MASTERARBEIT | MASTER'S THESIS

Titel | Title

Secure Hash Algorithms

verfasst von | submitted by

Margaretha Stephan BEd

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Education (MEd)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 199 507 520 02
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Lehramt Sek (AB) Unterrichtsfach Englisch Unterrichtsfach Mathematik
Betreut von | Supervisor:	ao. Univ.-Prof. Mag. Dr. Peter Raith

1 Zusammenfassung der Arbeit (Abstract)

Mit steigender Relevanz der Datensicherheit in der Entwicklung des digitalen Raums, werden auch Prozesse wie Hashing immer wichtiger. Jahrzehnte nach dem ersten von NIST standardisierten Hashing Algorithmus wird Fortschritt im Bereich Datenintegrität und digitale Verschlüsselung immer schneller zu einem essenziellen Teil der mathematischen und technischen Forschung. Dennoch fallen Antworten auf die Fragen „Wozu Hashing?“ und „Welcher Algorithmus garantiert die gewünschte Sicherheit?“ auch ihn vielen Anwendergruppen oft schwer. Anhand einer ausführlichen Literaturrecherche bietet diese Arbeit daher einen Überblick über die von NIST standardisierten Hashing Algorithmen SHA-1, SHA-2 und SHA-3, sowie einen Überblick über deren Entwicklung durch die Betrachtung der Algorithmen MD2, MD4 und MD5, welche direkte Vorfahren der ersten beiden NIST Algorithmen SHA-1 und SHA-2 darstellen. Im Folgenden werden außerdem grundliegende allgemeine und kryptografische Eigenschaften besprochen, welche eine Hashfunktion ausmachen. Zudem werden die zu diesem Zeitpunkt bekannten effizientesten Attacken auf SHA-1 und deren Vorgänger demonstriert. Zuguterletzt schließt die Arbeit mit einigen Beispielen, berechnet mithilfe eigens erstellter Implementierungen der Algorithmen in Python, welche einen Vergleich der einzelnen Sicherheitslevel direkt illustrieren und einen Überblick über mögliche Verwendungszwecke bieten.

2 Abstract

With the significance of data security increasing, hashing has become a vital part of both the computer sciences and mathematical research. Even though the first secure hashing algorithm has been defined by NIST decades ago, research on the field of secure hashing algorithms is now more relevant than ever. However, the choice of which security level to use in your application can often be a hard one. Using a literature approach, this paper offers an overview of all three secure hash algorithm families, SHA-1, SHA-2, and SHA-3, defined by NIST so far along with an analysis of their security status, including the most efficient known attacks for SHA-1, an algorithm pronounced insecure. It also provides an introduction into the evolution of secure hashing algorithms, introducing MD2, MD4, and MD5, as SHA-1 and SHA-2's direct predecessors as well as an insight on relevant properties for defining hash functions. Some general properties and properties specific to cryptographic hash functions are discussed. Lastly, using implementations of all algorithms in Python, the paper presents examples for all secure hash algorithms alongside a short guide on which security level to use for different purposes.

Contents

3 Introduction to Hashing

Hashing is a term used in modern computer sciences. It describes the process of mapping a certain data item, usually numbers or strings of text, onto so-called hash values to use them more efficiently or more practically in certain applications. As hashing is mainly used in the computer sciences, transforming information into strings that fulfill certain criteria is the reason hash functions have been developed and are used very frequently. The properties that hash functions are usually required to fulfill are discussed in the following chapter. [15]

Hashing can be used to create a so-called hash table, a data structure which uses hash values as indexes for data entries. In these hash tables information can then be searched for and, therefore, added, deleted, or read, very quickly and effectively by searching for its hash value. This concept not only speeds up the process of interacting with stored information but also ideally minimizes the risk of identification errors between very similar pieces of data. This effect is the second important reason that hash functions are a significant field of study nowadays, as will be the focus of this paper, namely the cryptographic application of hashing. [15]

The efficient data access and data validation through hash functions make them incredibly useful in three different kinds of applications. Firstly, hashing is used for efficient data records, where large data sets should be easily searchable. Secondly, it is a method used in validating data, for example, if data integrity is an issue, and lastly, cryptographic hash functions are used in modern data security. These cryptographic functions will be the focus of this paper. [15]

4 Definitions

4.1 Terms and Notation

Bit	Any binary digit with value `0` or `1`.
Byte	A unit of eight bits.
MD	Merkle-Damgård describes a group of functions and algorithms developed based on the Merkle-Damgård construction.
SHA	Secure Hash Algorithm.

Since hashing is a computer-based algorithm, messages are often operated on bit-wise. In order to write and work more efficiently some terminology will be introduced here.

1. Binary representations are grouped into 4-bit units. Each such unit can be represented by a so-called *hex digit*. A hex digit is an element of the set $\{0, 1, ..., 9, a, ..., f\}$ For example "`0111`" may be represented by the hex digit "7" while the hex digit "b" represents "`1011`".

