

Hardware Evaluation of the Luffa Hash Family

Miroslav Knežević and Ingrid Verbauwhede
Katholieke Universiteit Leuven

Department of Electrical Engineering - ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{miroslav.knezevic,ingrid.verbauwhede}@esat.kuleuven.be

ABSTRACT

Efficient hardware architectures for the *Luffa* hash algorithm are proposed in this work. We explore different trade-offs and propose several architectures, targeting both compact and high-throughput designs. Implemented using UMC 0.13 μm CMOS standard cell library, the most compact architecture of *Luffa*-224/256 contains 18,260 GE. The same version, optimized for speed, achieves a throughput of almost 32 Gbps, while the throughput of the pipelined design approaches 291.7 Gbps. Concerning the final throughput, our implementations outperform state of the art implementations of the existing hash standards.

Keywords

SHA-3 competition, *Luffa* hash algorithm, ASIC implementations.

1. INTRODUCTION

Hash functions are mathematical algorithms that transform arbitrary length sequences of bits into a hash result of a fixed, limited length (*e.g.* 256 bits), in a way that these hash results can be considered as unique "fingerprints". Hash functions are used in many cryptographic protocols, including digital signatures, the detection of file tampering, the verification of passwords or the derivation of cryptographic keys. Therefore, it is very important that they fulfill certain security properties. Those properties can be considered as follows. *Preimage Resistance*: It must be hard to find any preimage for a given hash output, *i.e.* given a hash output H getting M must be hard such that $H = \text{Hash}(M)$. *Second Preimage Resistance*: It must be hard to find another preimage for a given input, *i.e.* given M_0 and $\text{Hash}(M_0)$ getting M_1 must be hard such that $\text{Hash}(M_0) = \text{Hash}(M_1)$. *Collision Resistance*: It must be hard to find two different inputs of the same hash output, *i.e.* getting M_0 and M_1 must be hard such that $\text{Hash}(M_0) = \text{Hash}(M_1)$.

In response to the recent attacks on the most popular hash algorithms such as SHA-1 and MD5 [10, 11] the National

Institute of Standards and Technology (NIST) launched a worldwide competition for the development of a new hash function. The competition has received 64 submissions, 51 of which have advanced to the first round. During the Round one, many of the algorithms were analyzed from the security point of view and recently NIST has selected the second round candidates, narrowing the final choice down to only 14 hash proposals [9]. A family of cryptographic functions *Luffa*, proposed by Watanabe et. al. [1], has made through and is one of the Round two candidates.

Efficient hardware architectures for the *Luffa* hash algorithm are the topic of this work. We explore some of the possible trade-offs and propose several architectures, targeting both compact and high-throughput designs. The most compact architecture of 18,260 gate equivalences (GE) was achieved for the 224/256-bit version of *Luffa*. The same version, optimized for speed, achieves a throughput of almost 32 Gbps, while the pipelined design approaches the throughput of 291.7 Gbps. Techniques such as retiming, pipelining and simple multiplexing were used for the high-throughput, pipelined and compact implementations, respectively.

The remainder of this paper is structured as follows. Section 2 gives a short description of the *Luffa* hash algorithm. In Sect. 3 we give the figures of the hardware performance including both compact and high-throughput designs, and some of the possible trade-offs. Section 4 concludes the paper.

2. LUFFA HASH ALGORITHM

Figure 1 illustrates a generic construction of the *Luffa* hash algorithm. It consists of the intermediate mixing C' (called a *round function*) and the *finalization* C'' . The round function is a composition of a *message injection function* MI and a *permutation* P of w 256-bit inputs as shown in Fig. 2 (*Luffa*-224/256 variant). The permutation is divided into multiple sub-permutation Q_j of 256-bit inputs.

The family of hash functions *Luffa* consists of four variants specified by the output hash length (224, 256, 384 and 512 bits). A main difference is the number of sub-permutations w , and the message injection function MI used in each of them. The number of sub-permutations equals 3, 4 and 5 for the version of 224/256-bit, 384-bit and 512-bit *Luffa*, respectively.

