, editors, *CHES '08*, volume 5154 of *LNCS*, pages 283–299. Springer, 2008.
19. J. W. Bos and D. Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In *CHES 2010*, volume 6225 of *LNCS*, pages 279–293. Springer, 2010.
20. B. Brachtl, D. Coppersmith, M. Hyden, S. Matyas, Jr., C. Meyer, J. Oseas, S. Pilpel, and M. Schilling. Data authentication using modification detection codes based on a public one-way encryption function. U.S. Patent No 4,908,861, 1990.
21. C. D. Canniere, H. Sato, and D. Watanabe. Hash function Luffa: Supporting document. Submission to NIST (Round 2), 2009.
22. J. Daemen and V. Rijmen. *The design of Rijndael*. Springer, 2002.
23. W. Dai. Crypto++ library. `http://www.cryptopp.com`, 2010.
24. I. Damgård. A design principle for hash functions. In *Crypto 1989*, volume 435 of *LNCS*, pages 416–427. Springer, 1990.
25. Ecrypt II yearly report on algorithms and key sizes. Ecrypt II European Network of Excellence in Cryptology II, 30 March 2010.
26. E. Fleischmann, M. Gorski, and S. Lucks. Security of cyclic double block length hash functions. In *Cryptography and Coding 2009*, volume 5921 of *LNCS*, pages 153–175. Springer, 2009.

27. A. Fog. Instruction tables, lists of instruction latencies, throughputs and microoperation breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/, 2010.
28. I. O. for Standardization. ISO/IEC 10118-2:2000.information technology-security techniques-hash functions-hash functions using an n-bit block cipher. International Organization for Standardization, Geneva, Switzerland, 2000, First released in 1992.
29. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate, 2008.
30. S. Gueron. Intel's new AES instructions for enhanced performance and security. In *FSE 2009*, volume 5665 of *LNCS*, pages 51–66. Springer, 2009.
31. S. Gueron. Intel advanced encryption standard (AES) instructions set. Technical report, Intel, 2010. http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/.
32. S. Gueron and M. E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. Technical report, Intel, 2010. http://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode/.
33. S. Halevi, W. E. Hall, and C. S. Jutla. The hash function Fugue, 2009.
34. M. Hamburg. Accelerating AES with vector permute instructions. In *CHES*, volume 5747 of *LNCS*, pages 18–32. Springer, 2009.
35. S. Hirose. Some plausible constructions of double-block-length hash functions. In *FSE 2006*, volume 4047 of *LNCS*, pages 210–225. Springer, 2006.
36. S. Indesteege. The LANE hash function. Submission to NIST, 2008.
37. A. N. S. Institute. Public key cryptography using reversible algorithms for the financial services industry. American National Standards Institute, 1998.
38. E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *CHES 2009*, volume 5747 of *LNCS*, pages 1–17, 2009.
39. L. R. Knudsen, F. Mendel, C. Rechberger, and S. S. Thomsen. Cryptanalysis of MDC-2. In *Eurocrypt 2009*, volume 5479 of *LNCS*, pages 106–120. Springer, 2009.
40. L. R. Knudsen and B. Preneel. Construction of secure and fast hash functions using nonbinary error-correcting codes. *IEEE Transactions on Information Theory*, 48(9):2524–2539, 2002.
41. M. Krause, F. Armknecht, and E. Fleischmann. Preimage resistance beyond the birthday barrier – the case of blockcipher based hashing. Cryptology ePrint Archive, Report 2010/519, 2010. http://eprint.iacr.org/.
42. X. Lai and J. L. Massey. Hash functions based on block ciphers. In *Eurocrypt '92*, volume 658 of *LNCS*, pages 55–70. Springer, 1993.
43. J. Lee and D. Kwon. The security of Abreast-DM in the ideal cipher model. Cryptology ePrint Archive, Report 2009/225, 2009. http://eprint.iacr.org/.
44. J. Lee and J. H. Park. Preimage resistance of LPm kr with $r = m-1$. *Information Processing Letters*, 110(14-15):602–608, 2010.
45. J. Lee and M. Stam. A faster alternative to MDC-2. In *To appear in CT-RSA '11*, LNCS. Springer, 2011.
46. J. Lee, M. Stam, and J. Steinberger. The collision security of Tandem-DM in the ideal cipher model. Cryptology ePrint Archive, Report 2010/409, 2010. http://eprint.iacr.org/.
47. J. Lee and J. P. Steinberger. Multi-property-preserving domain extension using polynomial-based modes of operation. In *Eurocrypt 2010*, volume 6110 of *LNCS*, pages 573–596. Springer, 2010.
48. S. Lucks. A failure-friendly design principle for hash functions. In *ASIACRYPT '05*, volume 3788 of *LNCS*, pages 474–494. Springer, 2005.
49. R. Manley, P. Magrath, and D. Gregg. Code generation for hardware accelerated AES. pages 345–348, 2010.
50. K. Matusiewicz, M. Naya-Plasencia, I. Nikolic, Y. Sasaki, and M. Schläffer. Rebound attack on the full Lane compression function. In *Asiacrypt 2009*, volume 5912 of *LNCS*, pages 106–125. Springer, 2009.
51. A. Menezes, P. van Oorschot, and S. Vanstone. *CRC-Handbook of Applied Cryptography*. CRC Press, 1996.
52. R. C. Merkle. One way hash functions and DES. In *CRYPTO 1989*, volume 435 of *LNCS*, pages 428–446. Springer, 1990.
53. NIST. FIPS-197: Advanced encryption standard (AES), 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
54. NIST. Secure hash standard. FIPS 180-2, http://www.itl.nist.gov/fipspubs/fip180-2.htm, August 2002.
55. NIST. Cryptographic hash algorithm competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html, 2008.
56. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *FSE 2010*, volume 6147 of *LNCS*, pages 75–93. Springer, 2010.
57. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
58. O. Özen, T. Shrimpton, and M. Stam. Attacking the Knudsen-Preneel compression functions. In *FSE 2010*, volume 6147 of *LNCS*, pages 94–115. Springer, 2010.