2. A *word* is an "x"-bit string. Most words in hashing consist of either 32 or 64 bits. Each word can be represented as hex digits by converting each 4-bit unit individually. For example the 32-bit word

   `0100 1101 0100 0001 0101 0100 0100 1000`

   in hex digit representation can be expressed as "`4d415448`".

3. Furthermore, every *integer*, such as the message length l can be expressed as a word or pair of words. Each integer i with $0 \leq i \leq 2^{32} - 1$ can be represented as a 32-bit word. Similarly, positive integers up to 2^{64} can be represented by a 64-bit word, and so on. For example, the integer $410 = 2^8 + 2^7 + 2^4 + 2^3 + 2^1 = 256 + 128 + 16 + 8 + 2$ can be represented by the word "`0000019a`".

4. Nowadays, such bit-strings, such as bytes and words, are commonly bound to the *big-endian* convention, which means that the most significant bit or byte is in the left-most position of the byte or word. This convention is also called *high-order first*. In contrast, in earlier days, many definitions used *low-order first*, or *little-endian* convention, meaning that the least significant bit or byte is stored in the left-most position. For example, MD4 and MD5 use little-endian byte order within their words. For consistency, this paper uses big-endian conventions in writing examples unless indicated differently.

5. Lastly, an equation of the form $x = x + x'$ signifies that a certain variable x is assigned a new value overwriting the current value with a value that is computed recursively from the previous one.

6. When individual bits or bit-strings, such as bytes, words, or hash values, are concatenated after one another we write $||$. So, for example $a||b$ for $a = 1$ and $b = 0$ would be `10`.

4.2 Parameters

A	State Array in the SHA-3 algorithm.
$a, b, ..., h$	Working variables that are w-bit words used in the computation of hash values.
a_i, b_i, c_i, d_i, e_i	Value of a working variable at round i of computing the hash.
b	Width of the permutation of *KECCAK-p*.
c	Capacity of the Sponge function.
d	Length of the SHA-3 output string.
f, g, h, i, p, q, r, s	Functions of individual algorithms.
$h_0, ..., h_4$	Buffer values.
$H^{(i)}$	The i^{th} hash value. $H^{(0)}$ is called the *initial* hash value, while $H^{(N)}$ is the *final* hash value, used for the final message digest.
$H_j^{(i)}$	The j^{th} word of the i^{th} hash value. $H_0^{(i)}$ is the left-most word of the i^{th} hash value.
i_r	Round index in SHA-3's iota algorithm.
k	Number of zeros appended to the message in Padding.
l	Length of the message.
l	Lane size of the SHA-3 state array.
M	Message to be hashed.
M'	A second message that potentially collides with the first message M in collision attacks.
$M^{(i)}$	Message block of size m bits at position i.
$M_j^{(i)}$	The j^{th} word of the i^{th} message block. $M_0^{(i)}$ is the left-most word of the i^{th} message block.
n	Number of blocks being shifted or rotated in operations on words.
n_r	Number of rounds in the *KECCAK-p* algorithm.
N	Number of blocks in a padded message.
N	Input of the Sponge function.
r	Parameter rate of the Sponge function.
S, Z	Strings of characters within the hashing process.
$T, T_1, ..., T_4$	Temporary w-bit words used while computing.
t	Index for temporary words T.
$T[t]$	Bit or Byte at the index t of a temporary word T.
t	Round integer in SHA-3 algorithm.
w	Number of bits in a word in the SHA-3 algorithm.
W_t	The t^{th} w-bit word of a message schedule.
$W_t[i]$	The i-th bit of the corresponding word.
$W_t[\pm i_1, ..., \pm i_l]$	The i_1-th,...,i_l-th bit of a word gets changed. + symbolizes a switch from 0 to 1 and - the reverse switch.
x, y, z	Arbitrary w-bit words.
x_i	Value of the i-th bit of x.
$x*$	Following value of x. Also: x_{i+1}
x^2	Pair of words. Also: $(x, x*)$

4.3 Operations

$\wedge$	Bit-wise AND.
$\vee$	Bit-wise OR (inclusive-OR).
$\oplus$	Bit-wise XOR (exclusive-OR).
$\neg$	Bit-wise complement.
$+$	Addition modulo 2^w.
Δ	Difference between two elements. The difference is calculated by integer modular subtraction difference of the two items, so $\Delta M = M' - M$
$\ll$	Left-shift. $x \ll n$ is attained by discarding the left-most n bits of the word x and padding the remaining word with n zero bits on the right.
$\gg$	Right-shift. $x \gg n$ is attained by discarding the right-most n bits of the word x and padding the remaining word with n zero bits on the left.
$ROTL^n(x)$	Rotate-left (circular left shift). Let x be a w-bit word and n be an integer with $0 \leq n \leq w$. Then $ROTL^n(x) = (x \ll n) \vee (x \gg w - n)$.
$ROTR^n(x)$	Rotate-right (circular right shift). Let x be a w-bit word and n be an integer with $0 \leq n \leq w$. Then $ROTR^n(x) = (x \gg n) \vee (x \ll w - n)$.
$SHR^n(x)$	Right Shift. Let x be a w-bit word and n be an integer with $0 \leq n \leq w$. Then $SHR^n(x) = x \gg n$.

