Efficient Implementation of the SHA-512 Hash Function for 8-bit AVR Microcontrollers

Hao Cheng[1], Daniel Dinu[2], and Johann Großschädl[1]

[1] CSC and SnT, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
hao.cheng.001@student.uni.lu
johann.groszschaedl@uni.lu
[2] Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061, USA
ddinu@vt.edu

Abstract. SHA-512 is a member of the SHA-2 family of cryptographic hash algorithms that is based on a Davies-Mayer compression function operating on eight 64-bit words to produce a 512-bit digest. It provides strong resistance to collision and preimage attacks, and is assumed to remain secure in the dawning era of quantum computers. However, the compression function of SHA-512 is challenging to implement on small 8 and 16-bit microcontrollers because of their limited register space and the fact that 64-bit rotations are generally slow on such devices. In this paper, we present the first highly-optimized Assembler implementation of SHA-512 for the ATmega family of 8-bit AVR microcontrollers. We introduce a special optimization technique for the compression function based on a duplication of the eight working variables so that they can be more efficiently loaded from RAM via the indirect addressing mode with displacement (using the `ldd` and `std` instruction). In this way, we were able to achieve high performance without unrolling the main loop of the compression function, thereby keeping the code size small. When executed on an 8-bit AVR ATmega128 microcontroller, the compression function takes slightly less than 60k clock cycles, which corresponds to a compression rate of roughly 467 cycles per byte. The binary code size of the full SHA-512 implementation providing a standard Init-Update-Final (IUF) interface amounts to approximately 3.5 kB.

Keywords: Internet of Things (IoT), Lightweight Cryptography, AVR Microcontroller, Software Optimization, Performance Evaluation

1 Introduction

A cryptographic hash function takes data of arbitrary form and size as input to produce a fixed-length output of typically between 160 and 512 bits that can be seen as a "digital fingerprint" of the data [15]. Hash functions play a major role in IT security and serve a wide variety of purposes, ranging from verifying the integrity of data (e.g. messages, documents, software) over securing the storage

of sensitive credentials (e.g. passwords, credit card numbers) to realizing proof-of-work systems for digital currencies. In addition, hash functions are essential building blocks of digital signature schemes, key derivation functions, message authentication codes, and pseudo-random number generators. The design and analysis of cryptographic hash functions has been an active area of research in the past 30 years that yielded not only a large body of new algorithms but also many insights on the security of existing ones. Among the most important and commonly-used hash functions are the members of the SHA-2 family, which are approved by the NIST [17] and many other standardization bodies around the world. The SHA-2 family consists of six hash functions providing varying levels of security with digests ranging from 224 to 512 bits. However, only two of the six members, namely SHA-256 and SHA-512, can be seen as "original" designs since the other four are just variants of them with different initial hash values and truncated digests. SHA-256 and SHA-512 share many properties; they are both based on the Merkle-Damgård structure with a Davies-Meyer compression function that is solely composed of Boolean operations (bitwise AND, bitwise XOR), modular additions, as well as rotations (resp. shifts). These operations are performed on eight working variables, each of which has a length of 32 bits in SHA-256 and 64 bits in SHA-512.

The enormous expansion of the Internet of Things (IoT) in recent years has initiated a strong interest in *lightweight cryptography*, a relatively new subfield of cryptography dealing with (i) the design of novel cryptographic algorithms tailored to extremely constrained environments, and (ii) the efficient and secure implementation of cryptographic algorithms with the objective to minimize the execution time and resource requirements (e.g. power consumption and silicon area in the case of hardware implementation, and RAM footprint and code size when implemented in software) [1]. Optimization techniques to make SHA-256 suitable for resource-constrained IoT devices have been actively researched in the past 10 years and numerous lightweight implementations in both hardware [8] and software [4, 19] were reported in the literature. Balasch et al. describe in [4] an optimized Assembler implementation of SHA-256 for the 8-bit AVR platform that achieves a hash rate of 532 cycles per byte when hashing a 500-byte message on an ATtiny45 microcontroller. The to-date fastest SHA-256 software for an 8-bit processor was introduced by Osvik in [19] and reaches an execution time of 21440 cycles for the compression function alone, which corresponds to a compression rate of 335 cycles per byte. These results show that SHA-256 is difficult to implement efficiently on small 8-bit microcontrollers, mainly due to the severely limited register space of these platforms, which is not sufficient to accommodate the working variables *and* provide storage for temporary values or pointers to access RAM. Furthermore, rotations are generally slow on these processors since they do not feature dedicated rotation hardware, which means rotating an 8-bit register by n bits takes (at least) n clock cycles.

