Evaluation of SHA-3 Candidates for 8-bit Embedded Processors

Stefan Heyse, Ingo von Maurich, Alexander Wild, Cornel Reuber, Johannes Rave, Thomas Poeppelmann, Christof Paar

Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
{firstname.lastname}@rub.de

Abstract. In 2007, NIST published a call for participation in a contest for a new standardized hash function, the future SHA-3. The call targets software oriented designs and suggests optimization for high-performance platforms. However, NIST explicitly encourages the evaluation of the candidates on 8-bit platforms. In this paper we present an implementation of several SHA-3 candidates, namely BLAKE, Blue Midnight Wish, Grøstl, and SHAvite-3 on an 8-bit microcontroller platform. While BMW shows the highest throughput, BLAKE seems to be the best balanced solution for 8-bit platforms.

1 Introduction

In 2007, NIST published a call for participation in a contest for a new standardized hash function, the future SHA-3 [13]. NIST required the submission of two implementations, one optimized for 32-bit architectures and one optimized for 64-bit architectures. The call even specifies a reference platform for both profiles, showing that performance on high-end microprocessors is of great concern for the final choice of the SHA-3.

Nevertheless, NIST also encourages the evaluation of the candidates on 8-bit platforms. Small embedded microprocessors are widely used in various applications, including smart cards, household appliances, medical devices, transportation systems and many more. Many of these applications require efficient cryptography. Consequently, we expect a wide interest in finding out whether SHA-3 candidates can be efficiently implemented on small 8-bit embedded microprocessors.

Compared to the SHA-3 reference platform, 8-bit microprocessors are heavily constrained in resources such as program memory and RAM. Besides throughput, efficiency has an additional meaning in this context: resources needed by an implementation of a hash function should be kept small, since embedded applications are very often cost constrained. In fact, in many situations memory consumption can be more crucial than throughput, in particular since many embedded applications only process small payloads.

In this paper we present an implementation of several SHA-3 candidates on an 8-bit microcontroller platform. Out of 64 submitted candidates, 14 candidates have been chosen to enter the second round of the SHA-3 competition. Out of these 14, we present implementation results for BLAKE, Blue Midnight Wish, Grøstl, and SHAvite-3, together with an implementation of Lane. Following a short description of the implementation platform in Section 2, we describe the

different implementations in Section 3. The paper concludes with a comparison of the results in Section 4.

2 Target Platform

Our implementations are designed for 8-bit AVR processors, a family of 8-bit RISC microcontrollers widely used in many embedded applications. The Atmel AVR processors operate at clock frequencies of up to 32 MHz, provide few kBytes of SRAM, up to hundreds of kBytes of Flash program memory, and additional EEPROM or mask ROM. The devices of the AVR family have 32 general purpose registers of 8-bit word size. A powerful example is the Atmel ATmega128 general purpose microcontroller [2]. The AVR core is also available as a smart card processor known as the AT90SCxxx family [1].

AVR microcontrollers can be programmed in AVR-assembler and in C. The presented implementations are designed to be executable on an AVR processor providing 1 KByte SRAM and a few KBytes of program memory. All implementations were tested and run on a developer smart card equipped with an ATMega163 microcontoller. All algorithms are completely implemented in AVR assembler.

3 Implementations

In this section we describe the implementations of the different algorithms. Not all ciphers are equally well suited for an implementation on an 8-bit platform. We tried to choose the ciphers which fit the 8-bit platform well. For each cipher details and specific features of the implementation are given together with a short description of that cipher.