4.4 Functions

4.4.1 MD4

MD4 uses the basic logical operators which will be used in nearly all subsequent algorithms discussed in later sections. Each of these functions uses $x, y,$ and z, three 32-bit words, as an input and in turn outputs a new 32-bit word. All operations on the words are computed bit-wise. [26]

$$\begin{aligned} f(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z) \\ g(x,y,z) &= (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \\ h(x,y,z) &= x \oplus y \oplus z \end{aligned}$$

4.4.2 MD5

MD5, like MD4, uses functions built from logical operation on the input of 32-bit words each. [27]

$$\begin{aligned} f(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z) \\ g(x,y,z) &= (x \wedge z) \vee (y \wedge \neg z) \\ h(x,y,z) &= x \oplus y \oplus z \\ i(x,y,z) &= y \oplus (x \vee \neg z) \end{aligned}$$

$$\begin{aligned} p(a,b,c,d,k,n,t) &= b + ((a + f(b,c,d) + h_k + T[t] \ll n) \\ q(a,b,c,d,k,n,t) &= b + ((a + g(b,c,d) + h_k + T[t] \ll n) \\ r(a,b,c,d,k,n,t) &= b + ((a + h(b,c,d) + h_k + T[t] \ll n) \\ s(a,b,c,d,k,n,t) &= b + ((a + i(b,c,d) + h_k + T[t] \ll n) \end{aligned}$$

4.4.3 SHA-1

SHA-1 uses a sequence of logical functions, f_t where $0 \leq t \leq 79$. Each function operates on three 32-bit words and outputs one 32-bit word. $f_t(x,y,z)$ is defined as follows: [1]

$$f_t(x,y,z) = \begin{cases} Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) & \text{when } 0 \leq t \leq 19, \\ Parity(x,y,z) = x \oplus y \oplus z & \text{when } 20 \leq t \leq 39, \\ Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & \text{when } 40 \leq t \leq 59, \\ Parity(x,y,z) = x \oplus y \oplus z & \text{when } 60 \leq t \leq 79. \end{cases}$$

4.4.4 SHA-2

SHA-2 functions are used in SHA-224 as well as SHA-256. This group consists of six logical functions, each operating on 32-bit words $x, y,$ and z, and each resulting in one new 32-bit word. [1]

$$\begin{aligned} Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma_0^{256}(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\ \Sigma_1^{256}(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\ \sigma_0^{256}(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\ \sigma_1^{256}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \end{aligned}$$

Another variant of the second secure hash algorithm, SHA-512, uses the same basic functions, with different shifts. Each of these functions, however, operates on 64-bit words $x, y,$ and z, and outputs one 64-bit word. [1]

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0^{512}(x) &= ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \\
\Sigma_1^{512}(x) &= ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \\
\sigma_0^{512}(x) &= ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \\
\sigma_1^{512}(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)
\end{aligned}
$$

4.5 Constants

Within their functions all algorithms also use certain constants in different representations which are usually computed from mathematical objects with certain properties. The constants for the algorithms discussed in this paper are defined in this section.

4.5.1 MD2

MD2 uses one constant in calculating the checksum as well as step 3 of the MD2-calculation process. This constant is a 256-byte "random" permutation constructed from the digits of pi.

$$\begin{aligned} S[p] = \{&41, 46, 67, 201, 162, 216, 124, 1, 61, 54, 84, 161, 236, 240, 6, 19,\\ &98, 167, 5, 243, 192, 199, 115, 140, 152, 147, 43, 217, 188, 76, 130, 202,\\ &30, 155, 87, 60, 253, 212, 224, 22, 103, 66, 111, 24, 138, 23, 229, 18,\\ &190, 78, 196, 214, 218, 158, 222, 73, 160, 251, 245, 142, 187, 47, 238, 122,\\ &169, 104, 121, 145, 21, 178, 7, 63, 148, 194, 16, 137, 11, 34, 95, 33,\\ &128, 127, 93, 154, 90, 144, 50, 39, 53, 62, 204, 231, 191, 247, 151, 3,\\ &255, 25, 48, 179, 72, 165, 181, 209, 215, 94, 146, 42, 172, 86, 170, 198,\\ &79, 184, 56, 210, 150, 164, 125, 182, 118, 252, 107, 226, 156, 116, 4, 241,\\ &69, 157, 112, 89, 100, 113, 135, 32, 134, 91, 207, 101, 230, 45, 168, 2,\\ &27, 96, 37, 173, 174, 176, 185, 246, 28, 70, 97, 105, 52, 64, 126, 15,\\ &85, 71, 163, 35, 221, 81, 175, 58, 195, 92, 249, 206, 186, 197, 234, 38,\\ &44, 83, 13, 110, 133, 40, 132, 9, 211, 223, 205, 244, 65, 129, 77, 82,\\ &106, 220, 55, 200, 108, 193, 171, 250, 36, 225, 123, 8, 12, 189, 177, 74,\\ &120, 136, 149, 139, 227, 99, 232, 109, 233, 203, 213, 254, 59, 0, 29, 57,\\ &242, 239, 183, 14, 102, 88, 208, 228, 166, 119, 114, 248, 235, 117, 75, 10,\\ &49, 68, 80, 180, 143, 237, 31, 26, 219, 153, 141, 51, 159, 17, 131, 20\} \end{aligned}$$

