UvA-DARE (Digital Academic Repository)

Instruction sequence expressions for the secure hash algorithm SHA-256

Bergstra, J.A.; Middelburg, C.A.

Link to publication

Instruction Sequence Expressions for the Secure Hash Algorithm SHA-256

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 904, 1098 XH Amsterdam, the Netherlands
`J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl`

Abstract. The secure hash function SHA-256 is a function on bit strings. This means that its restriction to the bit strings of any given length can be computed by a finite instruction sequence that contains only instructions to set and get the content of Boolean registers, forward jump instructions, and a termination instruction. We describe such instruction sequences for the restrictions to bit strings of the different possible lengths by means of uniform terms from an algebraic theory.

Keywords: SHA-256, secure hash algorithm, secure hash function, single-pass instruction sequence.

1998 ACM Computing Classification: E.3, F.1.1.

1 Introduction

SHA-256 is one of the hash functions defined in the Secure Hash Standard of the U.S. National Institute of Standards and Technology [20]. To phrase it more precisely, the standard describes an algorithm that computes the hash function SHA-256 by means of pseudo-code. In this paper, unlike the standard, an algorithm that computes a function is distinguished from the computed function. SHA-256 is called a secure hash function because it is a hash function for which it is expected to be computationally infeasible to find an input with a given hash value and to find two different inputs with the same hash value. SHA-256 is implemented in some widely used security applications and protocols, including Bitcoin [15], S/MIME [17], TLS [8], SSH [19], and IPsec [11].

To our knowledge, the starting point of studies of the security of SHA-256 keeps being the above-mentioned pseudo-code description of an algorithm that computes it (see e.g. [10,12,13,14,16,18]). SHA-256 restricted to the bit strings of a given length can be computed by a finite single-pass instruction sequence that contains only instructions to set and get the content of Boolean registers, forward jump instructions, and a termination instruction (see [6]). In this paper, we describe such instruction sequences for the restrictions to bit strings of the different possible lengths by means of uniform terms from an algebraic theory of single-pass instruction sequences. Thus, we provide a mathematically precise alternative to the pseudo-code description from the standard.

In computer science, the meaning of programs usually plays a prominent part in the explanation of many issues concerning programs. Moreover, what is taken for the meaning of programs is mathematical by nature. Yet, it is customary that practitioners do not fall back on the mathematical meaning of programs in case explanation of issues concerning programs is needed. They phrase their explanations from an empirical perspective. An attempt to approach the semantics of programming languages from the emperical perspective that a program is in essence an instruction sequence is made in [1]. The groundwork for the approach is an algebraic theory of single-pass instruction sequences, called program algebra, and an algebraic theory of mathematical objects that represent the behaviours produced by instruction sequences under execution, called basic thread algebra.

As a continuation of the work on the approach to programming language semantics followed in [1], (a) the notion of an instruction sequence was subjected to systematic and precise analysis using the groundwork laid earlier and (b) selected issues relating to well-known subjects from the theory of computation and the area of computer architecture were rigorously investigated thinking in terms of instruction sequences. This led among other things to expressiveness results about the instruction sequences considered, variations of the instruction sequences considered, an analysis of the autosolvability requirement implicit in Turing's result regarding the undecidability of the halting problem, and an analysis of the effects of the presence of indirect jump instructions in the instruction set of a computer on points such as instruction sequence size and instruction sequence performance (see e.g. [2,3,5,7]).

The general aim of the above-mentioned continuation of the work on the approach to programming language semantics followed in [1] is to bring instruction sequences as a theme in computer science better into the picture. This is the general aim of the work presented in the current paper as well. Different from usual in the work referred to above, the accent is this time on a practical problem, viz. devising instruction sequences that compute the restrictions of SHA-256 to the bit strings of the different possible lengths. As in the work referred to above, this work is carried out in the setting of program algebra.

This paper is organized as follows. First, we survey program algebra and the particular fragment and instantiation of it that is used in this paper (Section 2). Next, we describe how we deal with 32-bit words by means of Boolean registers (Section 3) and how we compute the basic and derived operations on 32-bit words that are used in the standard to define SHA-256 (Section 4). Then, we give the description of instruction sequences that define SHA-256 (Section 5). Finally, we make some concluding remarks (Section 6).

2 Program Algebra

In this section, we present a brief outline of PGA (ProGram Algebra) and the particular fragment and instantiation of it that is used in the remainder of this paper. A mathematically precise treatment can be found in [6].

The starting-point of PGA is the simple and appealing perception of a sequential program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over.

It is assumed that a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions* has been given. The intuition is that the execution of a basic instruction may modify a state and produces a reply at its completion. The possible replies are 0 and 1. The actual reply is generally state-dependent. Therefore, successive executions of the same basic instruction may produce different replies. The set $\mathfrak{A}$ is the basis for the set of instructions that may occur in the instruction sequences considered in PGA. The elements of the latter set are called *primitive instructions*. There are five kinds of primitive instructions, which are listed below:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- a *termination instruction* !.