3.1 BLAKE

BLAKE is a family of four hash functions that were proposed by Aumasson et al. [3] as a candidate for SHA-3. The iteration mode is HAIFA [6], its internal structure is the local wide-pipe [4] and its compression algorithm is a derivate of the stream cipher ChaCha [5]. BLAKE-32 operates on 32-bit words, has a block size of 512 bits, allows 128 bits of optional salt and produces a 256-bit hash value. The compression function takes as input a 8-byte chaining value h, a 16-byte message block m, a 4-byte salt s and a 2-byte counter t. Its output is a new chain value h'. The compression function is split into three different parts. At the beginning an initialization part generates a 16-word state out of the chain value, salt, counter and constants. The round function is then iterated ten times where in each round the function G_i is applied eight times on different words of the state. This function is the most time-critical part of BLAKE-32. The finalization generates a new chain value h' from the state, initial chain value and salt. BLAKE-32 uses three arithmetic operations: a 32-bit XOR, a 32-bit addition modulo 2^{32} and 32-bit rotations with different numbers of rotations. To perform these operations on a 8-bit platform, special care has to be taken. The 32-bit XOR can simply be reduced to four 8-bit XORs and the 32-bit addition can be broken down to four 8-bit additions with carry. Rotations are more complicated as there are four different rotations distances $(7, 8, 12, 16)$ used in BLAKE-32 and for each rotation new code has to be written. The designers of

BLAKE implemented the compression functionof BLAKE-32 on a PIC18F2525 8-bit microcontroller with a memory requirement of 2470 bytes of program memory and 274 bytes of data memory. Generating a message digest for sufficiently large messages requires 406 cycles/byte.

Our implementation of BLAKE-32 on an Atmel ATmega163 requires 1804 bytes of program memory, consisting of 160 bytes for the permutation table, 32 bytes for the initialization vectors, 64 bytes for the constants and 1548 bytes of code. We require 251 bytes of data memory split into 64 bytes for the state, 64 bytes for constants, 64 bytes for the current message block, 32 bytes for the chain value, 16 bytes for the salt, eight bytes for the counter and three bytes for temporary data. Our implementation occupies 11.0 % of the program memory and 24.5 % of the data memory of the ATMega163. With 323.6 cycles/byte for sufficiently large messages it outperforms the implementation of the designers by 20.3 % while reducing the required program memory by 27 % and the required data memory by 9 %. This improvement is mainly achieved due to an increase in performance of the function G_i that is called 80 times per round. In Table 1 the cycle count for each line of G_i of the original is compared with our implementation.

The code starts with an initialization function that resets the counter, loads the constants, the salt and the initialization vectors from the program memory to the data memory. Having the constants in the data memory saves four cycles each time a 32-bit constant is accessed. This is crucial as two constants are accessed during one iteration of G_i. The hash function checks if the current message block is the last message block and if this is the case it appends the correct padding. After the straightforward initialization of the state, the round function begins. The correct inputs for G_i are prepared and G_i is called. At the transition from G_3 to G_4 the input d, which corresponds to the state variable v_{15}, has already been input to G_3 therefore loading it again can be skipped. As stated earlier, G_i is the most time-consuming function and has the highest impact on the overall performance of BLAKE-32. We therefore tried to remove every unnecessary cycle in this function. The current implementation loads the entries of the permutation table σ from the program memory as this table would require another 160 byte of data memory. However, if more data memory usage is not a problem the permutation table could be loaded into the data memory during the initialization phase. This would save two cycles per G_i call, leading to a total of 160 cycles per block and a overall performance of 321.1 cycles/byte.

3.2 BMW

The first version of the Blue Midnight Wish (BMW) hash function and the tweaked version for the second round of the SHA3 competition have been developed by Gligoroski et. al. [11]. In this paper we only deal with the first version but the results can be used as a basis to evaluate the performance of the second version proposal. BMW supports the implemented 256-bit and a 224-bit variant which only differs in minor aspects as well as a 384/512-bit variant which is too complex for a constrained device like the ATMega AVR.

The general execution flow of BMW is to set up the initial double pipe H and then process incoming message blocks by applying the f_0, f_1 and f_2 function to iteratively generate new values for H and the double pipe Q. When a signal indicates that the last block is to be processed, the padding is applied to that last block. In case that the length of the last block is greater than 447 bits the