SHA-512 is even more challenging to implement on small microcontrollers than SHA-256 since each of the eight working variables has a length of 64 bits instead of 32 bits, which doubles the register pressure. To our knowledge, it has

never been attempted before to optimize SHA-512 in Assembly language for an 8-bit microcontroller since the scientific literature does not provide any results and also public repositories like GitHub do not contain any source codes. This is quite surprising since SHA-512 is one of the most important hash functions and especially popular on 64-bit platforms, where it significantly outperforms SHA-256 [12]. It is expected that the (relative) importance of SHA-512 versus SHA-256 will increase in the future, not only in the realm of high-performance computing but also in the IoT, mainly for two reasons. The first reason is the emergence of *EdDSA*, a state-of-the-art signature scheme using elliptic curves in twisted Edwards form [6]. EdDSA is most commonly instantiated with a curve that is birationally equivalent to Curve25519, providing a security level of approximately 128 bits, and SHA-512 as hash function, which is assumed to have 256 bits of security against collision attacks [14]. While it is common practice to choose elliptic curves and hash functions of roughly equivalent security, the designers of EdDSA decided to deviate from this practice and utilize a double-size hash function in order to "help alleviate concerns regarding hash function security" [6]. EdDSA comes with a number of features that make it attractive for resource-limited IoT devices, most notably the outstanding efficiency of the underlying elliptic-curve arithmetic. Speed-optimized EdDSA implementations for the 8-bit AVR platform are described in [13, 16], but these implementations are mainly tuned towards fast arithmetic (i.e. fast scalar multiplication) and do not contain any Assembler optimizations for SHA-512.

The second reason why SHA-512 can be expected to increase in importance compared to SHA-256 is the emerging threat of *quantum cryptanalysis*. Some 20 years ago, Grover [11] introduced a quantum algorithm for finding a specific entry in an unsorted database of size n with complexity $\mathcal{O}(2^{n/2})$, providing (in theory) a quadratic speedup compared to classical exhaustive search. Grover's algorithm is believed to have an effect on cryptanalysis; for example, it could be applied to find the secret key used by an n-bit block cipher or a preimage of an n-bit hash value in $\sqrt{n}$ steps. Brassard et al. [7] proposed a quantum algorithm for generic collision search based on Grover's technique that needs to perform only $\mathcal{O}(2^{n/3})$ quantum evaluations of a hash function, but requires a very large amount of memory. Recently, Chailloux et al. [9] put forward a novel quantum algorithm for finding collisions with a quantum-query complexity and also time complexity of $\mathcal{O}(2^{2n/5})$, a quantum-memory complexity of just $\mathcal{O}(n)$ qbits, and a classical memory complexity of $\mathcal{O}(2^{n/5})$ bits. However, Bernstein [5] disputes the complexity analysis given in [9] and argues that all of the currently-known quantum algorithms for collision search are, in fact, less cost-effective than the best classical (i.e. pre-quantum) technique when execution time and hardware cost are considered appropriately. Nonetheless, the NIST published in [18] an assessment of the impact of quantum computers on present-day cryptographic algorithms according to which hash functions may need a larger output. In the the context of the SHA-2 family, this obviously suggests to use a hash function with a digest length of more than 256 bits, i.e. SHA-384 or SHA-512, when 128 bits of security against collision attacks are needed. Since research in quantum

cryptanalysis is still in its infancy, it seems prudent to err on the safe side and deploy SHA-512 in applications that require long-term security[3].

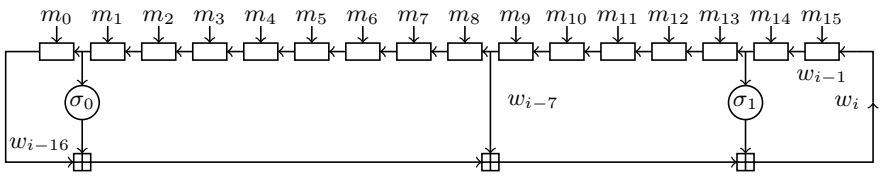
In this paper, we introduce a carefully-optimized Assembler implementation of the SHA-512 hash function for 8-bit AVR microcontrollers, in particular the ATmega series. As mentioned before, we are not aware of previous publications concerned with implementation and optimization aspects of SHA-512 on small 8-bit platforms. Therefore, it is not known how to optimize SHA-512 for these resource-constrained microcontrollers and what execution time can be reached with hand-written Assembler code. The present paper intends to answer these questions and describes two software optimization techniques that allow one to achieve significant performance gains compared to standard C implementations compiled with avr-gcc. Our first optimization technique aims to accelerate the four sigma operations, especially the rotations (resp. shifts) of 64-bit operands they perform. The other optimization speeds up the memory accesses that have to be carried out during the computation of the SHA-512 compression function by exploiting the indirect addressing mode with displacement (offset), which is supported by the AVR architecture. Combining both optimization techniques enabled our Assembler implementation of the compression function to outperform C code compiled with optimization level `-o2` by a factor of 4.42.

2 Overview of SHA-512

SHA-512 is a member of the NIST-standardized SHA-2 family of cryptographic hash functions that produces a 512-bit digest and, therefore, provides 256 bits of security against collisions [17]. The input message can have a length of up to $2^{128} - 1$ bits and is processed in blocks of 1024 bits. Like other members of the SHA-2 family, SHA-512 is based on the well-known Merkle-Damgård structure with a Davies-Meyer compression function that uses solely Boolean operations (i.e. bitwise AND, XOR, OR, and NOT), modular additions, as well as shifts and rotations. All operations are applied to 64-bit words.