We write $\mathfrak{I}$ for the set of all primitive instructions.

On execution of an instruction sequence, these primitive instructions have the following effects:

- the effect of a positive test instruction $+a$ is that basic instruction a is executed and execution proceeds with the next primitive instruction if 1 is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one — if there is no primitive instruction to proceed with, inaction occurs;
- the effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the value produced reversed;
- the effect of a plain basic instruction a is the same as the effect of $+a$, but execution always proceeds as if 1 is produced;
- the effect of a forward jump instruction $\#l$ is that execution proceeds with the lth next primitive instruction of the instruction sequence concerned — if l equals 0 or there is no primitive instruction to proceed with, inaction occurs;
- the effect of the termination instruction ! is that execution terminates.

To build terms, PGA has a constant for each primitive instruction and two operators. These operators are: the binary concatenation operator $\,;\,$ and the unary repetition operator $^\omega$. We use the notation $\overset{\bullet}{\underset{i=0}{\,;\,}}{}^{n} P_i$, where $P_0, \ldots, P_n$ are PGA terms, for the PGA term $P_0\,;\,\ldots\,;\,P_n$.

The instruction sequences that concern us in the remainder of this paper are the finite ones, i.e. the ones that can be denoted by closed PGA terms in which the repetition operator does not occur. Moreover, the basic instructions that concern us are instructions to set and get the content of Boolean registers.

More precisely, we take the set

$$\{\mathsf{in}{:}i.\mathsf{get} \mid i \in \mathbb{N}^+\} \cup \{\mathsf{out}{:}i.\mathsf{set}{:}b \mid i \in \mathbb{N}^+ \wedge b \in \{0,1\}\}$$
$$\cup \{\mathsf{aux}{:}i.\mathsf{get} \mid i \in \mathbb{N}^+\} \cup \{\mathsf{aux}{:}i.\mathsf{set}{:}b \mid i \in \mathbb{N}^+ \wedge b \in \{0,1\}\}$$

as the set $\mathfrak{A}$ of basic instructions.

Each basic instruction consists of two parts separated by a dot. The part on the left-hand side of the dot plays the role of the name of a Boolean register and the part on the right-hand side of the dot plays the role of a command to be carried out on the named Boolean register. For each $i \in \mathbb{N}^+$:

- $\mathsf{in}{:}i$ serves as the name of the Boolean register that is used as ith input register in instruction sequences;
- $\mathsf{out}{:}i$ serves as the name of the Boolean register that is used as ith output register in instruction sequences;
- $\mathsf{aux}{:}i$ serves as the name of the Boolean register that is used as ith auxiliary register in instruction sequences.

On execution of a basic instruction, the commands have the following effects:

- the effect of get is that nothing changes and the reply is the content of the named Boolean register;
- the effect of $\mathsf{set}{:}0$ is that the content of the named Boolean register becomes 0 and the reply is 0;
- the effect of $\mathsf{set}{:}1$ is that the content of the named Boolean register becomes 1 and the reply is 1.

Let $n, m \in \mathbb{N}$, let $f : \{0,1\}^n \to \{0,1\}^m$, and let X be a finite instruction sequence that can be denoted by a closed PGA term in the case that $\mathfrak{A}$ is taken as specified above. Then X *computes* f if there exists a $k \in \mathbb{N}$ such that for all $b_1, \ldots, b_n \in \{0,1\}$: if X is executed in an environment with n input registers, m output registers, and k auxiliary registers, the content of the input registers with names $\mathsf{in}{:}1, \ldots, \mathsf{in}{:}n$ are $b_1, \ldots, b_n$ when execution starts, and the content of the output registers with names $\mathsf{out}{:}1, \ldots, \mathsf{out}{:}m$ are $b'_1, \ldots, b'_m$ when execution terminates, then $f(b_1, \ldots, b_n) = b'_1, \ldots, b'_m$.

3   Dealing with 32-Bit Words

This section is concerned with dealing with bit strings of length 32 by means of Boolean registers. It contains definitions which facilitate the description of instruction sequences that define SHA-256 in Section 5. In the sequel, bit strings of length 32 will mostly be called 32-*bit words* or shortly *words*.

Let $\kappa \in \{\mathsf{in}, \mathsf{out}, \mathsf{aux}\}$, let $i \in \mathbb{N}^+$, and let $\kappa{:}i$ be the name of a Boolean register. Then κ and i are called the *kind* and *number* of the Boolean register. Successive Boolean registers are Boolean registers of the same kind with successive numbers. Words are stored by means of Boolean registers such that the successive bits of a stored word are the content of successive Boolean registers and the first bit of the word is the content of a Boolean register whose number is in the set $\{n \in \mathbb{N} \mid n \bmod 32 = 1\}$.

4

The words that form a part of the message to which SHA-256 is to be applied are stored in advance of the computation in input registers, starting with the input register with number 1, the words that form a part of the message digest that results from applying SHA-256 are stored during the computation in output registers, starting with the output register with number 1, and the words that form a part of intermediate results that arise during the computation, such as message schedules, hash values, and working values, are stored in auxiliary registers.