Table 1. Comparison between costs of the function G_i

Line of G_i	**Original** (PIC)	**Our code** (AVR)	**Improvement**
$a \leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$	76 cycles	53 cycles	30,3 %
$d \leftarrow (d \oplus a) \ggg 16$	24 cycles	7 cycles	70,8 %
$c \leftarrow c + d$	24 cycles	4 cycles	83,3 %
$b \leftarrow (b \oplus c) \ggg 12$	34 cycles	28 cycles	17,6 %
$a \leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$	67 cycles	41 cycles	38,8 %
$d \leftarrow (d \oplus a) \ggg 8$	22 cycles	9 cycles	59,1 %
$c \leftarrow c + d$	24 cycles	4 cycles	83,3 %
$b \leftarrow (b \oplus c) \ggg 7$	29 cycles	14 cycles	51,7 %
Preparing input, calling function	22 cycles	73 cycles	-231,8 %
Sum	**322 cycles**	**233 cycles**	**27,6 %**

padding is extended into a second block. After that, the hash output is present in the memory area of the double pipe H. Unlike other variants, BMW is not based on AES but uses shift, rotation, modulo addition and XOR operations on 32-bit words in the 224/256-bit variant. This is comfortable and fast on standard 32-bit CPUs but a challenge on an eight-bit AVR.

In our implementation we start by setting up the double pipe H with initial values. To save space in the program memory and due to their pattern, these values are created on the fly. This is also faster than loading them from the program memory. The first of the following iteratively applied f_j functions is f_0 which carries out the bijective transform of $W = A(M \oplus H)$ in the first part. A simple optimization is to precompute $X_i = M_i \oplus H_i$ as these values are needed multiple times. Moreover, every line can be generalized as $W_i = X_a \circ X_b \circ X_c \circ X_d \circ X_e$ which makes it possible to design an assembler routine that is given the indexes of X and the designated operation (addition or subtraction). This routine can be called for every of the sixteen lines and therefore saves a huge amount of program memory. In our implementation this is done by using the LSB r_0 of every register to encode the operation and $r_{7..1}$ to store an offset allowing efficient pointer arithmetic. In some lines (e.g. W_5) the used values of X_i are not entirely ordered which makes it necessary to subtract a given correction offset which is applied after X_a. The resulting W_i needs no extra space in the RAM as it is stored in the second half of the quadruple pipe Q.

In the next part of the f_0 function the previously computed values W_i are now the input of the s_0, s_1, s_2, s_3 and s_4 functions resulting in the first half of the quadruple pipe Q. They require mostly shifting and XORing of the input which is expensive on the AVR because each register can only be shifted one position left or right in a single cycle. Furthermore, all operations take place on 32-bit words requiring a chained shift of four registers in order to perform one SHL^1 or SHR^1 function. Therefore, an increase in execution speed can be gained by tuning every shift and rotation sequence by hand and considering the fact that shifts or rotations by a multitude of eight costs almost no cycles on the AVR, as they can be performed by simply moving the registers. Next the f_1 function, which is designed as a weak block cipher takes the message block M and the first part of the quadruple pipe Q as input and outputs the second part of the quadruple pipe Q. This is achieved by multiple calls of the

$expand_1(j)$ and $expand_2(j)$ functions. The $expand_1(j)$ function reuses the $s_{0..4}$ functions of f_0. To make them applicable in both parts, a small modification in the implementation indicates whether the result should be stored in memory (f_0) or added to some registers for further use (f_1). Moreover, the constant K_j is precomputed and added in every iteration. After that the f_2 function takes the message block M and the quadruple pipe Q as input and outputs the new double pipe H, which also stores the hash after the final round. As the cumulative temporary variables XL and XH are used in every line of f_2 they are not placed in memory but held in the first eight registers. For code size reduction a generic function works on every line, after the non-generic part has been done. In H_0 to H_7 the only non-generic part is the number of shifts of XH and Q. For H_8 to H_{15} the rotation of H and the shift of XL has to be performed prior to the application of a generic function.