SHA-512 consists of two stages: preprocessing and hash computation. In the former stage, the eight *working variables*, denoted as a, b, c, d, e, f, g, and h in [17], are initialized to certain fixed constants. Furthermore, the input message is padded and then divided into 1024-bit blocks. The actual hash computation passes each message block (represented by 16 words m_0, m_1, ... m_{15} of 64 bits each) through a *message schedule* (illustrated in Fig. 1) to expand them to 80 words w_i with $0 \leq i \leq 79$. Then, the eight working variables are updated using a *compression function* that consists of 80 rounds. A round of the compression function is exemplarily depicted in Fig. 2. The processing of a 1024-bit message block results in eight 64-bit intermediate hash values. After the whole message has been processed, the 512-bit digest is generated by simply concatenating the eight intermediate hash values.

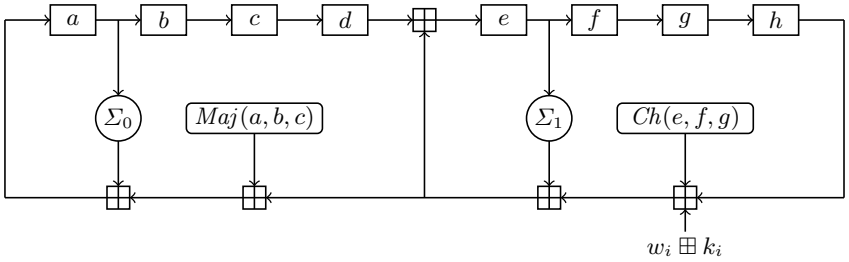
[3] A common example of a class of IoT devices with long-term security requirements are smart meters because they are expected to reach lifetimes of between 10 and 25 years (according to [22, Table 2]) without requiring regular maintenance.



$$\sigma_{0,i} = (w_{i-15} \ggg 1) \oplus (w_{i-15} \ggg 8) \oplus (w_{i-15} \gg 7)$$
$$\sigma_{1,i} = (w_{i-2} \ggg 19) \oplus (w_{i-2} \ggg 61) \oplus (w_{i-2} \gg 6)$$
$$w_i = w_{i-16} \boxplus \sigma_{0,i} \boxplus w_{i-7} \boxplus \sigma_{1,i}$$

Fig. 1. The message schedule of SHA-512.

Each round of the compression function consists of seven modular additions of 64-bit words and four relatively more costly operations: *Maj* (Majority) and *Ch* (Choice) are composed of a sequence of bitwise operations (mainly logical AND and XOR), whereas the two sigma operations Σ_0, Σ_1 perform XORs and rotations of a 64-bit word by a fixed amount of bit-positions. Furthermore, in each round i of the compression function, the set of eight working variables is rotated word-wise. The word k_i in Fig. 2 is one of eighty 64-bit constants.



$$\Sigma_{0,i} = (a_i \ggg 28) \oplus (a_i \ggg 34) \oplus (a_i \ggg 39)$$
$$Maj_i = (a_i \wedge b_i) \oplus (a_i \wedge c_i) \oplus (b_i \wedge c_i)$$
$$t_{2,i} = \Sigma_{0,i} \boxplus Maj_i$$
$$\Sigma_{1,i} = (e_i \ggg 14) \oplus (e_i \ggg 18) \oplus (e_i \ggg 41)$$
$$Ch_i = (e_i \wedge f_i) \oplus (\bar{e}_i \wedge g_i)$$
$$t_{1,i} = h_i \boxplus \Sigma_{1,i} \boxplus Ch_i \boxplus k_i \boxplus w_i$$
$$(h_{i+1}, g_{i+1}, f_{i+1}, e_{i+1}) = (g_i, f_i, e_i, d_i \boxplus t_{1,i})$$
$$(d_{i+1}, c_{i+1}, b_{i+1}, a_{i+1}) = (c_i, b_i, a_i, t_{1,i} \boxplus t_{2,i})$$

Fig. 2. A round of the SHA-512 compression function.

As pointed out in [17, Sect. 6.1.3], the message schedule can either be pre-computed (so that the expanded message block is available before starting the

first round of the compression function) or executed "on the fly" in an iterative fashion, e.g. 16 words at a time. The latter method is more RAM-friendly since there is no need to store all 80 words w_i of the expanded message block, while the former approach is usually a bit faster. A more detailed description of the SHA-512 hash function can be found in the specification [17].

3 Evaluation of Existing C/C++ Implementations

As indicated in Sect. 1, we are not aware of any previous Assembler implementation of SHA-512 for 8-bit AVR microcontrollers. However, there exist various lightweight C/C++ implementations that can be compiled to run on AVR.

3.1 8-bit AVR Architecture

The popular AVR architecture was originally developed by Atmel Corporation (now part of Microchip Technology, Inc.) on basis of the RISC philosophy and a modified Harvard memory model. Its latest revision supports 129 instructions altogether, the vast majority of which have a fixed length of two bytes [2]. The register file is relatively large and contains 32 general-purpose working registers (named `R0` to `R31`) of 8-bit width that are directly connected to the Arithmetic Logic Unit (ALU). Standard arithmetic/logical instructions have a two-address format, which means they can read two 8-bit operands independently from two of the working registers and write the result back to one of them. Since AVR is a "Harvard-based" architecture, it uses separate memories, buses, and address spaces for program and data to maximize performance and parallelism. Three pairs of working registers can operate as 16-bit pointers (`X`, `Y`, and `Z`) to access data memory, whereby five addressing modes are supported. Furthermore, the pointer `Z` can be used to read from (and write to) program memory.