The finalization C'' consists of iterating an *output function* OF and a (blank) round function with a fixed message $0x00 \dots 0$. A blank round with a fixed message block $0x00 \dots 0$ is always applied at the beginning of the finalization. The output function XORs all the block values and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

4th Workshop on Embedded Systems Security 2009, Grenoble, France.
Copyright 2009 ACM 978-1-60558-700-4 ...\$10.00.

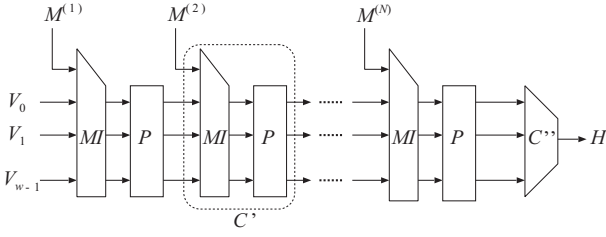


Figure 1: A generic construction of the *Luffa* hash algorithm [1].

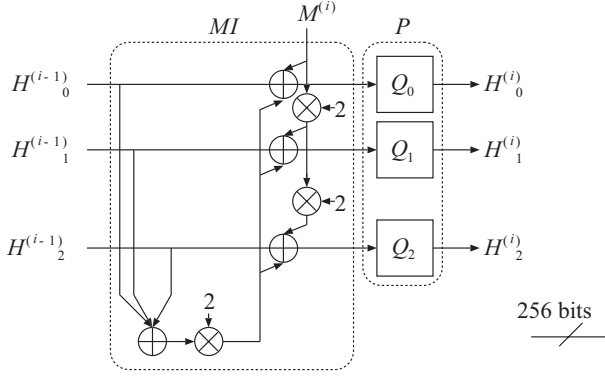


Figure 2: The round function C' ($w = 3$) [1].

outputs the result of 256-bits. Figure 3 illustrates the finalization function. The output of the hash function is defined as Z_0 for the 224/256-bit version, Z_0 concatenated with the most significant half of Z_1 for the 384-bit version, and Z_0 concatenated with Z_1 for the 512-bit version of *Luffa*.

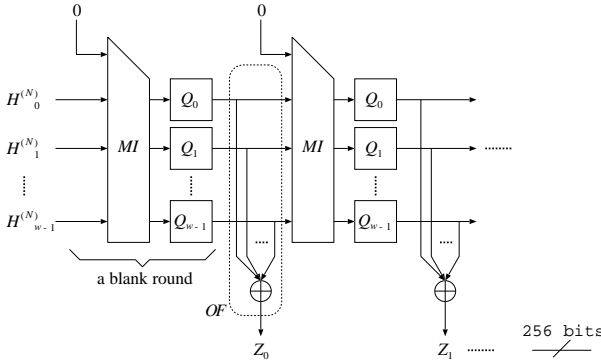


Figure 3: The finalization function C'' [1].

The round function of *Luffa* uses a non-linear permutation Q_j with the input and output size of 256 bits. A main component of the permutation Q_j is the *step function* which consists of *SubCrumb* and *MixWord* blocks, as it is illustrated in Fig. 4. *SubCrumb* block contains 4-bit input Sboxes (see Fig. 5) while *MixWord* represents a linear permutations of two 32-bit words (see Fig. 6). The parameters σ_i are fixed and given as $\sigma_1 = 2$, $\sigma_2 = 14$, $\sigma_3 = 10$, $\sigma_4 = 1$. Finally, *AddConstant* is performed before the output of the step function

is ready. It is a simple XOR with the precalculated constant values. To perform the complete round, one needs to execute the message injection function once and the step function 8 times. For a detailed description of the *Luffa* hash family please refer to [1].

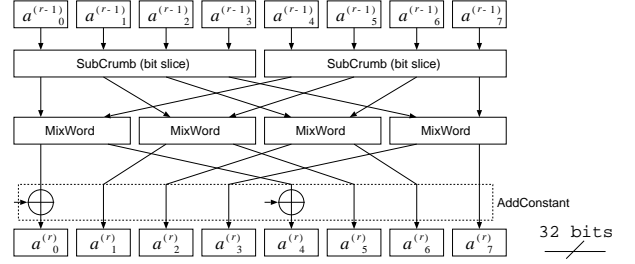


Figure 4: The step function [1].