4.5.2 MD4

MD4 uses two constants, one for round 2 and one for round 3. These constants are computed by calculating parts of the square root of 2 and 3, which are each used for their respective rounds. [26]

$$\sqrt{2} : 013240474631$$
$$\sqrt{3} : 015666365641$$

However, these constants are used in hex representation with high-order digits in the leftmost positions, which signifies a change in MD4 convention. [26]

$$Round\,2 : \sqrt{2} = \texttt{5a827999}$$
$$Round\,3 : \sqrt{3} = \texttt{6ed9eba1}$$

For the extension of MD4 other constants are used in hex representation with high-order digits in the leftmost position. [26]

	Octal	*Hex*
$Round\,2$:	$\sqrt[3]{2} = 012050505746$	$\sqrt[3]{2} = \texttt{50a28be6}$
$Round\,3$:	$\sqrt[3]{3} = 013423350444$	$\sqrt[3]{3} = \texttt{5c4dd124}$

4.5.3 MD5

As a constant for each of the four rounds of MD5 the algorithm uses a table of 64 elements computed in the following way:
For $i = 0$ to 64:

$$4294967296 * |sin(i)|$$

Of these results, only the "integer" part, which is achieved by rounding down the resulting number to an integer, is saved into the list. The resulting table in hex representation looks like this. [27]

```
[d76aa478  e8c7b756  242070db  c1bdceee
 f57c0faf  4787c62a  a8304613  fd469501
 698098d8  8b44f7af  ffff5bb1  895cd7be
 6b901122  fd987193  a679438e  49b40821
 f61e2562  c040b340  265e5a51  e9b6c7aa
 d62f105d  02441453  d8a1e681  e7d3fbc8
 21e1cde6  c33707d6  f4d50d87  455a14ed
 a9e3e905  fcefa3f8  676f02d9  8d2a4c8a
 fffa3942  8771f681  6d9d6122  fde5380c
 a4beea44  4bdecfa9  f6bb4b60  bebfbc70
 289b7ec6  eaa127fa  d4ef3085  04881d05
 d9d4d039  e6db99e5  1fa27cf8  c4ac5665
 f4292244  432aff97  ab9423a7  fc93a039
 655b59c3  8f0ccc92  ffeff47d  85845dd1
 6fa87e4f  fe2ce6e0  a3014314  4e0811a1
f7537e82  bd3af235  2ad7d2bb  eb86d391]
```

4.5.4 SHA-1

For SHA-1 constant 32-bit words $K_0, K_1, ..., K_{79}$ are given such that

$$K_t = \begin{cases} \texttt{5a827999} & \text{when } 0 \leq t \leq 19, \\ \texttt{6ed9eba1} & \text{when } 20 \leq t \leq 39, \\ \texttt{8f1bbcdc} & \text{when } 40 \leq t \leq 59, \\ \texttt{ca62c1d6} & \text{when } 60 \leq t \leq 79. \end{cases}$$

[1]

4.5.5 SHA-256

For both SHA-256 and SHA-224, sixty-four constants $K_0^{\{256\}}, K_1^{\{256\}}, ..., K_{63}^{\{256\}}$ are required. These sixty-four words are 32-bit representations of the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex representation

$K_0^{\{256\}}, K_1^{\{256\}}, ..., K_{63}^{\{256\}}$ are (from left to right) [1]

```
428a2f98  71374491  b5c0fbcf  e9b5dba5
3956c25b  59f111f1  923f82a4  ab1c5ed5
d807aa98  12835b01  243185be  550c7dc3
72be5d74  80deb1fe  9bdc06a7  c19bf174
e49b69c1  efbe4786  0fc19dc6  240ca1cc
2de92c6f  4a7484aa  5cb0a9dc  76f988da
983e5152  a831c66d  b00327c8  bf597fc7
c6e00bf3  d5a79147  06ca6351  14292967
27b70a85  2e1b2138  4d2c6dfc  53380d13
650a7354  766a0abb  81c2c92e  92722c85
a2bfe8a1  a81a664b  c24b8b70  c76c51a3
d192e819  d6990624  f40e3585  106aa070
19a4c116  1e376c08  2748774c  34b0bcb5
391c0cb3  4ed8aa4a  5b9cca4f  682e6ff3
748f82ee  78a5636f  84c87814  8cc70208
90befffa  a4506ceb  bef9a3f7  c67178f2
```