It is convenient to have available the names used in the standard for the words of the message blocks, the message schedule, the hash value, the working values, and the temporary values in the current setting for the Boolean registers that contain the least significant bit of these words. It is also convenient to have available the names $D_0, \ldots, D_7$ for the Boolean registers that contain the least significant bit of the words of the message digest, the names $t_1, \ldots, t_6, t'_1, \ldots, t'_4$ for the Boolean registers that contain the least significant bit of the words of additional intermediate values that are temporarily stored,[1] and the name cb for the Boolean register that contains the carry bit that is repeatedly stored when computing the addition operation. Therefore, we define:

$$M_j^{(i)} \triangleq \text{in:}k \quad \text{where } k = 512 \cdot (i-1) + 32 \cdot j + 1 \quad (1 \le i \le 2^{55}, 0 \le j \le 15),$$
$$W_j \quad \triangleq \text{aux:}k \ \text{ where } k = 32 \cdot j + 1 \qquad\qquad (0 \le j \le 63),$$
$$H_j \quad \triangleq \text{aux:}k \ \text{ where } k = 32 \cdot j + 2049 \qquad (0 \le j \le 7),$$
$$a \triangleq \text{aux:2305},\ b \triangleq \text{aux:2337},\ c \triangleq \text{aux:2369},\ d \triangleq \text{aux:2401},\ e \triangleq \text{aux:2433},$$
$$f \triangleq \text{aux:2465},\ g \triangleq \text{aux:2497},\ h \triangleq \text{aux:2529},\ T_1 \triangleq \text{aux:2561},\ T_2 \triangleq \text{aux:2593},$$
$$t_1 \triangleq \text{aux:2625},\ t_2 \triangleq \text{aux:2657},\ t_3 \triangleq \text{aux:2689},\ t_4 \triangleq \text{aux:2721},\ t_5 \triangleq \text{aux:2753},$$
$$t_6 \triangleq \text{aux:2785},\ t'_1 \triangleq \text{aux:2817},\ t'_2 \triangleq \text{aux:2849},\ t'_3 \triangleq \text{aux:2881},\ t'_4 \triangleq \text{aux:2913},$$
$$cb \triangleq \text{aux:2945},$$
$$D_j \quad \triangleq \text{out:}k \ \text{ where } k = 32 \cdot j + 1 \qquad\qquad (0 \le j \le 7).$$

It is also convenient to have available the names used in the standard for the words of the initial hash value:

$$H_0^{(0)} \triangleq 01101010000010011111001100110011 ,$$
$$H_1^{(0)} \triangleq 10111011011001111010111010000101 ,$$
$$H_2^{(0)} \triangleq 00111100011011101111001101110010 ,$$
$$H_3^{(0)} \triangleq 10100101010011111111010100111010 ,$$
$$H_4^{(0)} \triangleq 01010001000011100101001001111111 ,$$
$$H_5^{(0)} \triangleq 10011011000001010110100010001100 ,$$
$$H_6^{(0)} \triangleq 00011111100000111101100110101011 ,$$
$$H_7^{(0)} \triangleq 01011011111000011001101000011001 ;$$

---

[1] The Boolean registers with names $t'_1, \ldots, t'_4$ are reserved for the least significant bit of intermediate values that arise when computing one of the derived operations on bit strings introduced in Section 4.

and the names used in the standard for the "SHA-256 constants":

$$K_0 \triangleq 01000010100010100010111110011000 \, ,$$
$$K_1 \triangleq 01110001001101110100010010010001 \, ,$$
$$\vdots$$
$$K_{63} \triangleq 11000110011100010111100011110010 \, .^2$$

4 Computing Operations on 32-Bit Words

This section is concerned with computing operations on bit strings of length 32. It contains definitions which facilitate the description of instruction sequences that define SHA-256 in Section 5.

The basic operations on bit strings that are relevant to SHA-256 are bitwise negation, bitwise conjunction, bitwise exclusive disjunction, shift right n positions, rotate right n positions $(0 < n < 32)$, and addition. For these operations, we define parameterized instruction sequences computing them in case the parameters are properly instantiated (see below):

$NOT(s{:}k, d{:}l) \triangleq$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{31} (d{:}l{+}i.\mathsf{set}{:}0 \, ; -s{:}k{+}i.\mathsf{get} \, ; d{:}l{+}i.\mathsf{set}{:}1) \, ,$

$AND(s_1{:}k_1, s_2{:}k_2, d{:}l) \triangleq$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{31} (d{:}l{+}i.\mathsf{set}{:}0 \, ; -s_1{:}k_1{+}i.\mathsf{get} \, ; \#4 \, ; -s_2{:}k_2{+}i.\mathsf{get} \, ; \#2 \, ; d{:}l{+}i.\mathsf{set}{:}1) \, ,$