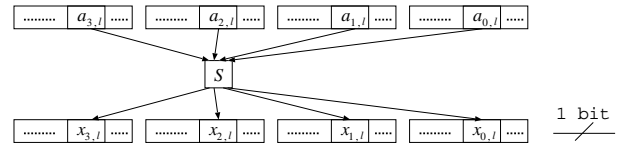


Figure 5: SubCrumb block [1].

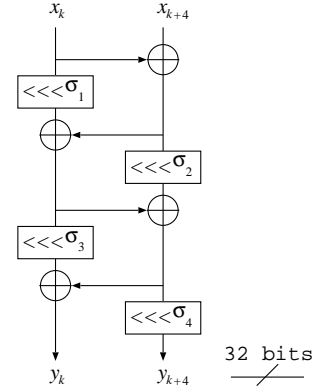


Figure 6: MixWord block [1].

3. HARDWARE IMPLEMENTATION

In this section we propose three different architectures for the *Luffa* hash family. First, we target a high-throughput architecture. Second, we focus on a compact design, and finally we propose a fully pipelined architecture that reaches a throughput of 291.7 Gbps.¹

A hardware performance evaluation of the *Luffa* hash family was done by synthesizing the proposed designs using UMC 0.13 μm CMOS High-Speed standard cell library.

¹This is a fully parallel implementation and achieves the highest throughput in case of hashing 8 independent messages in parallel.

The code was first written in GEZEL [8] and tested for its functionality using the test vectors provided by the software implementations. The GEZEL code was then translated to VHDL and synthesized using the Synopsys Design Vision tool version Y-2006.06.

Note here that the obtained results are based on the synthesis only. After the place and route is performed we expect a slight decrease of the performance of all designs.

3.1 High-Throughput Implementation

For the high-throughput implementation, the goal was to minimize the critical path. To implement the round function, we have used w permutation blocks in parallel, each of them containing 64 Sboxes and 4 MixWord blocks. The straightforward implementation, outlined in Fig. 7, resulted in the critical path of $1.21 ns$ and the cycle count of 8. The critical path was placed from the input of the message injection function to the output of the permutation block (dashed arrow). Inputs and outputs of the step function (permutation block) are denoted as $A_j^{(r)}$ where $1 \leq r \leq 8$ and $0 \leq j \leq w - 1$.

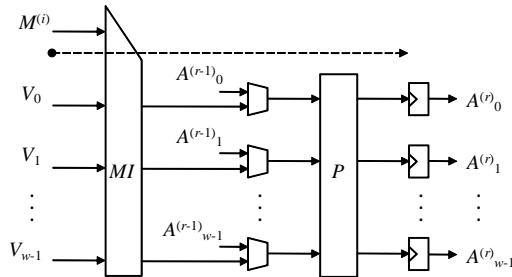


Figure 7: Straightforward implementation of the round function.

As the message injection function is performed only once at the beginning of every round, we moved the state registers at the input of the permutation blocks (see Fig. 8). This technique, called *retiming*, widely used in design of the digital signal processing (DSP) systems, is a transformation technique that changes the locations of unit-delay elements in a circuit without affecting the input/output characteristic of the circuit [6]. It resulted in a faster design, shortening the critical path to only $0.89 ns$. One more clock cycle had to be spent in order to perform the complete round, but the final throughput got increased for about 20 %.

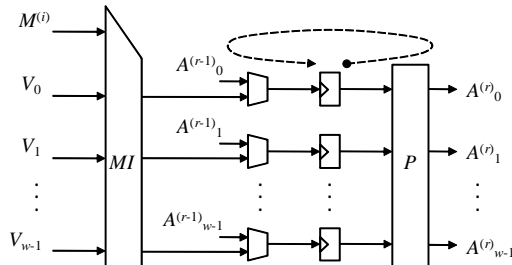


Figure 8: High-throughput implementation of the round function.