4.5.6 SHA-512

Similarly to its smaller varieties, SHA-512 also uses the constants $K_0^{\{512\}}, K_1^{\{512\}}, ..., K_{79}^{\{512\}}$. All eighty constants are 64-bit words, representing the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. In hex representation

$K_0^{\{512\}}, K_1^{\{512\}}, ..., K_{79}^{\{512\}}$ are (from left to right) [1]

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

5 Properties of Hash Functions

As previously mentioned, hash functions are used for several different purposes. In order to fulfill these purposes, common modern hash functions are usually required to have a multitude of special properties. Even though required properties differ significantly across certain areas, some basic criteria are used to ascertain whether a hash function can be functional for any kind of application. These very general properties, which every hash function's value is based on, are "Uniformity", "Deterministic Behavior", and "Efficiency".

5.1 General Properties

5.1.1 Deterministic Behavior

The most basic property that a hash function has to fulfill to be considered a proper hash function is that it needs to be a deterministic function, which specifically means that the mapping algorithm must be mathematically a function. Therefore, we define that for every hashing algorithm given the same input x twice, the result $H(x)$ has to be identical. $x = x \mapsto H(x) = H(x)$ [28]

5.1.2 Uniformity

A second property every hash function requires to be considered effective is that its output hashes need to be as uniformly distributed as possible. This property is required of every hash function, as clusters of hash outputs resulting from the function generating certain hashes with higher probability can negatively influence various issues, such as security or time efficiency in searching hash tables. [28]

5.1.3 Efficiency

Naturally, every system or application strives to work as efficiently as possible. In the field of hash functions efficiency can be discussed separately in two different aspects: time efficiency and space requirements. These factors indirectly depend on one another, which will be discussed in detail in the following chapters. In cryptographic applications, such as the ones that are the focus of this paper, time efficiency in computing the hash as well as memory requirements for certain computational steps and values sometimes have to give way to some extent to security requirements. [19]

5.2 Cryptographic Properties

Additionally to these practical properties, cryptographic hash functions, such as the secure hash algorithms, need to fulfill certain security properties. These properties are required to ensure that encryptions using these hashing algorithms are secure from cryptographic attacks. This means that essentially the task of a cryptographic hash algorithm is for the hash value computed to represent the message that was hashed without revealing the message itself. In order to do so the function is judged on the following three properties. It should be noted that if it can be shown that a hash function lacks at least one of these properties, it is considered insecure. Most older hash functions have been shown to have some flaws fulfilling at least one of the properties which will be discussed and proven in the following chapters.

5.2.1 Pre-image Resistance

The first, most basic, security property any hash function has to fulfill is pre-image resistance. This means that given a certain hash value, meaning the output hash of a computation, no information about the message that has been hashed is revealed. The hash function has to be "hard to invert", which means that it is computationally infeasible, given a certain hash value, to find the corresponding input message, or given H^N, finding x such that $H(x) = H^N$ is a hard problem. [19]

5.2.2 Second Pre-image Resistance

Another, rather similar, property that is required is second pre-image resistance. Similarly to finding a message from the output, given any message, it should also not be possible to find a second message which produces the same message hash as the one given. Again, it has to be computationally infeasible, given a message x and its hash $H(x)$ to find another message y, different from x such that $H(x) = H(y)$. [19]

5.2.3 Collision Resistance

Lastly, cryptographic hash functions are required to be collision-resistant. This means that finding any two distinct messages x and y such that $H(x) = H(y)$ is a hard problem, or more concretely computationally infeasible. [19]

It has to be noted at this point that a complete collision resistance is not possible. This is due to the fact that a hash function takes any input of variable length and maps it onto a hash value of fixed length. Therefore, collisions exist for every hash function, since we have an infinite number of messages, but a finite number of possible hash values. However, since these collisions should be distributed evenly among the possibilities, it is possible to create hash functions where finding a collision is a hard problem. Usually when testing a hash function for these properties, hash functions are considered secure if there are no collision attacks with complexity below $2^{n/2}$. Similarly, pre-image and second pre-image attacks are usually required to be of complexity below 2^n for the function to be considered insecure. This is due to the fact that a birthday attack, which will be explained in more detail later, can be used to find collisions for any hash function with complexity around $2^{n/2}$, while brute force searches can statistically find pre-images and second pre-images for every hash function with complexity of about 2^n. [18]

6 Evolution of Modern Hash Functions

This chapter will describe hashing algorithms that are considered insecure which will be proven to be vulnerable to potential attacks in the following chapter. The first algorithm discussed is MD2. MD2 is an algorithm using the Merkle-Damgård construction which is the construction used in the first two officially standardized secure hash algorithms SHA-1 and SHA-2. MD2 can therefore be considered the first descendant of modern secure hash algorithms. The following sections will describe MD2 followed by its descendants MD4 and MD5, eventually leading to SHA-1. [17] [1]

6.1 MD2

The following definition of MD2 follows [17].