59. O. Özen and M. Stam. Another glance at double-length hashing. In *Cryptography and Coding 2009*, volume 5921 of *LNCS*, pages 176–201. Springer, 2009.
60. O. Özen and M. Stam. Collision attacks against the Knudsen-Preneel compression functions. In *To appear in Asiacrypt 2010*, LNCS. Springer, 2010.
61. T. Peyrin, H. Gilbert, F. Muller, and M. J. B. Robshaw. Combining compression functions and block cipher-based hash functions. In *Asiacrypt 2006*, volume 4284 of *LNCS*, pages 315–331. Springer, 2006.
62. B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In *Crypto '93*, volume 773 of *LNCS*, pages 368–378. Springer, 1994.
63. M. O. Rabin. Digitalized signatures. In *Foundations of Secure Computations*, pages 155–166. Academic Press, 1978.
64. R. Rivest. The MD5 message-digest algorithm, request for comments (RFC) 1320. Technical report, Internet Activities Board, Internet Privacy Task Force, 1992.
65. P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In B. K. Roy and W. Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 371–388. Springer, 2004.
66. P. Rogaway and J. Steinberger. Security/efficiency tradeoffs for permutation-based hashing. In *Eurocrypt 2008*, volume 4965 of *LNCS*, pages 220–236. Springer, 2008.
67. P. Rogaway and J. P. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In *Crypto 2008*, volume 5157 of *LNCS*, pages 433–450. Springer, 2008.
68. Y. Seurin and T. Peyrin. Security analysis of constructions combining FIL random oracles. In *FSE 2007*, volume 4593 of *LNCS*, pages 119–136. Springer, 2007.
69. T. Shrimpton and M. Stam. Building a collision-resistant compression function from non-compressing primitives. In *International Colloquium on Automata, Languages and Programming 2008*, volume 5126 of *LNCS*, pages 643–654. Springer, 2008.
70. M. Stam. Beyond uniformity: Better security/efficiency tradeoffs for compression functions. In *Crypto 2008*, volume 5157 of *LNCS*, pages 397–412. Springer, 2008.
71. M. Stam. Blockcipher-based hashing revisited. In *FSE 2009*, volume 5665 of *LNCS*, pages 67–83. Springer, 2009.
72. J. P. Steinberger. The collision intractability of MDC-2 in the ideal-cipher model. In *Eurocrypt 2007*, volume 4515 of *LNCS*, pages 34–51. Springer, 2007.
73. S. Tillich and J. Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. In *CHES 2006*, volume 4249 of *LNCS*, pages 270–284. Springer, 2006.
74. S. Tillich, J. Großschädl, and A. Szekely. An instruction set extension for fast and memory-efficient AES implementation. In *Communications and Multimedia Security*, volume 3677 of *LNCS*, pages 11–21. Springer, 2005.
75. S. Tillich and C. Herbst. Boosting AES performance on a tiny processor core. In *CT-RSA*, volume 4964 of *LNCS*, pages 170–186, 2008.
76. E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23:37–71, 2010.
77. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Crypto 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
78. X. Wang and H. Yu. How to break MD5 and other hash functions. In *Eurocrypt 2005*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.
79. H. Wu. The hash function JH. Submission to NIST (updated), 2009.