The synthesis results are given in Table 1. As the blank round needs to be performed always, it will mostly affect the "one-block" messages². For a very long message, the number of cycles spent for the blank round is negligible and therefore the throughput is doubled compared to the case of "one-block" message. The throughput for "one-block" message is calculated according to the following equation:

$$\text{Throughput}_{\text{OB}} = \frac{\text{Frequency}}{\# \text{ of Cycles} \times 2} \times 256 \text{ bit} ,$$

while the throughput for the very long message is calculated as:

$$\text{Throughput}_{\text{LM}} = \frac{\text{Frequency}}{\# \text{ of Cycles}} \times 256 \text{ bit} .$$

To show some of the possible trade-offs regarding the high-throughput implementation, we have synthesized a number of different designs only by changing the clock frequency and hence, changing the total performance of the design (see Fig. 9). A design of *Luffa*-224/256 has a throughput of 16.7 Gbps and contains only 24.6 kGE.

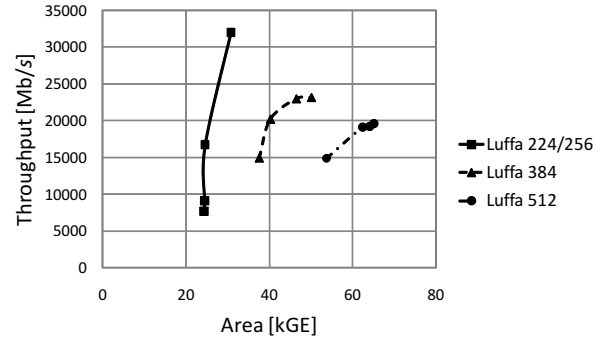


Figure 9: High-Throughput implementations.

3.2 Compact Implementation

A compact implementation was made using only one non-linear permutation block. Inside the permutation block we have used 64 Sboxes and 4 MixWord blocks. This approach resulted in a smaller circuit while decreasing the final throughput. To maintain the internal state, we used w 256-bit registers all of which had 3-to-1 multiplexers at the inputs. The first input was used for the initialization, the second one for the permutation function and finally, the third one was used for the message initialization of the blank round. We also used a clock gating technique to preserve the state of the registers while they were idle.

For each of the *Luffa* hash functions we have synthesized a few versions, some of them with the constraints on area and some with the constraints on speed. The most compact designs are given in Table 2, whereas Fig. 10 further shows the throughput-area trade-offs for the given implementations.

As can be seen from Table 2, the most compact implementation is obtained for *Luffa*-224/256 algorithm and consumes approximately 18.26 kGE. Note that our only goal for the compact implementation was to have a small circuit size, regardless of the final throughput. Hence, we fixed ²"One-block" message is a message of exactly 256 bits after the padding is performed.

Table 1: Throughput optimized implementations of the *Luffa* hash family.

Design	Area [GE]	Frequency [MHz]	# of cycles per round	Throughput [Mbps]	
				One Block	Long Message
<i>Luffa</i> -224/256	30,834	1,124	9	15,980.0	31,960.0
<i>Luffa</i> -384	50,068	813	9	11,563.0	23,126.0
<i>Luffa</i> -512	65,102	690	9	9,808.5	19,617.0

Table 2: Area optimized implementations of the *Luffa* hash family.

Design	Area [GE]	Frequency [MHz]	# of cycles per round	Throughput [Mbps]	
				One Block	Long Message
<i>Luffa</i> -224/256	18,260	250	26	1,230.3	2,461.5
<i>Luffa</i> -384	27,130	250	34	941.2	1,882.4
<i>Luffa</i> -512	37,348	250	42	761.9	1,523.8

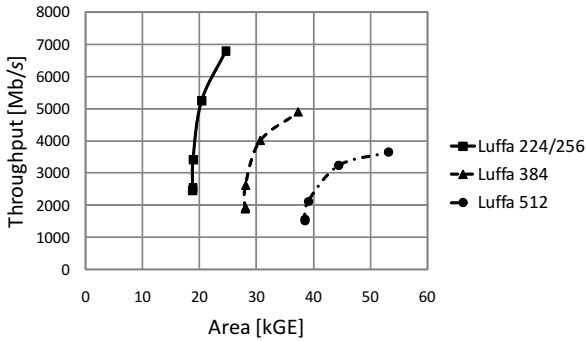


Figure 10: Compact implementations.

the maximum frequency to 250 MHz and synthesized our designs.