The first step of computing MD2, as for other hash algorithms, is padding the message. The message is padded to be a multiple of 16 bytes in length. This is achieved by adding the required number of bytes. Suppose the message is of length l. Then k bytes of value k are appended to the message so that $l+k$ is a multiple of 16. If the length is already a multiple of 16 bytes, 16 more bytes are appended.

After padding the message, a 16-byte checksum of the message is also appended. In this step, the MD2-constant is used. Let $S[p]$ be the $[p]^{th}$ element of this table. The checksum is calculated and appended as follows.

1. The working variables $c_1, c_2, ..., c_{15}$ as well as T are initialized and each set to 0.
2. Each 16-word block $M^{(i)}$ gets processed individually in the following way.
   For $i = 1$ to N: {
   (a) For $t = 0$ to 15: {
      i. c_t is calculated by $c_t = S[M_t^{(i)} \oplus T]$
      ii. T is set to c_t before repeating the step for c_{t+1}
   }
   }
3. After repeating step two for every block the resulting 16-byte checksum is appended to the message.

Lastly, in order to compute the MD2-hash, a 48-byte buffer $H = h_0, h_1, ..., h_{47}$, which is initialized as 0 and the MD2-constant are required. Each 16-byte block $M^{(i)}$ is again processed individually.

For $i = 0$ to N:

{

1. The working variables $h_0, h_1, ..., h_{47}$ are computed.

   For $t = 0$ to 15:
   {

(a) $h_{t+16} = M_t^{(i)}$

(b) $h_{t+32} = h_{t+16} \oplus h_t$

}

2. The working variable p is initialized as 0.

3. This step is repeated 18 times.

For $j = 0$ to 17:
{

For $t = 0$ to 47:
{

(a) h_t is computed by $h_t^{(j)} = h_t^{(j-1)} \oplus S[p]$.

(b) p is set to h_t.

}
p is set to $(p + j)$ mod 256.
}

}

Finally, the message digest is

$$h_0 \| h_1 \| h_2 \| h_3 \| h_4 \| h_5 \| h_6 \| h_7 \| h_8 \| h_9 \| h_{10} \| h_{11} \| h_{12} \| h_{13} \| h_{14} \| h_{15}.$$

6.2 MD4

MD4, the second hashing algorithm using Merkle-Damgård, was created by Ronald Rivest in 1990. It was proven to be insecure a few years later and is no longer used. However, aspects of the algorithm have been used in later algorithms of the MD-, RIPEMD-, and SHA-family. [31]

Before defining the MD4 algorithm, it should be noted that, unlike common modern convention, MD4 uses the little-endian convention, which means that within each word the lowest-order byte comes first. The definition of MD4, including functions and constants in the previous chapters, follows the original Rivest definition in [26].

As common in all hashing algorithms, the first step is to pad the initial message. Suppose that a message M is of length l. Every message M is padded in three steps. First, the bit "1" is appended to the message, followed by several zero bits. This part of the padding adds k bits of zero, where k is the smallest non-negative solution to the equation $l+1+k \equiv 448 \bmod 512$. Lastly, the number l in binary representation is appended as a 64-bit block written as two 32-bit words. For this instance, it is important to note that not only are the words in little-endian convention, but the two words used to represent l are also switched. In the case of the representation of the message length being longer than sixty-four bits, only the leftmost sixty-four would be appended.
As a second step, MD4 initializes buffers. These buffers are initialized in hex representation, again low-order bit first, as follows:

$$a = \texttt{01 23 45 67}$$
$$b = \texttt{89 ab cd ef}$$
$$c = \texttt{fe dc ba 98}$$
$$d = \texttt{76 54 32 10}$$

It should be noted at this point that nowadays big-endian convention is usually preferred and that MD4 is now commonly defined to use these buffers high-order first, which are the following. [31]

$$a = \texttt{67 45 23 01}$$
$$b = \texttt{ef cd ab 89}$$
$$c = \texttt{98 ba dc fe}$$
$$d = \texttt{10 32 54 76}$$

Finally, each 512-bit block $M^{(0)}, M^{(1)}, ..., M^{(N)}$ is parsed into 32-word blocks $M_0^{(i)}, M_1^{(i)}, ..., M_{15}^{(i)}$ which are each processed individually in the following way.