A Illustrations of Related Compression Functions

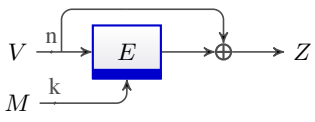

Fig. 1. The DM compression function, $n = k = 256$. The underlying blockcipher is RIJNDAEL-256.

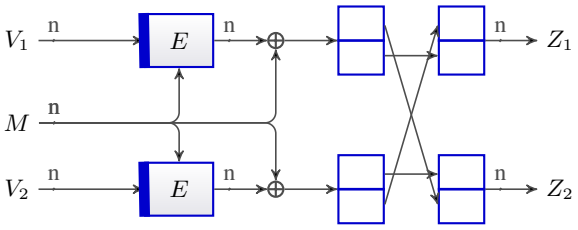

Fig. 2. The MDC-2 compression function, $n = 128$. The underlying blockcipher is AES-128.

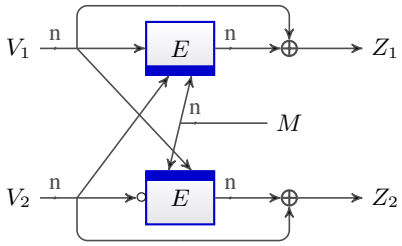

Fig. 3. The ABREAST-DM compression function, $n = 128$. The underlying blockcipher is AES-256 and $\circ$ denotes the bitwise complementation.

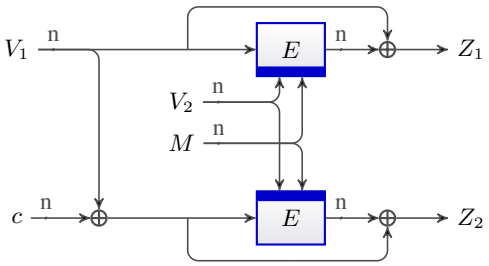

Fig. 4. The HIROSE-DBL compression function, $n = 128$. The underlying blockcipher is AES-256 and $c \in \{0,1\}^n \setminus \{0\}^n$.

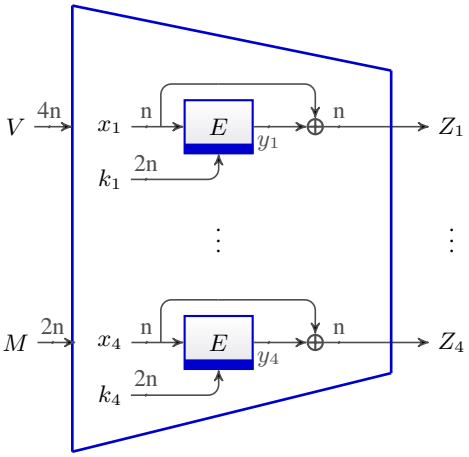

Fig. 5. The KNUDSEN-PRENEEL compression function, $n = 128$. The underlying blockcipher is AES-256. For $V = V_1||\ldots||V_4 \in \{0,1\}^{512}$, $M = M_1||M_2 \in \{0,1\}^{256}$ and different constants $c_i \in \{0,1\}^{128} \setminus \{0\}^{128}$, we define:
$(x_1, k_1) = (V_1 \oplus c_1, V_2||V_3), \quad (x_2, k_2) = (V_4 \oplus c_2, M_1||M_2),$
$(x_3, k_3) = (V_3 \oplus M_1 \oplus c_3, V_1 \oplus V_2 \oplus M_2||V_2 \oplus V_3 \oplus V_4 \oplus M_1),$
$(x_4, k_4) = (V_1 \oplus V_3 \oplus M_1 \oplus M_2 \oplus c_4, V_1 \oplus V_4 \oplus M_1 \oplus M_2||V_2 \oplus V_4 \oplus M_2).$