$XOR(s_1{:}k_1, s_2{:}k_2, d{:}l) \triangleq$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{31} (d{:}l{+}i.\mathsf{set}{:}0 \, ; -s_1{:}k_1{+}i.\mathsf{get} \, ; \#4 \, ; -s_2{:}k_2{+}i.\mathsf{get} \, ; \#5 \, ; \#3 \, ;$
$\qquad\qquad\qquad\qquad\qquad\qquad +s_2{:}k_2{+}i.\mathsf{get} \, ; \#2 \, ; d{:}l{+}i.\mathsf{set}{:}1) \, ,$

$SHR^n(s{:}k, d{:}l) \triangleq$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{31-n} (d{:}l{+}i.\mathsf{set}{:}0 \, ; +s{:}k{+}i{+}n.\mathsf{get} \, ; d{:}l{+}i.\mathsf{set}{:}1) \, ;$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{n-1} (d{:}l{+}i{+}32{-}n.\mathsf{set}{:}0) \, ,$

$ROTR^n(s{:}k, d{:}l) \triangleq$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{31-n} (d{:}l{+}i.\mathsf{set}{:}0 \, ; +s{:}k{+}i{+}n.\mathsf{get} \, ; d{:}l{+}i.\mathsf{set}{:}1) \, ;$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{n-1} (d{:}l{+}i{+}32{-}n.\mathsf{set}{:}0 \, ; +s{:}k{+}i.\mathsf{get} \, ; d{:}l{+}i{+}32{-}n.\mathsf{set}{:}1) \, ,$

$ADD(s_1{:}k_1, s_2{:}k_2, d{:}l) \triangleq$
$\quad cb.\mathsf{set}{:}0 \, ;$
$\quad {\overset{\bullet}{\underset{i=0}{,}}}^{31} (d{:}l{+}i.\mathsf{set}{:}0 \, ; -s_1{:}k_1{+}i.\mathsf{get} \, ; \#7 \, ; -s_2{:}k_2{+}i.\mathsf{get} \, ; \#10 \, ; -cb.\mathsf{get} \, ; \#10 \, ;$
$\qquad d{:}l{+}i.\mathsf{set}{:}1 \, ; \#8 \, ; -s_2{:}k_2{+}i.\mathsf{get} \, ; \#8 \, ; -cb.\mathsf{get} \, ; \#8 \, ; \#3 \, ; -cb.\mathsf{get} \, ; \#5 \, ;$
$\qquad cb.\mathsf{set}{:}1 \, ; \#5 \, ; -cb.\mathsf{get} \, ; \#2 \, ; d{:}l{+}i.\mathsf{set}{:}1 \, ; cb.\mathsf{set}{:}0) \, ,$

---

[2] All 64 definitions have been put into an appendix.

where s, s_1, s_2 range over $\{\mathsf{in}, \mathsf{aux}\}$, d ranges over $\{\mathsf{aux}, \mathsf{out}\}$, and k, k_1, k_2, l range over $\{n \in \mathbb{N} \mid n \bmod 32 = 1\}$. For each of these parameterized instruction sequences, all but the last parameter correspond to the operands of the operation concerned and the last parameter corresponds to the result of the operation concerned.

The intended operations are computed provided that the instantiation of the last parameter differs from the instantiation of each of the other parameters. We could have prevented this condition at the cost of longer instruction sequences. In this paper, the condition will always be satisfied.

In the standard, for SHA-256, six derived operations on bit strings are defined in terms of the above-mentioned basic operations.[3] For these operations, we define parameterized instruction sequences computing them:

$$CH(s_1{:}k_1, s_2{:}k_2, s_3{:}k_3, d{:}l) \triangleq$$
$$NOT(s_1{:}k_1, t'_1)\,;\,AND(s_1{:}k_1, s_2{:}k_2, t'_2)\,;\,AND(t'_1, s_3{:}k_3, t'_3)\,;$$
$$XOR(t'_2, t'_3, d{:}l)\,,$$

$$MAJ(s_1{:}k_1, s_2{:}k_2, s_3{:}k_3, d{:}l) \triangleq$$
$$AND(s_1{:}k_1, s_2{:}k_2, t'_1)\,;\,AND(s_1{:}k_1, s_3{:}k_3, t'_2)\,;\,AND(s_2{:}k_2, s_3{:}k_3, t'_3)\,;$$
$$XOR(t'_1, t'_2, t'_4)\,;\,XOR(t'_3, t'_4, d{:}l)\,,$$

$$\Sigma_0(s{:}k, d{:}l) \triangleq$$
$$ROTR^2(s{:}k, t'_1)\,;\,ROTR^{13}(s{:}k, t'_2)\,;\,ROTR^{22}(s{:}k, t'_3)\,;$$
$$XOR(t'_1, t'_2, t'_4)\,;\,XOR(t'_3, t'_4, d{:}l)\,,$$

$$\Sigma_1(s{:}k, d{:}l) \triangleq$$
$$ROTR^6(s{:}k, t'_1)\,;\,ROTR^{11}(s{:}k, t'_2)\,;\,ROTR^{25}(s{:}k, t'_3)\,;$$
$$XOR(t'_1, t'_2, t'_4)\,;\,XOR(t'_3, t'_4, d{:}l)\,,$$