For $i = 0$ to N:
{

1. Save a as T_1, b as T_2, c as T_3, and d as T_4.

2. Do the following operations:

(a) Round 1.

$$
\begin{aligned}
a &= (a + f(b,c,d) + M_0^{(i)}) \lll 3 \\
d &= (d + f(a,b,c) + M_1^{(i)}) \lll 7 \\
c &= (c + f(d,a,b) + M_2^{(i)}) \lll 11 \\
b &= (b + f(c,d,a) + M_3^{(i)}) \lll 19 \\
a &= (a + f(b,c,d) + M_4^{(i)}) \lll 3 \\
d &= (d + f(a,b,c) + M_5^{(i)}) \lll 7 \\
c &= (c + f(d,a,b) + M_6^{(i)}) \lll 11 \\
b &= (b + f(c,d,a) + M_7^{(i)}) \lll 19 \\
a &= (a + f(b,c,d) + M_8^{(i)}) \lll 3 \\
d &= (d + f(a,b,c) + M_9^{(i)}) \lll 7 \\
c &= (c + f(d,a,b) + M_{10}^{(i)}) \lll 11 \\
b &= (b + f(c,d,a) + M_{11}^{(i)}) \lll 19 \\
a &= (a + f(b,c,d) + M_{12}^{(i)}) \lll 3 \\
d &= (d + f(a,b,c) + M_{13}^{(i)}) \lll 7 \\
c &= (c + f(d,a,b) + M_{14}^{(i)}) \lll 11 \\
b &= (b + f(c,d,a) + M_{15}^{(i)}) \lll 19
\end{aligned}
$$

(b) Round 2.

$$
\begin{aligned}
a &= (a + g(b,c,d) + M_0^{(i)} + \texttt{5A827999}) \lll 3 \\
d &= (d + g(a,b,c) + M_4^{(i)} + \texttt{5A827999}) \lll 5 \\
c &= (c + g(d,a,b) + M_8^{(i)} + \texttt{5A827999}) \lll 9 \\
b &= (b + g(c,d,a) + M_{12}^{(i)} + \texttt{5A827999}) \lll 13 \\
a &= (a + g(b,c,d) + M_1^{(i)} + \texttt{5A827999}) \lll 3 \\
d &= (d + g(a,b,c) + M_5^{(i)} + \texttt{5A827999}) \lll 5 \\
c &= (c + g(d,a,b) + M_9^{(i)} + \texttt{5A827999}) \lll 9 \\
b &= (b + g(c,d,a) + M_{13}^{(i)} + \texttt{5A827999}) \lll 13 \\
a &= (a + g(b,c,d) + M_2^{(i)} + \texttt{5A827999}) \lll 3 \\
d &= (d + g(a,b,c) + M_6^{(i)} + \texttt{5A827999}) \lll 5 \\
c &= (c + g(d,a,b) + M_{10}^{(i)} + \texttt{5A827999}) \lll 9 \\
b &= (b + g(c,d,a) + M_{14}^{(i)} + \texttt{5A827999}) \lll 13 \\
a &= (a + g(b,c,d) + M_3^{(i)} + \texttt{5A827999}) \lll 3 \\
d &= (d + g(a,b,c) + M_7^{(i)} + \texttt{5A827999}) \lll 5 \\
c &= (c + g(d,a,b) + M_{11}^{(i)} + \texttt{5A827999}) \lll 9 \\
b &= (b + g(c,d,a) + M_{15}^{(i)} + \texttt{5A827999}) \lll 13
\end{aligned}
$$

(c) Round 3.

$$
\begin{aligned}
a &= (a + h(b,c,d) + M_0^{(i)} + \texttt{6ED9EBA1}) \lll 3 \\
d &= (d + h(a,b,c) + M_8^{(i)} + \texttt{6ED9EBA1}) \lll 9 \\
c &= (c + h(d,a,b) + M_4^{(i)} + \texttt{6ED9EBA1}) \lll 11 \\
b &= (b + h(c,d,a) + M_{12}^{(i)} + \texttt{6ED9EBA1}) \lll 15 \\
a &= (a + h(b,c,d) + M_2^{(i)} + \texttt{6ED9EBA1}) \lll 3 \\
d &= (d + h(a,b,c) + M_{10}^{(i)} + \texttt{6ED9EBA1}) \lll 9 \\
c &= (c + h(d,a,b) + M_6^{(i)} + \texttt{6ED9EBA1}) \lll 11 \\
b &= (b + h(c,d,a) + M_{14}^{(i)} + \texttt{6ED9EBA1}) \lll 15 \\
a &= (a + h(b,c,d) + M_1^{(i)} + \texttt{6ED9EBA1}) \lll 3 \\
d &= (d + h(a,b,c) + M_9^{(i)} + \texttt{6ED9EBA1}) \lll 9 \\
c &= (c + h(d,a,b) + M_5^{(i)} + \texttt{6ED9EBA1}) \lll 11 \\
b &= (b + h(c,d,a) + M_{13}^{(i)} + \texttt{6ED9EBA1}) \lll 15 \\
a &= (a + h(b,c,d) + M_3^{(i)} + \texttt{6ED9EBA1}) \lll 3 \\
d &= (d + h(a,b,c) + M_{11}^{(i)} + \texttt{6ED9EBA1}) \lll 9 \\
c &= (c + h(d,a,b) + M_7^{(i)} + \texttt{6ED9EBA1}) \lll 11 \\
b &= (b + h(c,d,a) + M_{15}^{(i)} + \texttt{6ED9EBA1}) \lll 15
\end{aligned}
$$