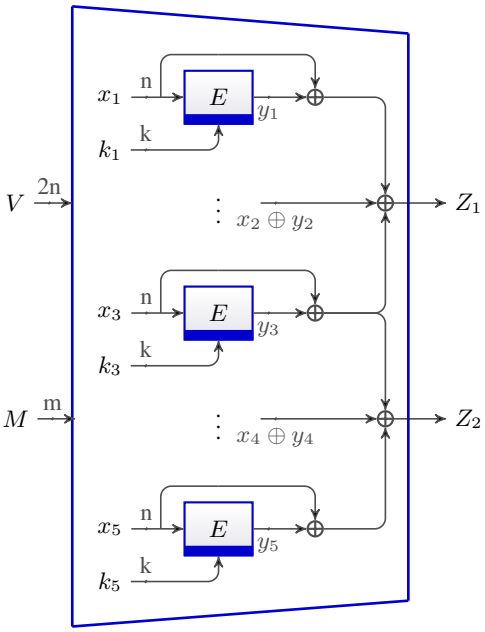

Fig. 6. The PEYRIN ET AL. compression function $H : \{0,1\}^{2n+m} \rightarrow \{0,1\}^{2n}$. The underlying five blockciphers operate on n bits and they support k-bit keys each. We consider two scenarios (for different constants $c_i \in \{0,1\}^{128} \setminus \{0\}^{128}$):
(i) $m = k = n = 128$ (using AES-128), for $V = V_1||V_2$ set
$(x_1, k_1) = (V_1 \oplus c_1, V_2)$, $(x_2, k_2) = (V_2 \oplus c_2, M)$, $(x_3, k_3) = (M \oplus c_3, V_1 \oplus V_2)$,
$(x_4, k_4) = (V_1 \oplus c_4, M)$, $(x_5, k_5) = (V_1 \oplus c_5, V_2)$.
(ii) $m = k = 2n = 256$ (using AES-256), for $V = V_1||V_2$, $M = M_1||M_2$ set
$(x_1, k_1) = (V_1 \oplus c_1, V_2||M_1)$, $(x_2, k_2) = (V_1 \oplus c_2, V_2||M_2)$, $(x_3, k_3) = (V_1 \oplus c_3, M_1||M_2)$,
$(x_4, k_4) = (V_1 \oplus c_4, V_2||M_1)$, $(x_5, k_5) = (V_2 \oplus c_5, M_1||M_2)$.

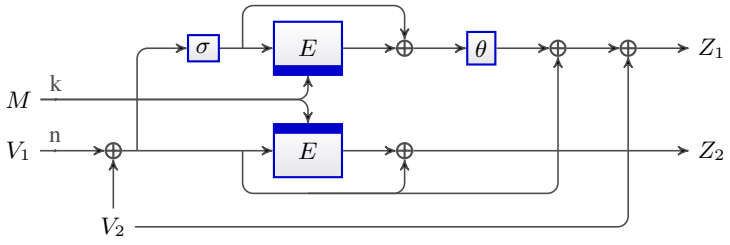

Fig. 7. The MJH and MJH-DOUBLE compression functions for $n = 128$. In the former, $k = n = 128$ whereas in the latter $k = 2n = 256$. The underlying blockciphers are AES-128 and AES-256 respectively, σ is an involution (e.g., an addition with a non-zero constant) and θ is a multiplication with a constant in $\mathbb{F}_{2^{128}}$.

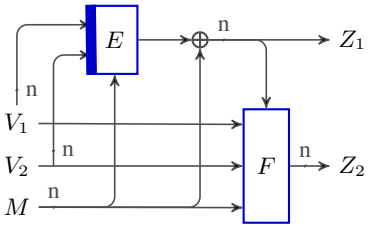

Fig. 8. The QPB-DBL compression function, $n = 128$. The underlying blockcipher is AES-256 and $Z_2 = V_1 Z_1^2 + V_2 Z_1 + M$.