$$\sigma_0(s{:}k, d{:}l) \triangleq$$
$$ROTR^7(s{:}k, t'_1)\,;\,ROTR^{18}(s{:}k, t'_2)\,;\,SHR^3(s{:}k, t'_3)\,;$$
$$XOR(t'_1, t'_2, t'_4)\,;\,XOR(t'_3, t'_4, d{:}l)\,,$$

$$\sigma_1(s{:}k, d{:}l) \triangleq$$
$$ROTR^{17}(s{:}k, t'_1)\,;\,ROTR^{19}(s{:}k, t'_2)\,;\,SHR^{10}(s{:}k, t'_3)\,;$$
$$XOR(t'_1, t'_2, t'_4)\,;\,XOR(t'_3, t'_4, d{:}l)\,,$$

where s, s_1, s_2, s_3 range over $\{\mathsf{in}, \mathsf{aux}\}$, d ranges over $\{\mathsf{aux}, \mathsf{out}\}$, k, k_1, k_2, k_3, l range over $\{n \in \mathbb{N} \mid n \bmod 32 = 1\}$.

We also define a parameterized instruction sequence by which the successive bits in a constant 32-bit word become the content of 32 successive Boolean registers and a parameterized instruction sequence by which the successive bits

---

[3] In the standard, basic operations and derived operations are called operations and functions, respectively.

in a 32-bit word that are the content of 32 successive Boolean registers become the content of 32 other successive Boolean registers:

$$SET(b_0 \ldots b_{31}, d{:}l) \triangleq {}^{\bullet}{}_{\displaystyle ,}{}^{31}_{i=0}(d{:}l+i.\mathsf{set}{:}b_i) \ ,$$

$$MOV(s{:}k, d{:}l) \triangleq {}^{\bullet}{}_{\displaystyle ,}{}^{31}_{i=0}(d{:}l+i.\mathsf{set}{:}0 \,;\, +s{:}k+i.\mathsf{get} \,;\, d{:}l+i.\mathsf{set}{:}1) \ ,$$

where $b_0, \ldots, b_{31}$ range over $\{0, 1\}$, s ranges over $\{\mathsf{in}, \mathsf{aux}\}$, d ranges over $\{\mathsf{aux}, \mathsf{out}\}$, and k, l range over $\{n \in \mathbb{N} \mid n \bmod 32 = 1\}$.

Moreover, we use the abbreviation

$$\mathsf{CONC\ FOR}\ i = l\ \mathsf{TO}\ l' : \{P_i\} \quad \text{for} \quad P_l \,;\, \ldots \,;\, P_{l'} \ ,$$

where $l, l' \in \mathbb{N}$ are such that $l < l'$, and $P_l, \ldots, P_{l'}$ are instruction sequences. We write $\mathsf{CONC\ FOR}$ instead of FOR to emphasize that we have to do here with an abbreviation for the concatenation of two or more instruction sequences.

The calculation of the lengths of the parameterized instruction sequences defined above is a matter of simple additions and multiplications. The lengths of the instruction sequences corresponding to the basic operations on bit strings relevant to SHA-256 are as follows:

$$\text{len}(NOT(s{:}k, d{:}l)) = 96 \ ,$$
$$\text{len}(AND(s_1{:}k_1, s_2{:}k_2, d{:}l)) = 192 \ ,$$
$$\text{len}(XOR(s_1{:}k_1, s_2{:}k_2, d{:}l)) = 288 \ ,$$
$$\text{len}(SHR^n(s{:}k, d{:}l)) = 96 - 2 \cdot n \ ,$$
$$\text{len}(ROTR^n(s{:}k, d{:}l)) = 96 \ ,$$
$$\text{len}(ADD(s_1{:}k_1, s_2{:}k_2, d{:}l)) = 705 \ ;$$

the lengths of the instruction sequences corresponding to the derived operations on bit strings defined in the standard are as follows:

$$\text{len}(CH(s_1{:}k_1, s_2{:}k_2, s_3{:}k_3, d{:}l)) = 768 \ ,$$
$$\text{len}(MAJ(s_1{:}k_1, s_2{:}k_2, s_3{:}k_3, d{:}l)) = 1152 \ ,$$
$$\text{len}(\Sigma_0(s{:}k, d{:}l)) = 864 \ ,$$
$$\text{len}(\Sigma_1(s{:}k, d{:}l)) = 864 \ ,$$
$$\text{len}(\sigma_0(s{:}k, d{:}l)) = 858 \ ,$$
$$\text{len}(\sigma_1(s{:}k, d{:}l)) = 844 \ ;$$

and the lengths of the SET and MOV instruction sequences are as follows:

$$\text{len}(SET(b_0 \ldots b_{31}, d{:}l)) = 32 \ ,$$
$$\text{len}(MOV(s{:}k, d{:}l)) = 96 \ .$$

5 SHA-256 Hash Computation

In this section, we give the description of instruction sequences that define SHA-256 using the definitions given in Sections 3 and 4.