3.3 Grøstl

The hash function Grøstl [10] is one of the candidates that advanced to the second round of the NIST SHA-3 competition. It is based on an iterated structure using a single compression function. This function is build around the permutations P and Q, which utilize the same ideas as the AES round function. Due to the bigger state of Grøstl, 512–bit for Grøstl-244/256 and 1024–bit for Grøstl-384/512, the round transformations were slightly changed. The most noticeable change compared to the AES [8], despite the bigger state, is the lack of a key, which was left out to prevent weak key attacks. P and Q are used to construct the compression function, which has the form $f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h$. In this case h is either the previous block processed by f or an initialization vector, whereas m denotes the message block to be hashed. Both permutations are build from the same round structure, which is applied ten or 14 times depending on internal state's size. During each round the functions AddRoundConstant, SubBytes, ShiftBytes and MixBytes are applied to the internal data. The behavior of the first function marks the only difference between the permutations P and Q. According to the running permutation, a derived round counter is added to a distinct byte in the state. SubBytes uses the AES S–Box to perform a non–linear transformation by substituting every value in the state with the corresponding value from the S–Box. The ShiftBytes function shifts the bytes within a row cyclically, depending on the row number. Following this is the last step in the permutation function called MixBytes. This function can be described as a matrix multiplication in $\mathbb{F}_{256}$, which is defined with the same irreducible polynomial as in the AES. In contrast to the AES algorithm another circular matrix B with higher values is used. After all blocks have been processed an output transformation is used to cut down the internal state to the output size. In this step the permutation P is used again.

The Grøstl algorithm uses a byte–oriented structure to represent the internal state, which is a perfect fit for the 8–bit AVR microcontroller. Due to the resource constraints on the chosen controller only the 512–bit state size version was implemented. To save cycles, the functions AddRoundConstant, SubBytes and ShiftBytes were merged into one function. This allows to perform the necessary operations without repeated RAM access. As storing and loading each byte costs four cycles, 512 cycles are saved each round. The next step would be to include MixBytes into this combined function to save another 256 cycles. Unfortunately, this is only possible by increasing the amount of needed RAM,

as the MixBytes operation can not be performed in–place. AddRoundConstant, SubBytes and ShiftBytes operate only on the byte level, meaning that the values in the state do not affect each other. MixBytes however combines different values of the state to new values by a matrix multiplication. In–place calculation would replace values needed for later calculations. Therefore, additional RAM for out–of–place calculation of this function is needed. This is not the only difficult point in implementing the Grøstl MixBytes function. Due to the size and high values in the multiplication matrix B, this step is the most expensive to implement. The drawback can be minimized by using a 256–byte look–up table for the multiplication with two. Using look–up tables for other multiplications does not result in a performance increase, so that the two–times table can be used again, minimizing the needed program memory. Another way, optimizing MixBytes, is to reuse intermediate results, which is easily possible by calculating whole columns at a time. The performance could be potentially increased by copying the look–up tables into the RAM before calling the function. Given the specification of the used device this was however not feasable.

3.4 Lane

Lane is an iterated cryptographic hash function which reuses components from the AES block cipher [12]. We implemented the Lane-256 function which uses a digest length of 256-bit, a blocksize of 512-bit and a chaining value of 256-bit. The Lane compression function consists of a message expansion part in which the 512-bit message and the 256-bit chaining value get expanded and separated into six 256 Bit states w_0 to w_6. Each state is transformed by the permutation P and the results are XORed together in groups of three. The resulting two states are then used as input to the permutation Q. The XORed outputs of the two Q permutations form the output of the Lane compression function. Both permutations consist of repeated calls of the round function which is based on the components known from the AES block cipher.

The architecture of the AVR microcontroller only provides 32 8-bit registers. Six of these are already used for addresspointers while another eight are used for temporary values and the computation of the constants. This allows us to store only one half of the state in the registers. As we will see later on in each round inside the permutations only the last step (SwapColumns) requires both 128-bit states as input. Therefore we can perform the permutations on the 128-bit state in the registers but we need to reload the second part of the state every round.

The round functions inside the permutations are specified a repetitions of transformations which are mostly known from the AES block cipher. The SubBytes transformation is identical to the corresponding function of the AES but operates on a larger state. For our 8-Bit implementation this transformation was implemented as a lookup table which is stored in the program memory. Since the ShiftRows operation is just a permutation of the 128-bit AES state this function could be included in the SubBytes operation with a cost of only four cycles. The next transformation is the MixColumns operation. As described in [12] the

transformation for each columns of the 128 Bit state can be computed as follows:

$$\begin{aligned} y'_0 &= y_2 \oplus y_3 \oplus 2y_0 \oplus 3y_1 \\ y'_1 &= y_0 \oplus y_3 \oplus 2y_1 \oplus 3y_2 \\ y'_2 &= y_0 \oplus y_1 \oplus 2y_2 \oplus 3y_3 \\ y'_3 &= y_1 \oplus y_2 \oplus 2y_3 \oplus 3y_0 \end{aligned} \tag{1}$$