The specific AVR microcontroller on which we simulated the execution time of our SHA-512 software is the ATmega128 [3]. It features a two-stage pipeline capable to execute an instruction while the next instruction is fetched from the program memory. Conventional ALU instructions take one clock cycle, whereas load/store instructions from/to data memory have a latency of two cycles. The memory sub-system includes 128 kB flash memory and 4 kB SRAM.

3.2 Performance Analysis

At the beginning of our research effort towards high-speed SHA-512 hashing on 8-bit AVR micrcontrollers was a careful analysis of some existing open-source C and C++ implementations, including the SHA-512 software contained in the Arduino Cryptography Library [20]. We compiled the source codes with version 5.4.0 of avr-gcc (optimization option `-o2`) and determined the execution times through simulations using the ATmega128 as target platform [3]. The SHA-512 implementation from the Arduino Cryptography Library needs 1341398 cycles to compute the digest of a 500-byte message, which corresponds to a hash rate

of 2683 cycles per byte. We also evaluated a couple of other C implementations that were available on source-code repositories like GitHub and found them to be very similar in terms of performance. In addition, we searched the literature for lightweight SHA-512 software, but did not discover any recent (i.e. less than five years old) publications with implementation results for AVR. However, we came across a paper by Wenzel-Brenner et al. [21] from 2012 that analyzes the performance of several hash algorithms, including a C implementation of SHA-512, on an ATmega1284P. They reported a hash rate of more than 8000 cycles per byte when hashing a long message, but this result should taken with some caution because it was obtained with a now-outdated version of avr-gcc that is not state-of-the-art anymore. For comparison, the to-date best implementation of SHA-256 was written in AVR Assembly language and reaches a compression rate of 335 cycles per byte [19]. These results immediately prompt the question of why the compiler-generated code for SHA-512 is approximately eight times less efficient than the hand-optimized Assembler code for SHA-256. In order to answer this question, we inspected the Assembler output generated by avr-gcc and found two reasons for the rather poor performance of SHA-512. First, the four sigma operations are much slower than they could be because avr-gcc uses the "generic" rotation functions for 64-bit words from libgcc. Second, the code generated by avr-gcc contains an unnecessarily large number of load and store instructions due to register spills, which could be massively reduced by means of a sophisticated register allocation strategy. What makes things even worse is that many of these load/store operations involve costly address arithmetic.

The four sigma operations have a major impact on the overall performance of the SHA-512 compression function. In essence, they comprise a few rotations of 64-bit words by a fixed number of bit-positions and logical XOR operations [17]. Rotations are generally slow on 8-bit AVR microcontrollers since, unlike to ARM processors, they do not feature a dedicated functional unit that could execute rotations by several bit-positions in a single clock cycle. Therefore, all multi-bit rotations need to be composed of several rotations by one bit. Whenever avr-gcc discovers a rotation of a 64-bit word in C code, it implements this rotation using functions named `rotldi3` (to rotate left) or `rotrdi3` (to rotate right), which are part of the avr-gcc low-level runtime library libgcc. Both are generic rotation functions that consist of two loops; the first loop performs one or more bytewise rotations (by simply copying the content of registers via the `mov` instruction [2]) and the second loop a sequence of bitwise rotations. When rotating by n bits, then the first loop is iterated (at most) $\lfloor n/8 \rfloor$ times and the second loop (n mod 8) times. Unfortunately, it seems avr-gcc is not capable to "globally" optimize the sigma operations since it just calls `rotldi3`/`rotrdi3` to perform the rotations and then executes `eor` instructions without taking into account that the bytewise rotations and XORs can be merged. We describe in Subsect. 4.1 hand-written Assembler implementations of the sigma operations that reduce the execution time significantly compared to C code.

Another reason for the relatively poor performance of C implementations is that each of the eight working variables has a length if 64 bits, which amounts

to 64 bytes altogether. However, the register file of an AVR microcontroller can accommodate only 32 bytes, i.e. just half of the size of all working variables. In addition, it has to be taken into account that not every register can be used to store working variables since some registers are needed for temporary variables or to hold the 16-bit pointers. Thus, it is only possible to keep (at most) three 64-bit words in the register file at any time, which implies the majority of the working variables has to be kept in RAM rather than in registers. Our analysis of the Assembler code generated by avr-gcc showed that the register allocation strategy is far from ideal, which causes a massive number of memory accesses (i.e. loads and stores) that could be avoided. Most of these loads/stores require further instructions for address (i.e. pointer) arithmetic. Particularly costly in terms of memory accesses is the word-wise cyclic rotation of the set of working variables, which, as explained in Sect. 2, has to be carried out in each iteration of the compression function. However, it is possible to completely avoid these word-wise rotations by (partially) unrolling the main loop and replicating the loop body eight times. This unrolling enables one to accomplish the word-wise rotation in an implicit (i.e. "hard-coded") way by simply adapting the order in which the eight working variables are accessed. Unfortunately, the performance gained by this technique has to be paid with a significant increase of code size due to the loop unrolling. We present in Subsect. 4.2 a new approach to reduce the cost of the word-wise rotations that increases the code size only slightly.