The padding of messages to a bit length that is a multiple of 512 is left out. It is assumed that messages are already padded. Thus, the bit length of a message is always a multiple of 512. Suppose that N is the bit length of a message divided by 512. Because the maximum bit length of a message is 2^{64}, we have that $1 \leq N \leq 2^{55}$.

We write $\mathcal{M}_N$, where $1 \leq N \leq 2^{55}$, for $\{0,1\}^{512 \cdot N}$, and we write $\mathcal{M}$ for $\bigcup \{\mathcal{M}_N \mid 1 \leq N \leq 2^{55}\}$. Moreover, we write $\mathcal{D}$ for $\{0,1\}^{256}$. SHA-256 is a function from $\mathcal{M}$ to $\mathcal{D}$. We write SHA-256$_N$ for the restriction of SHA-256 to $\mathcal{M}_N$. Clearly, SHA-256 is the unique function from $\mathcal{M}$ to $\mathcal{D}$ such that, for each N with $1 \leq N \leq 2^{55}$, for each $w \in \mathcal{M}_N$, SHA-256$(w) =$ SHA-256$_N(w)$.

In Table 1, an instruction sequence IS$_{\text{SHA-256}_N}$ is uniformly described for all N with $1 \leq N \leq 2^{55}$.

Claim. *For each N with $1 \leq N \leq 2^{55}$, the instruction sequence* IS$_{\text{SHA-256}_N}$ *computes the function* SHA-256$_N$.

Because SHA-256 is not formally defined in the standard, we cannot formally prove this claim. However, we follow the standard so precisely in the description of IS$_{\text{SHA-256}_N}$ that the claim is unlikely to be wrong unless the pseudo code from the standard should not be interpreted as to be expected.

An easy calculation leads to the following result.

Fact. *For each N with $1 \leq N \leq 2^{55}$, the length of the instruction sequence* IS$_{\text{SHA-256}_N}$ *is* $780152 \cdot N + 1025$.

The calculation is a matter of simple additions and multiplications, using the lengths of the parameterized instruction sequences defined in Section 4:

$$
\begin{aligned}
& 8 \cdot 32 + \\
& N \cdot (16 \cdot 96 + \\
& \quad\quad 48 \cdot (844 + 858 + 3 \cdot 705) + \\
& \quad\quad 8 \cdot 96 + \\
& \quad\quad 64 \cdot (864 + 768 + 32 + 4 \cdot 705 + \\
& \quad\quad\quad\quad 864 + 1152 + 705 + \\
& \quad\quad\quad\quad 3 \cdot 96 + 705 + 3 \cdot 96 + 705) + \\
& \quad\quad 8 \cdot (96 + 705)) + \\
& 8 \cdot 96 + \\
& 1 \\
& \quad = \\
& 780152 \cdot N + 1025 \ .
\end{aligned}
$$

Table 1. The instruction sequence $\text{IS}_{\text{SHA-256}_N}$

CONC FOR $j = 0$ TO 7 :
  {
    $SET(H_i^{(0)}, H_i)$
  } ;
CONC FOR $i = 1$ TO N :
  {
   CONC FOR $j = 0$ TO 15 :
    {
     $MOV(M_j^{(i)}, W_j)$
    } ;
   CONC FOR $j = 16$ TO 63 :
    {
     $\sigma_1(W_{j-2}, t_1)$; $\sigma_0(W_{j-15}, t_2)$;
     $ADD(t_1, W_{j-7}, t_3)$; $ADD(t_2, W_{j-16}, t_4)$; $ADD(t_3, t_4, W_j)$
    } ;
   $MOV(H_0, a)$; $MOV(H_1, b)$; $MOV(H_2, c)$; $MOV(H_3, d)$;
   $MOV(H_4, e)$; $MOV(H_5, f)$; $MOV(H_6, g)$; $MOV(H_7, h)$;
   CONC FOR $j = 0$ TO 63 :
    {
     $\Sigma_1(e, t_1)$; $CH(e, f, g, t_2)$; $SET(K_j, t_3)$;
     $ADD(t_1, h, t_4)$; $ADD(t_2, t_3, t_5)$; $ADD(t_5, W_j, t_6)$; $ADD(t_4, t_6, T_1)$;
     $\Sigma_0(a, t_1)$; $MAJ(a, b, c, t_2)$; $ADD(t_1, t_2, T_2)$;
     $MOV(g, h)$; $MOV(f, g)$; $MOV(e, f)$; $ADD(d, T_1, e)$;
     $MOV(c, d)$; $MOV(b, c)$; $MOV(a, b)$; $ADD(T_1, T_2, a)$
    } ;
   $MOV(H_0, t_1)$; $ADD(a, t_1, H_0)$; $MOV(H_1, t_1)$; $ADD(b, t_1, H_1)$;
   $MOV(H_2, t_1)$; $ADD(c, t_1, H_2)$; $MOV(H_3, t_1)$; $ADD(d, t_1, H_3)$;
   $MOV(H_4, t_1)$; $ADD(e, t_1, H_4)$; $MOV(H_5, t_1)$; $ADD(f, t_1, H_5)$;
   $MOV(H_6, t_1)$; $ADD(g, t_1, H_6)$; $MOV(H_7, t_1)$; $ADD(h, t_1, H_7)$
  } ;
CONC FOR $j = 0$ TO 7 :
  {
   $MOV(H_j, D_j)$
  } ;
!