Since $2y_i$ may require a reduction step while $3y_i$ is only the XOR of $2y_i$ and y_i we compute all the $2y_i$ values for a column in a first step. With these values we build two temporary values $y_{t0} = 2y_1 \oplus y_2 \oplus y_3$ and $y_{t1} = y_0 \oplus y_1 \oplus 2y_3$. The new column can then be computed as:

$$\begin{aligned} y'_0 &= y_{t0} \oplus y_1 \oplus 2y_0 \\ y'_1 &= y_{t0} \oplus y_0 \oplus 2y_2 \\ y'_2 &= y_{t1} \oplus y_3 \oplus 2y_2 \\ y'_3 &= y_{t1} \oplus y_2 \oplus 2y_0 \end{aligned} \tag{2}$$

This way for each byte only three XOR operations are needed compared to four using the naive implementation. The speed of the MixColumns operation highly influences the overall speed of the compression function since it is used in all rounds in both permutations.

The AddConstants operation XORs a constant to each column of the state. Due to the limited memory of the microcontroller the constants can not be stored and therefore need to be calculated during runtime using the LFSR specified in the Lane algorithm. Depending on the round counter r one half of a 64-bit counter holding the number of message bits hashed so far is added to the fourth column of the first AES state. Because only 4-byte of the counter are needed for the addition while 8-byte need to be stored, the counter is written in the RAM rather than in the registers. The SwapColumns operation exchanges the third and fourth columns of the first 128-bit state with the first and second column of the second 128-bit state. Due to this operation we can not operate only on one half of the 256-bit state over the whole permutation. Therefore, each round the first part of the state in the registers needs to be exchanged with the second part of the state in the RAM. To reduce the RAM access the SwapColumns operation can be performed while loading the state for the next round. This increases the performance of the implementation but requires additional 256 Bits of RAM.

3.5 SHAvite-3

SHAvite-3 is composed of a compression function and an iteration function. To transform a block cipher into a compression function SHAvite-3 uses the Davies-Meyer transformation. The used block cipher build upon a Feistel structure with an AES round as building block. As iteration function SHAvite-3 uses **Ha**sh **I**terative **Fra**mework (HAIFA) which is needed to handle arbitrary message length [6][14].

SHAvite-3 uses two different compression functions. One to build hashes less equal than 256 bit and one to build hashes larger than 256 bit. Scope in this paper is only the function for a bit length up to 256 bit. The state of the compression function is initialized with a fixed value and divided into two parts, R_0 and L_0. R_0 is XORed with a subkey and put into three iterative AES round functions

with two additional subkeys. The last AES round has an all zero subkey. After this, R_0 is XORed with L_0 resulting in R_1 and R_0 will be L_1. This construction will be iterated twelve times. The message to be hashed is only used to calculate the subkeys. Altogether 36 128-bit subkeys for twelve iterations are needed. The first four subkeys consist of the message itself. For further subkeys the message is put into an AES round with a salt as key and the result is XORed with a counter and old subkeys.

Besides of a few XORs and iterations the whole complexity stems from the AES round function. A complete implementation of AES is not applicable here, because instead of 16 sequential AES rounds only three rounds with a changed key derivation is used. Therefore, the implementation on an AVR XMEGA with AES module will would not improve the implementation. The AES round function is called 52 times per block. To speed up the implementation a fast AES round function is essential. One implementation trick is to combine SubBytes and ShiftRows. This means the output of the S-Box only has to be saved in other registers. Another trick is to store the S-Box at specific positions in program memory. The AES S-Box has a size of 256 byte and the AVR program memory is addressed byte-wise. This means if the S-Box is stored at an address with the lower address byte being zero, the upper address byte is always the same for each S-Box entry. This saves a few cycles during the address calculation. Our implementation calculates eight subkeys while a call. To speed up the implementation all 36 subkeys could be calculated at once, but this would require 576 byte in RAM just for the subkeys. The decision for the alternative with the eight subkey calculations was just a time, size tradeoff.