4 Our Assembler Implementation

In the following, we present the Assembler optimizations we developed to speed up the sigma operations and memory accesses of the compression function.

4.1 Optimization of the Sigma Operations

Our decision to develop Assembler optimizations for the four sigma operations σ_0, σ_1, Σ_0, and Σ_1, which are used in the SHA-512 compression function, was motivated by the high potential for improvement we identified by analyzing the Assembler code generated by avr-gcc and comparing it with the best-optimized implementations of rotations on 8-bit AVR from [10]. In short, our strategy to speed up the sigma operations was to minimize the overall number of bitwise rotations and to "merge" the bytewise rotations with XORs.

The only rotation instructions supported by the 8-bit AVR architecture are rotations of 8-bit operands by one bit to the left (`rol`) and right (`ror`). Therefore, rotations of 8-bit words by other amounts, and also rotations of operands that are longer than eight bits, have to be carried out by executing a sequence of 1-bit rotations and other instructions [10]. For example, a 1-bit left-rotation of a 64-bit operand consists of a 1-bit logical left-shift (`lsl`), followed by seven 1-bit left-rotations through carry (`rol`), and an addition with carry (`adc`). The overall execution time of this sequence of instructions is nine clock cycles (see [10, p. 251] for further details). A rotation of a 64-bit operand by n bits, where

$1 < n < 8$, can be computed by repeating n times the sequence of instructions for rotating a 64-bit quantity by one bit in the same direction. However, it has to be pointed out that a left-rotation and a right-rotation by one and the same number of bits can have different execution times. For example, a right-rotation of a 64-bit operand by one bit requires ten cycles, which is one cycle more than what is needed for a left-rotation. Consequently, it makes sense to compare the two possible options for a rotation, namely rotation by n bits to the left versus rotation by $(64 - n)$ bits to the right, to select the most efficient one [10].

By combining our efficient implementations of the rotations with the XORs performed in each of the four sigma operations, we were able to further reduce the execution time of the compression function. Our basic idea is to exploit the fact that a bytewise rotation of a 64-bit operand can be executed for free when it is XORed with another 64-bit operand. More concretely, an operation of the form $x = x \oplus (y \ggg 8)$ can be performed with just eight `eor` instructions on an AVR processor by simply XORing the bytes of x and y in such a way that the bytewise rotation is "implicitly" carried out. This approach can also be applied for rotations by other amounts. For example, the operation $x = x \oplus (y \ggg 7)$ is normally implemented as a bitwise rotation to the left by one bit (which takes nine clock cycles), followed by a bytewise right-rotation (also nine cycles), and an XOR (eight cycles). However, by integrating the bytewise right-rotation into the XOR, the total execution time decreases to 17 clock cycles. Similar savings in execution time are possible when rotations by a larger number of bits have to be performed since our idea also works for multiples of eight bits.

The "small" sigma operations σ_0, σ_1 are used in the message schedule, while the "big" sigma operations Σ_0, Σ_1 are essential components of the compression function. They can be expressed through the following formulae [17].

$$\begin{aligned}
\sigma_0 &= (x \ggg 1) \oplus (x \ggg 8) \oplus (x \gg 7) \\
\sigma_1 &= (x \ggg 19) \oplus (x \ggg 61) \oplus (x \gg 6) \\
\Sigma_0 &= (x \ggg 28) \oplus (x \ggg 34) \oplus (x \ggg 39) \\
\Sigma_1 &= (x \ggg 14) \oplus (x \ggg 18) \oplus (x \ggg 41)
\end{aligned}$$

By applying the optimization techniques described above, the formulae for the four sigma operations can be rewritten to clearly show the bitwise and bytewise rotations and whether they go to the left or to the right. Rotations by amounts that are multiple of eight, marked in green, are executed implicitly (i.e. merged into a 64-bit XOR operation) and do not increase the execution time.

$$\begin{aligned}
\sigma_0 &= (x \ggg 1) \oplus (x \ggg 8) \oplus (((x \lll 1) \ggg 8) \wedge (2^{57} - 1)) \\
\sigma_1 &= (((x \lll 2) \ggg 8) \wedge t) \oplus ((x \lll 2) \lll 1) \oplus ((((x \lll 2) \lll 1) \lll 2) \ggg 24) \\
\Sigma_0 &= ((x \ggg 40) \lll 1) \oplus ((x \ggg 2) \ggg 32) \oplus (((x \ggg 2) \ggg 2) \ggg 24) \\
\Sigma_1 &= ((x \ggg 16) \lll 2) \oplus ((x \ggg 1) \ggg 40) \oplus (((x \ggg 1) \ggg 1) \ggg 16)
\end{aligned}$$

We implemented the four sigma operations in AVR Assembly language using the optimized formulae given above. The value of t on the right side of the formula

Table 1. Execution time and code size of the sigma operations on the ATmega128.