The left-hand side of this equation is laid out in such a way that the structure of the description in Table 1 is clearly reflected.

Recall that the instruction sequence $\mathrm{IS}_{\text{SHA-256}_N}$ $(1 \leq N \leq 2^{55})$ contains only instructions to set and get the content of Boolean registers, forward jump instructions, and a termination instruction. It is shown in [6] that, in the case of instruction sequences of this kind, instruction sequence length is a computational complexity measure closely related to non-uniform time complexity. Notice that, if the message has the maximum bit length $(\pm 1.8 \cdot 10^{19})$, the length of the instruction sequence is $\pm 2.8 \cdot 10^{22}$.

The maximum number of input registers needed is 2^{64} and the number of output registers needed is 256. The number of auxiliary registers used is 2945. We expect that number of auxiliary registers used by instruction sequence is a computational complexity measure closely related to non-uniform space complexity. Notice that the number of auxiliary registers used here does not depend on the length of the message.

6   Concluding Remarks

We have described instruction sequences that compute the restrictions of the secure hash function SHA-256 to the bit strings of the different possible lengths by means of uniform terms from the algebraic theory of single-pass instruction sequences known as PGA. Thus, we have provided a mathematically precise alternative to the pseudo-code description of an algorithm that computes SHA-256 found in the standard.

In previous work that is carried out in the setting of PGA, the work always concerns rigorous investigation of theoretical issues thinking in terms of instruction sequences (see e.g. [4]). This may give the impression that PGA is only suitable for such work. The use of PGA in the work presented in this paper shows that it is more versatile. However, this work has also shown that scalability calls for extension of PGA to an instruction sequence calculus that includes among other things a variable binding generalized concatenation operator and a suitable definition mechanism.

It is shown in [6] that, in the case of instruction sequences of the kind that we have dealt with in this paper, instruction sequence length is a computational complexity measure closely related to non-uniform time complexity. An option for future work is investigating the possible role of this complexity measure in issues concerning the complexity of the different kinds of attack on secure hash functions like SHA-256.

Acknowledgements

References

1. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming 51(2), 125–156 (2002)
2. Bergstra, J.A., Middelburg, C.A.: Indirect jumps improve instruction sequence performance. Scientific Annals of Computer Science 22(2), 253–265 (2012)
3. Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. Acta Informatica 49(3), 139–172 (2012)
4. Bergstra, J.A., Middelburg, C.A.: Instruction Sequences for Computer Science, Atlantis Studies in Computing, vol. 2. Atlantis Press, Amsterdam (2012)
5. Bergstra, J.A., Middelburg, C.A.: On the expressiveness of single-pass instruction sequences. Theory of Computing Systems 50(2), 313–328 (2012)
6. Bergstra, J.A., Middelburg, C.A.: Instruction sequence based non-uniform complexity classes. `arXiv:1301.3297v2 [cs.CC]` (January 2013)
7. Bergstra, J.A., Ponse, A.: An instruction sequence semigroup with involutive anti-automorphisms. Scientific Annals of Computer Science 19, 57–92 (2009)
8. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. The Internet Society, IETF RFC 5246 (August 2008)
9. Diertens, B.: A toolset for PGA. Electronic Report PRG0302, Programming Research Group, University of Amsterdam (October 2003), available at `http://www.science.uva.nl/research/prog/publications.html#prg-ereports`
10. Gilbert, H., Handschuh, H.: Security analysis of SHA-256 and sisters. In: Matsui, M., Zuccherato, R. (eds.) SAC 2003. Lecture Notes in Computer Science, vol. 3006, pp. 175–193. Springer-Verlag (2004)
11. Kent, S., Seo, K.: Security Architecture for the Internet Protocol. The Internet Society, IETF RFC 4301 (December 2005)
12. Mendel, F., Nad, T., Schläffer, M.: Finding SHA-2 characteristics: Searching through a minefield of contradictions. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. Lecture Notes in Computer Science, vol. 7073, pp. 288–307. Springer-Verlag (2011)
13. Mendel, F., Nad, T., Schläffer, M.: Improving local collisions: New attacks on reduced SHA-256. In: Johansson, T., Nguyen, P. (eds.) EUROCRYPT 2013. Lecture Notes in Computer Science, vol. 7881, pp. 262–278. Springer-Verlag (2013)
14. Mendel, F., Pramstaller, N., Rechberger, C., Rijmen, V.: Analysis of step-reduced SHA-256. In: Robshaw, M.J.B. (ed.) FSE 2006. Lecture Notes in Computer Science, vol. 4047, pp. 126–143. Springer-Verlag (2006)
15. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf` (2008)
16. Nikolić, I., Biryukov, A.: Collisions for step-reduced SHA-256. In: Nyberg, K. (ed.) FSE 2008. Lecture Notes in Computer Science, vol. 5086, pp. 1–15. Springer-Verlag (2008)
17. Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. The Internet Society, IETF RFC 5751 (January 2010)
18. Sanadhya, S.K., Sarkar, P.: New collision attacks against up to 24-step SHA-256. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. Lecture Notes in Computer Science, vol. 5365, pp. 91–103. Springer-Verlag (2008)
19. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Transport Layer Protocol. The Internet Society, IETF RFC 4253 (January 2006)
20. Secure Hash Standard. National Institute of Standards and Technology, FIPS PUB 180-4 (March 2012)