Figure 10 shows possible trade-offs for the compact designs. For example, a design of *Luffa*-224/256 containing only 19.8 kGE achieves a throughput of more than 5.2 Gbps.

To further explore the possibilities concerning the compact implementations, we have tried another approach where inside the permutation block we have used only two Sboxes for implementing the SubCrumb block (see Fig. 11). To perform the whole SubCrumb operation we have used eight 32-bit registers $a_0 \dots a_7$, regularly using the most significant bits of the registers to be the inputs of the Sboxes. The registers were then shifted to the left and the least significant bits were updated using the outputs of the Sboxes. Additionally, a single MixWord block was used for performing the MixWord operation (see Fig. 12). Again, to maintain the internal state, we have used w 256-bit registers.

However, due to the use of eight additional 32-bit registers, this approach resulted in almost the same circuit size, while the final throughput significantly decreased.

3.3 Pipelined Implementation

When hashing independent message blocks, one can benefit from using the pipelined architecture as illustrated in Fig. 13. Multiple non-linear permutation blocks need to be added ($8w$ blocks) as well as $8w$ pipelined 256-bit registers

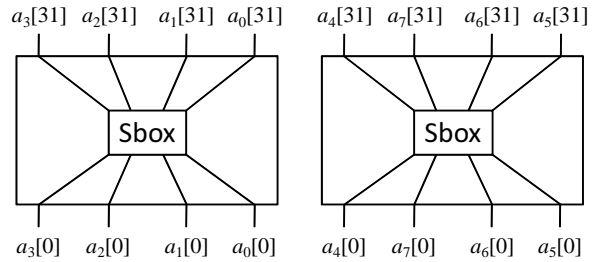


Figure 11: Compact SubCrumb block.

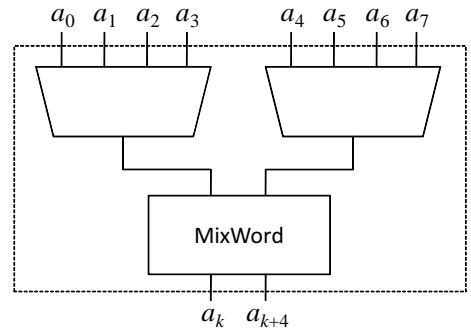


Figure 12: Compact MixWord block.

(one for each permutation block). This approach effectively increases the throughput more than 8 times at the cost of additional area overhead. As can be seen from Table 3, a throughput of 291.7 Gbps³ is achieved for the *Luffa*-224/256 version at the cost of 151.3 kGE.

3.4 Comparison With Previous Standards

To compare our implementations with the implementations of the previous standards (SHA-1 and SHA-2), we provide Table 4 with the state of art results indicating the ASIC

³This is a fully pipelined implementation and achieves the highest throughput in case of hashing 8 independent messages in parallel.

Table 3: Pipelined implementations of the *Luffa* hash family.

Design	Area [GE]	Frequency [MHz]	# of cycles per round	Throughput* [Mbps]	
				One Block	Long Message
<i>Luffa</i> -224/256	156, 613	508	9	57,799.0	115, 598.0
<i>Luffa</i> -384	217, 936	483	9	54,954.5	109, 909.0
<i>Luffa</i> -512	272, 413	478	9	54,385.5	108, 771.0

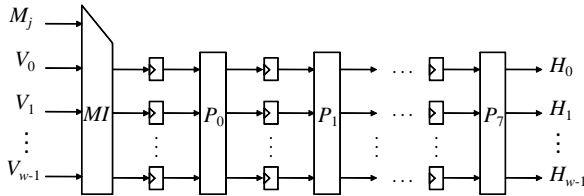
* Throughput for independent message blocks.

Table 4: Comparison results with the previous standards.