Operation	Assembler		C language	
	Time (cycles)	Code size (bytes)	Time (cycles)	Code size (bytes)
σ_0	41	82	525	168
σ_1	71	142	399	168
Σ_0	71	142	431	168
Σ_1	58	116	521	168

for σ_1 is $2^{58} - 1$ so that it can be used to mask off the six most-significant bits of a 64-bit operand. Our implementations perform the rotations by a multiple of eight bit-positions implicitly (i.e. "merged" into an XOR); only the rotations by less than eight bits to the left or right are actually executed. Table 1 shows the execution time and code size of our optimized Assembler implementations and that obtained by compiling C code for the sigma functions. The C sources were compiled with avr-gcc, which uses generic rotation functions from the low-level runtime library libgcc, such as `rotldi`, to rotate a 64-bit operand. Our results show that the optimized Assembler implementations are between 5.6 and 12.8 times faster and up to 2.04 times smaller than their C counterparts.

4.2 Optimization of the Memory Accesses

As analyzed in Subsect. 3.2, one of the reasons for the suboptimal performance of the binary code generated by the avr-gcc compiler is an unnecessarily large number of memory access to load or store the eight working variables (or parts of them) from/to the SRAM. Especially costly is the word-wise rotation of the set of working variables that has to be carried out during each iteration of the compression function as is explained in Sect. 2 and illustrated in Fig. 2. Some implementations unroll (or partially unroll) the loops to avoid these word-wise rotation, which comes at the expense of increased code size. To overcome this problem, we developed a special optimization strategy that maintains two sets of working variables in SRAM (i.e. we "duplicate" the eight working variables) and intensively uses the indirect addressing mode with displacement [2] of the AVR architecture to avoid the word-wise rotation in each iteration *without* the need to unroll the main loop of the compression function. In this way, we were able to achieve both high performance and small code size.

The indirect addressing mode with displacement is one of several modes to address data in SRAM and can be used with the 16-bit pointer registers `Y` and `Z`. This mode forms the actual address by adding a "displacement" (sometimes called *offset*) to the content of a pointer register. In AVR, the displacement is a 6-bit constant that is embedded into the instruction word (i.e. the maximum displacement is 63). Other embedded RISC architectures like ARM are more flexible since the displacement (or offset) does not need to be constant but can also be a variable stored in a general-purpose register. Two AVR instructions

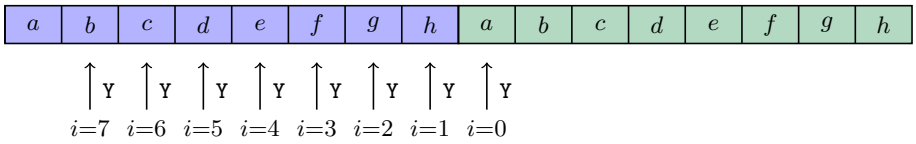


Fig. 3. Graphical representation of the "duplicated" set of working variables stored in an array of 128 bytes.

that support indirect addressing with displacement are `ldd` and `std`, which can be used to efficiently access a given element of an array when its index is fixed and known a priori (i.e. at compile time). Concretely, when pointer `Y` holds the start address of a byte-array S, then the Assembler statement `ldd r4,Y+2` can be used to load $S[2]$ (i.e. the third byte of the array S) to register `r4`, whereby the pointer `Y` itself does not get modified in any way.

Our optimized Assembler implementation of the compression function uses two sets of working variables to simplify the address calculations. At the beginning (i.e. before the very first iteration), the second set is simply a copy of the first one, which means both sets are identical. The two sets are stored in a byte array S of length 128 bytes in SRAM in such a way that the first set occupies the upper half of the array (i.e. the bytes from $S[64]$ to $S[127]$) and the second set the lower half ($S[0]$ to $S[63]$). Figure 3 illustrates the byte array S with the first set of working variables colored in green on the right side and the second set in blue on the left side. We use pointer-register `Y` to access the array. In the very first iteration (i.e. in the iteration with loop-counter $i = 0$), `Y` contains the start address of working variable a of the first set (i.e. the least-significant byte of the green a), which is the address of $S[64]$. All eight working variables of the first (green) set can be conveniently accessed via the indirect addressing mode with displacement since the offset will never be bigger than 63. Thanks to this addressing mode, no costly address arithmetic needs to be performed to obtain the actual addresses of the bytes of the working variables. Our implementation first computes the *Maj* operation, which requires to load the working variables a, b, and c from the green set. The offsets for these three working variables are $0-7$, $8-15$, and $16-23$, respectively. Thereafter, Σ_0, *Ch*, and Σ_1 are computed and finally the temporary values t_1 and t_2. In a "conventional" implementation of the compression function (i.e. an implementation that actually performs the word-wise rotation of the set of working variables), the sum $t_1 + t_2$ is assigned to a and the sum $d + t_1$ to e (see Sect. 2). However, our implementation does not perform a word-wise rotation and, therefore, $d + t_1$ is written to the green d and $t_1 + t_2$ to the blue h. The last step is to decrement `Y` by eight.