A Definitions of the SHA-256 constants

$$K_0 \triangleq 01000010100010100010111110011000,$$
$$K_1 \triangleq 01110001001101110100100100010001,$$
$$K_2 \triangleq 10110101110000011111011111001111,$$
$$K_3 \triangleq 11101001101101011101101110100101,$$
$$K_4 \triangleq 00111001010101101100001001011011,$$
$$K_5 \triangleq 01011001111100010001000111110001,$$
$$K_6 \triangleq 10010010001111111000001010100100,$$
$$K_7 \triangleq 10101011000111000101111011010101,$$
$$K_8 \triangleq 11011000000001111010101010011000,$$
$$K_9 \triangleq 00010010100000110101101100000001,$$
$$K_{10} \triangleq 00100100001100011000010110111110,$$
$$K_{11} \triangleq 01010101000011000111110111000011,$$
$$K_{12} \triangleq 01110010101111100101101011101110100,$$
$$K_{13} \triangleq 10000000110111101011000111111110,$$
$$K_{14} \triangleq 10011011110111000000011010100111,$$
$$K_{15} \triangleq 11000001100110111110001011110100,$$
$$K_{16} \triangleq 11100100100110110110100111000001,$$
$$K_{17} \triangleq 11101111101111100100011110000110,$$
$$K_{18} \triangleq 00001111110000011001110111000110,$$
$$K_{19} \triangleq 00100100000011001010000111001100,$$
$$K_{20} \triangleq 00101101111010010010110001101111,$$
$$K_{21} \triangleq 01001010011101001000010010101010,$$
$$K_{22} \triangleq 01011100101100001010100111011100,$$
$$K_{23} \triangleq 01110110111110011000100011011010,$$
$$K_{24} \triangleq 10011000001111100101000101010010,$$
$$K_{25} \triangleq 10101000011000111000110011011011101,$$
$$K_{26} \triangleq 10110000000001100100111111001000,$$
$$K_{27} \triangleq 10111111010110010111111111000111,$$
$$K_{28} \triangleq 11000110111000000000101111110011,$$
$$K_{29} \triangleq 11010101101001111001000101000111,$$
$$K_{30} \triangleq 00000110110010100110001101010001,$$
$$K_{31} \triangleq 00010100000101001001010010110111,$$
$$K_{32} \triangleq 00100111101101110000101010000101,$$
$$K_{33} \triangleq 00101110000110110010000100111000,$$
$$K_{34} \triangleq 01001101001011000110110111111100,$$

$K_{35} \triangleq 01010011001110000000110100010011$,
$K_{36} \triangleq 01100101000010100111001101010100$,
$K_{37} \triangleq 01110110011010100000101010111011$,
$K_{38} \triangleq 10000001110000101100100100101110$,
$K_{39} \triangleq 10010010011100100010110010000101$,
$K_{40} \triangleq 10100010101111111101000101001$,
$K_{41} \triangleq 10101000000110100110011001001011$,
$K_{42} \triangleq 11000010010010111000101101110000$,
$K_{43} \triangleq 11000111011011000101000110100011$,
$K_{44} \triangleq 11010001100100101110100000011001$,
$K_{45} \triangleq 11010110100110010000011000100100$,
$K_{46} \triangleq 11110100000011100011010110000101$,
$K_{47} \triangleq 00010000011010101010000001110000$,
$K_{48} \triangleq 00011001101001001100000100010110$,
$K_{49} \triangleq 00011110001101110110110000001000$,
$K_{50} \triangleq 00100111010010000111011101001100$,
$K_{51} \triangleq 00110100101100001011110010110101$,
$K_{52} \triangleq 00111001000111000000110010110011$,
$K_{53} \triangleq 01001110110110001010101001001010$,
$K_{54} \triangleq 01011011100111001100101001001111$,
$K_{55} \triangleq 01101000001011100110111111110011$,
$K_{56} \triangleq 01110100100011111000001011101110$,
$K_{57} \triangleq 01111000101001010110001101101111$,
$K_{58} \triangleq 10000100110010000111100000010100$,
$K_{59} \triangleq 10001100110001110000001000001000$,
$K_{60} \triangleq 10010000101111011111111111111010$,
$K_{61} \triangleq 10100100010100001101100111101011$,
$K_{62} \triangleq 10111110111110011010001111110111$,
$K_{63} \triangleq 11000110011100010111100011110010$.

14