Design	Technology [μm]	Area [GE]	Throughput [Mbps]
SHA-1 [4] [†]	0.18	54, 133	3, 103
SHA-224/256 [5] [†]	0.13	22, 025	5, 975
SHA-224/256 [2] [†]	0.13	N/A	> 7, 420
SHA-384/512 [5] [†]	0.13	43, 330	9, 096
<i>Luffa</i> -224/256 [†]	0.13	30, 834	31, 960
<i>Luffa</i> -384 [†]	0.13	50, 068	23, 126
<i>Luffa</i> -512 [†]	0.13	65, 102	19, 617
SHA-1 [3] [‡]	0.25	6, 812	130
SHA-224/256 [7] [‡]	0.13	11, 484	1, 096
SHA-384/512 [7] [‡]	0.13	23, 146	1, 455
<i>Luffa</i> -224/256 [‡]	0.13	18, 260	2, 461
<i>Luffa</i> -384 [‡]	0.13	27, 130	1, 882
<i>Luffa</i> -512 [‡]	0.13	37, 348	1, 523

[†] High-Throughput designs.

[‡] Compact designs.

**Figure 13: Pipelined round function of the *Luffa* hash family.**

technology. Observing the results we can conclude that concerning a throughput, the *Luffa* hash family outperforms all the previous standards. Due to the larger internal state (w 256-bit registers), our compact implementations consume more area compared to the implementations of the previous standards.

4. CONCLUSION

The hardware implementations of the *Luffa* hash family have been evaluated in this paper. We conclude that the design is very well suited for both compact and high-throughput implementations. The most compact architecture of 18,260 GE was achieved for the 224/256-bit version of *Luffa*. The same version achieves a maximum throughput of 32 Gbps, while the pipelined design reaches a throughput of 291.7 Gbps. Due to an ample parallelism provided by the

Luffa hash family, it is possible to make plenty of trade-offs and choose the most appropriate design for a specific application. For example, a design of *Luffa*-224/256 achieving 16.7 Gbps consumes 24.6 kGE, while the design that consumes only 18.5 kGE achieves the throughput of more than 3.4 Gbps.

Regarding the hardware implementations, one can further explore the different levels of parallelism and make trade-offs by trading the throughput for the circuit size and vice versa. Especially challenging part remains the compact implementation of the hash functions in general and hence, we expect more research effort in that direction.

5. ACKNOWLEDGMENTS

The authors would like to thank Dai Watanabe and Christophe De Cannière for providing useful comments. Work supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State, by FWO project G.0300.07, by the European Commission under contract number ICT-2007-216676 ECRYPT NoE phase II, and by K.U. Leuven-BOF (OT/06/40).

6. REFERENCES

- [1] C. D. Cannière, H. Sato, and D. Watanabe. Hash Function *Luffa*. In *The First SHA-3 Candidate Conference*, 2009.
- [2] L. Dadda, M. Macchetti, and J. Owen. An ASIC design for a high speed implementation of the hash

- function SHA-256 (384, 512). In *ACM Great Lakes Symposium on VLSI (2004)*. ACM, 2004.
- [3] M. Kim and J. Ryou. Power Efficient Hardware Architecture of SHA-1 Algorithm for Trusted Mobile Computing. In *Information and Communications Security, Volume 4861/2008*. LNCS, 2008.
- [4] Y. Lee, H. Chan, and I. Verbauwhede. Throughput Optimized SHA-1 Architecture Using Unfolding Transformation. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2006)*. IEEE, 2006.
- [5] Y. Lee, H. Chan, and I. Verbauwhede. Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations. In *Information Security Applications, 8th International Workshop, WISA 2007, Lecture Notes in Computer Science 4867*. LNCS, 2007.
- [6] K. K. Parhi. *VLSI Digital Signal Processing Systems - Design and Implementation*.
- [7] A. Satoh and T. Inoue. ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS. volume 40, pages 3–10, Amsterdam, The Netherlands, The Netherlands, 2007. Elsevier Science Publishers B. V.
- [8] P. Schaumont and I. Verbauwhede. Domain-specific tools and methods for application in security processor design. In *Kluwer Journal for Design Automation of Embedded Systems*, 2002.
- [9] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/submissions_rnd2.html. National institute of standards and technology.
- [10] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology, CRYPTO 2005*, 2005.
- [11] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology, EUROCRYPT 2005*, 2005.