At the beginning of the second iteration (i.e. the iteration where $i = 1$), the pointer-register `Y` contains the address of the least-significant byte of the blue h, i.e. the address of $S[56]$. The second iteration is carried out quite similar to the first iteration, which means we start with computing the *Maj* operation as before. This computation requires to load the blue h and the green a, b from

SRAM. As pointed out above, our implementation does not rotate the working variables word-wise, and therefore the blue h contains the sum $t_1 + t_2$ that was calculated in the previous iteration (i.e. the blue h corresponds to variable a in a "rotated" implementation). These variables have the offsets $0-7$, $8-15$, and $16-23$, respectively, exactly like in the first iteration. Also Σ_0, Ch, and Σ_1 are computed in the same fashion as above. At the end of the second iteration, the pointer register `Y` is again decremented by eight. Subsequent iterations are also performed in this way. It is important to understand that in each iteration, the offsets used to access the bytes of the working variables are always the same as in the first iteration, only the base address in `Y` is different. In this way, we can avoid the word-wise rotations without loop unrolling. However, after eight iterations, the two sets of working variables must be "synchronized," which means the second (blue) set has to be copied to the first (green) set so that both sets are identical again. Fortunately, this duplication of the variables is only needed in every eighth iteration; therefore, it is much more efficient than rotating the set of working variables word-wise in each iteration. The only drawback of this approach is a slight increase in RAM consumption (by 64 bytes) since two sets of working variables are needed.

5 Results and Comparison

We developed besides the AVR Assembler implementation of the compression function also a C version based on the same optimization techniques using two sets of working variables. All other parts of the SHA-512 algorithm, such as the padding, were written only in C since they are not really performance-critical [19]. Our software provides both a high-level and a low-level API, whereby the former consists of just a single function, namely `sha512_hash`. The low-level API, on the other hand, comes with the standard IUF (`init`, `update`, `final`) functions that allow for hashing of very large or fragmented data without the need to have the full data in SRAM.

We used Atmel Studio v7.0 as development environment with an extension that provides the 8-bit AVR GNU toolchain including avr-gcc version 5.4.0. All execution times reported in this section were determined with the help of the cycle-accurate instruction set simulator of Atmel Studio, whereby we used the ATmega128 microcontroller as target device. We simulated the execution time of the Assembler version and the C version of the compression function alone and also the time required to hash a 500-byte message so that we can compare our results with those from previous papers such as [4]. The latter performance test was carried out with the help of the high-level API by simply calling the `sha512_hash` function. Our simulations gave an execution time of 59768 cycles for the AVR-Assembler version of the compression function and 264133 cycles for the C implementation. In both cases, the function call overhead is included in the specified cycle count. These execution times represent compression rates of approximately 467 and 2064 cycles per byte, respectively, which means the Assembler implementation outperforms the C version by a factor of 4.42. We

Table 2. Performance (i.e. hash rate when hashing a 500-byte message) and code size of different hash functions on the ATmega128.

Reference	Algorithm	Impl.	Hash rate (cyc/byte)	Code size (bytes)
This paper	SHA-512	C	2654	4610
This paper	SHA-512	C+Asm	611	3460
Weatherley [20]	SHA-512	C++	2683	8072
Osvik [19]	SHA-256 (CP)	Asm	335	2720
Balasch et al. [4]	SHA-256	Asm	532	1090
Balasch et al. [4]	Blake (256 bit)	Asm	562	1166
Balasch et al. [4]	Grøstl (256 bit)	Asm	686	1400
Balasch et al. [4]	Keccak (256 bit)	Asm	1432	868
Balasch et al. [4]	Photon (256 bit)	Asm	6210	1244

also simulated the execution time of the compression function of the SHA-512 software contained in the Arduino Cryptography Library [20] and found it to be slightly slower than our own C implementation.

When hashing a message of length 500 bytes (using the high-level function `sha512_hash`), we obtained an overall execution time of 1327132 cycles for the "pure" C implementation and 305303 cycles for the Assembler version. These two cycle counts translate to hash rates of approximately 611 and 2654 cycles per byte, respectively, which means the Assembler optimizations we proposed yield a speed-up by a factor of about 4.34 over the C code. The hash rates are significantly worse than the compression rates, which is mainly because of the padding. Normally, a 500-byte message fits into four 128-byte blocks, but due to padding, the compression function gets executed five times altogether. If we would hash e.g. a 620-byte message, we get exactly the same execution time as for a 500-byte message (since in both cases the compression function is called five times), but the hash rate is much better because the cycle count is divided by 620 instead of 500. Table 2 compares our results with that of a few previous SHA-2 implementations and three SHA-3 candidates from [4]. It is remarkable that our Assembler version of the SHA-512 hash algorithm is only slightly less efficient than Balasch et al.'s Assembler implementation of SHA-256 [4], even though using a 500-byte message for benchmarking favors SHA-256 over SHA-512. Namely, as pointed out before, hashing 500 bytes with SHA-512 requires five calls of the compression function (whereby each time a 128-byte block gets processed), but in the case of SHA-256 the compression function is only called eight times (to process 64 bytes each time). When we hash a 620-byte message instead of the 500-byte message, then our SHA-512 software actually achieves a *better* hash rate than the SHA-256 implementation of Balasch et al.

The code size of our implementation, including both the high-level function `sha512_hash` and the low-level functions `init`, `update`, `final`, along with the compression function (in Assembler), amounts to 3460 bytes. The compression function alone has a size of 2206 bytes and occupies 158 bytes on the stack.