In the original SHAvite-3 paper an 8-bit implementation was estimated with 20370 cycles per block. Our implementation requires about 21100 cycles and is therefore as fast as expected. The additional cycles could be a result from the not considered storing complexity. The state has a bit length of 256 bit and fits exactly into the 32 AVR registers, but then no registers are remaining for temporary results or pointers. Because of the Feistel structure, only half of the state has to be handled at once. The other half can be pushed on the stack for later use [7].

4 Results and Conclusion

We now present the achieved results for our implementations. In Table 2, the code size includes all precomputed tables. Also the RAM value includes the necessary space for the 64 byte message block, which is hashed. The values in the short message column are measured for a message which is only in the size of a single block. These values include the time for initialization, padding and a single round of the corresponding hash function. Finally, the last column is an asymptotic value for very large messages, where the time for initialization and padding is negligible. The first and last block of a message are not included in this value, because they need a special handling.

Although performance is always of high interest for the choice of an algorithm, in embedded systems the code size and RAM consumption often play a significant role a well. The less resources are used, the smaller and consequently cheaper the used microcontroller can be [9]. By performance, BMW is outperforming the other implementations, followed by BLAKE and SHAvite-3. Grøstl is the only candidate that shows a significant difference in performance for the

Table 2. Size and Performance Results

Hashfunction	code size	RAM	Short message	Long message
BLAKE-32	1804 bytes	251 bytes	332 cycles/byte	324 cycles/byte
BMW-256	3558 bytes	320 bytes	247 cycles/byte	227 cycles/byte
Grøstl-256	4852 bytes	261 bytes	1044 cycles/byte	687 cycles/byte
Lane	2088 bytes	104 bytes	600 cycles/byte	588 cycles/byte
SHAvite-3	5254 bytes	232 bytes	338 cycles/byte	331 cycles/byte

hashing of long messages, but its overall performance is not very good. In terms of code size, BLAKE and to a certain extend also Lane do quite well with about 2 kBytes of code. The other implementations are significantly larger. The RAM requirement of Lane is also outstandingly low.

BLAKE shows the best overall performance, featuring small resource consumption and a decent throughput. BMW gives a slightly better throughput at the cost of roughly doubling the code size. Grøstl and to a lesser extend SHAvite-3 are less interesting due to the large code size and, in the case of Grøstl, the considerably lower throughput.

References

1. Atmel Corp. Overview of Secure AVR Microcontrollers 8-/16-bit RISC CPU, 2007. `http://www.atmel.com/products/SecureAVR/`.
2. Atmel Corp. Specifications of the ATmega128 Microcontroller, 2007. `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`.
3. J. Aumasson, L. Henzen, W. Meier, and R. Phan. SHA-3 proposal BLAKE. *Submission to NIST*, 2008.
4. J. Aumasson, W. Meier, and R. Phan. The hash function family LAKE. In *Fast Software Encryption*, pages 36–53. Springer, 2008.
5. D. Bernstein. ChaCha, a variant of Salsa20. *See http://cr.yp.to/chacha.html*.
6. E. Biham and O. Dunkelman. A framework for iterative hash functions HAIFA. In *Second NIST Cryptographic Hash Workshop*. Citeseer, 2006.
7. E. Biham and O. Dunkelman. The shavite-3 hash function. Submission to NIST (Round 02), 2009.
8. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.
9. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A Survey of Lightweight Cryptography Implementations, November/December 2007.
10. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008.
11. D. Gligoroski, V. Klima, S. J. Knapskog, M. El-Hadedy, J. Amundsen, and S. F. Mjolsnes. Cryptographic hash function blue midnight wish. Submission to NIST (Round 2), 2009.
12. S. Indesteege. The lane hash function. Submission to NIST, 2008.
13. NIST. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA3) Family. Federal Register / Vol. 72, No. 212 / Notices, November 2007.
14. R. S. Winternitz. A secure one-way hash function built from des. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.