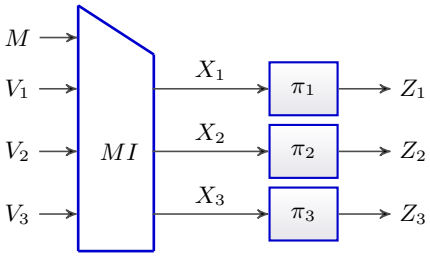

Fig. 9. The LUFFA-256 compression function, the horizontal lines carry 256 bits. The underlying permutations are obtained from the fixed-key RIJNDAEL-256 with varying keys. The message injection step MI is defined as follows: $X_i = V_i \oplus (0\text{x}02 \cdot (V_1 \oplus V_2 \oplus V_3)) \oplus 0\text{x}02^i \cdot M$, for $i \leq 1 \leq 3$. Note that the hash function outputs 256 bits by performing an output transformation. We refer to [21] for all details, especially how the multiplication by $0\text{x}02$ is defined.

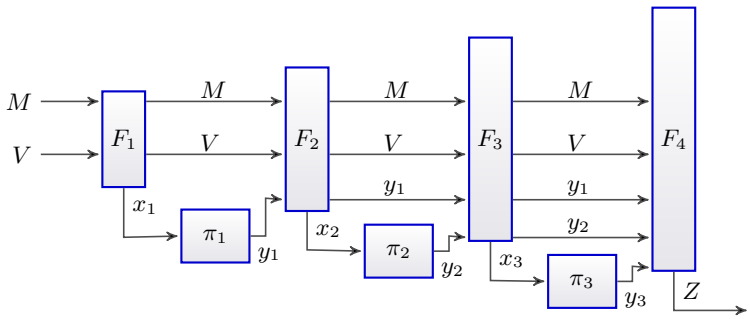

Fig. 10. The ROGAWAY-STEINBERGER framework of compression functions. The LP231 and SS compression function are illustrated, all lines carry 256 bits. The underlying fixed-key permutations π_1, π_2, π_3 are derived from RIJNDAEL-256. The F_i compute the x_i as follows (see Sec. 3 for the specific values for a_j):
$x_1 = a_{11} \cdot M + a_{12} \cdot V$, $x_3 = a_{31} \cdot M + a_{32} \cdot V + a_{33} \cdot y_1 + a_{34} \cdot y_2$,
$x_2 = a_{21} \cdot M + a_{22} \cdot V + a_{23} \cdot y_1$, $x_4 = a_{41} \cdot M + a_{42} \cdot V + a_{43} \cdot y_1 + a_{44} \cdot y_2 + a_{45} \cdot y_3$.
The LP362 compression function works similarly, the lines carry 128 bits and six calls to AES-128 are made. Finally, two 128 bit value are the output.

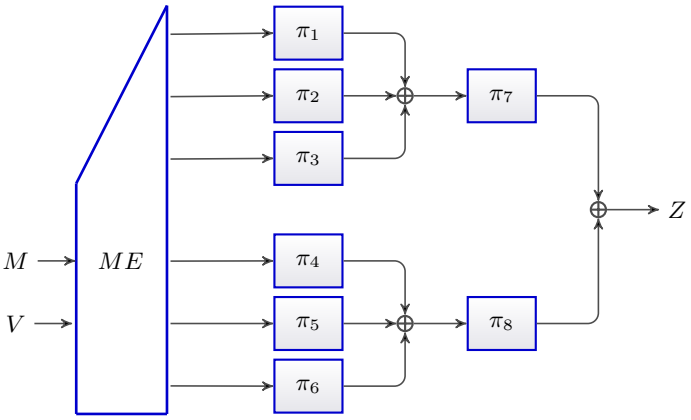

Fig. 11. The LANE-256 compression function with $M \in \{0,1\}^{512}$ and $V, Z \in \{0,1\}^{256}$. Horizontal lines carry 256 bits. The underlying permutations are obtained from the fixed-key RIJNDAEL-256 with varying keys. Here ME denotes the so called message expansion algorithm which is based on a $[6,3,4]$ linear error correcting code over the finite field $\mathbb{F}_4$. We refer to [36] for the exact specification of ME.