6 Conclusions

We demonstrated that the execution time of SHA-512 on 8-bit AVR microcontrollers can be significantly improved through hand-optimized Assembler code for the compression function. Our implementation of the compression function takes 59768 clock cycles on an ATmega128 microcontroller, which corresponds to a compression rate of approximately 467 cycles per byte. For comparison, an implementation in C compiled with avr-gcc 5.4.0 is about 4.42 times slower as it requires 264133 cycles (i.e. roughly 2064 cycles per byte). Hashing a 500-byte message with and without Assembler optimizations takes 305303 and 1327132 cycles, respectively, which represents a nearly 4.35-fold difference in execution time. We achieved this performance gain through a careful optimization of the four sigma operations and by minimizing the cost of memory (SRAM) accesses via a novel approach that duplicates the working variables but does not require loop unrolling. The latter optimization technique can potentially be applied in various other contexts beyond the acceleration of SHA-512 on AVR. It can be easily adapted to other architectures that feature an indirect addressing mode (e.g. MSP430, ARM) and may be useful for other cryptosystems that perform word-wise rotation of working variables or a state. We hope that our work will contribute to a more wide-spread deployment of SHA-512 (and cryptosystems that use SHA-512, in particular EdDSA) on constrained IoT devices.

References

1. C. Alippi, A. Bogdanov, and F. Regazzoni. Lightweight cryptography for constrained devices. In *Proceedings of the 14th International Symposium on Integrated Circuits (ISIC 2014)*, pages 144–147. IEEE, 2014.
2. Atmel Corporation. 8-bit AVR Instruction Set. User guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, 2008.
3. Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, 2008.
4. J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Pöppelmann, F. Regazzoni, F.-X. Standaert, G. Van Assche, R. Van Keer, L. van Oldeneel tot Oldenzeel, and I. von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. In S. Mangard, editor, *Smart Card Research and Advanced Applications — CARDIS 2012*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 2013.
5. D. J. Bernstein. Quantum algorithms to find collisions. The cr.yp.to blog, available online at `http://blog.cr.yp.to/20171017-collisions.html`, 2017.
6. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer Verlag, 2011.
7. G. Brassard, P. Høyer, and A. Tapp. Quantum cryptanalysis of hash and claw-free functions. In C. L. Lucchesi and A. V. Moura, editors, *LATIN '98: Third Latin*

American Symposium on Theoretical Informatics, volume 1380 of *Lecture Notes in Computer Science*, pages 163–169. Springer Verlag, 1998.
8. X. Cao and M. O'Neill. Application-oriented SHA-256 hardware design for low-cost RFID. In *Proceedings of the 45th IEEE International Symposium on Circuits and Systems (ISCAS 2012)*, pages 1412–1415. IEEE, 2012.
9. A. Chailloux, M. Naya-Plasencia, and A. Schrottenloher. An efficient quantum collision search algorithm and implications on symmetric cryptography. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology — ASIACRYPT 2017*, volume 10625 of *Lecture Notes in Computer Science*, pages 211–240. Springer Verlag, 2017.
10. D. Dinu. *Efficient and Secure Implementations of Lightweight Symmetric Cryptographic Primitives*. PhD thesis, University of Luxembourg, 2017.
11. L. K. Grover. A fast quantum mechanical algorithm for database search. In G. L. Miller, editor, *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC 1996)*, pages 212–219. ACM Press, 1996.
12. S. Gueron, S. Johnson, and J. Walker. SHA-512/256. Cryptology ePrint Archive, Report 2010/548, 2010. Available for download at `http://eprint.iacr.org/2010/548`.
13. M. Hutter and P. Schwabe. NaCl on 8-bit AVR microcontrollers. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *Progress in Cryptology — AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer Verlag, 2013.
14. S. Josefsson and I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). Internet Research Task Force, Crypto Forum Research Group, RFC 8032, Jan. 2017.
15. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, 1996.
16. E. Nascimento, J. López, and R. Dahab. Efficient and secure elliptic curve cryptography for 8-bit AVR microcontrollers. In R. S. Chakraborty, P. Schwabe, and J. A. Solworth, editors, *Security, Privacy, and Applied Cryptography Engineering — SPACE 2015*, volume 9354 of *Lecture Notes in Computer Science*, pages 289–309. Springer Verlag, 2015.
17. National Institute of Standards and Technology (NIST). Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-4, available for download at `http://dx.doi.org/10.6028/NIST.FIPS.180-4`, Aug. 2015.
18. National Institute of Standards and Technology (NIST). Report on Post-Quantum Cryptography. Internal Report 8105, available for download at `http://dx.doi.org/10.6028/NIST.IR.8105`, Apr. 2016.
19. D. A. Osvik. Fast embedded software hashing. Cryptology ePrint Archive, Report 2012/156, 2012. Available for download at `http://eprint.iacr.org/2012/156`.
20. R. Weatherley. Arduino Cryptography Library. Source code, available online at `http://github.com/rweather/arduinolibs`, 2018.
21. C. Wenzel-Benner, J. Gräf, J. Pham, and J.-P. Kaps. XBX benchmarking results January 2012. In *Proceedings of the 3rd SHA-3 Candidates Conference*, 2012.
22. S. Zhou and M. A. Brown. Smart meter deployment in Europe: A comparative case study on the impacts of national policy schemes. *Journal of Cleaner Production*, 144:22–32, Feb. 2017.