3. Lastly, add the following.

$$
\begin{aligned}
a &= a + T_1 \\
b &= b + T_2 \\
c &= c + T_3 \\
d &= d + T_4
\end{aligned}
$$

}

The message digest of 128 bits is

$$a \| b \| c \| d.$$

There is also an extension to MD4 creating a 256-bit output instead. To compute this version, the same algorithm is used, once as described above and a second time using the alternate constants

$$
\begin{aligned}
a' &: \texttt{00112233} \\
b' &: \texttt{44556677} \\
c' &: \texttt{8899aabb} \\
d' &: \texttt{ccddeeff}
\end{aligned}
$$

as well as changing $\sqrt{2}$ and $\sqrt{3}$ to $\sqrt[3]{2}$ and $\sqrt[3]{3}$. The 256-bit message digest is

$$a' \| b \| c \| d \| a \| b' \| c' \| d'.$$

6.3 MD5

MD5 was presented by Ronald Rivest in 1992 as an attempt to provide a strengthened version of the previous MD4. Several attacks have been found since then, the first in 1993 and nowadays it is not considered safe for security purposes. One of the newer, efficient proofs of its vulnerabilities will be shown in the chapter following this. [32]

The following definition of the algorithm as well as definitions of MD5 functions and constants follow the current RFC as described in [27].

As a first step, MD5 uses the same padding algorithm as MD4, its predecessor. Similar to MD4, it also uses low-order first byte-order for its words. However, when padding the message with the final 64-bit representation of l the two words are not switched. Afterwards, the MD buffer is initialized. Like the padding algorithm, MD5 also uses the same initial hash value as MD4, which is shown here in both, the original little-endian hex representation and the now more commonly used big-endian convention.

$$\begin{aligned} a &= \texttt{01 23 45 67} & a &= \texttt{67 45 23 01} \\ b &= \texttt{89 ab cd ef} & b &= \texttt{ef cd ab 89} \\ c &= \texttt{fe dc ba 98} & c &= \texttt{98 ba dc fe} \\ d &= \texttt{76 54 32 10} & d &= \texttt{10 32 54 76} \end{aligned}$$

For computing the MD5 algorithm, the message gets parsed into message blocks $M^{(0)}, M^{(1)}, ..., M^{(N)}$, each consisting of sixteen 32-bit words, $M_0^{(i)}, M_1^{(i)}, ..., M_{15}^{(i)}$. Each block is processed after another.

For $i = 0$ to N:
{

1. Each word is copied into the buffer.
   For $j = 0$ to 15 :{
   $h_j = M_j^{(i)}$ }
2. Save a as T_1, b as T_2, c as T_3, and d as T_4.
3. The following are computed.

$$\begin{aligned} a &= p(a, b, c, d, 0, 7, 1) \\ d &= p(d, a, b, c, 1, 12, 2) \\ c &= p(c, d, a, b, 2, 17, 3) \\ b &= p(b, c, d, a, 3, 22, 4) \\ a &= p(a, b, c, d, 4, 7, 5) \\ d &= p(d, a, b, c, 5, 12, 6) \\ c &= p(c, d, a, b, 6, 17, 7) \\ b &= p(b, c, d, a, 7, 22, 8) \\ a &= p(a, b, c, d, 8, 7, 9) \\ d &= p(d, a, b, c, 9, 12, 10) \\ c &= p(c, d, a, b, 10, 17, 11) \\ b &= p(b, c, d, a, 11, 22, 12) \\ a &= p(a, b, c, d, 12, 7, 13) \\ d &= p(d, a, b, c, 13, 12, 14) \\ c &= p(c, d, a, b, 14, 17, 15) \\ b &= p(b, c, d, a, 15, 22, 16) \end{aligned}$$

4. The following are computed.

$$\begin{aligned}
a &= q(a,b,c,d,1,5,17)\\
d &= q(d,a,b,c,6,9,18)\\
c &= q(c,d,a,b,11,14,19)\\
b &= q(b,c,d,a,0,20,20)\\
a &= q(a,b,c,d,5,5,21)\\
d &= q(d,a,b,c,10,9,22)\\
c &= q(c,d,a,b,15,14,23)\\
b &= q(b,c,d,a,4,20,24)\\
a &= q(a,b,c,d,9,5,25)\\
d &= q(d,a,b,c,14,9,26)\\
c &= q(c,d,a,b,3,14,27)\\
b &= q(b,c,d,a,8,20,28)\\
a &= q(a,b,c,d,13,5,29)\\
d &= q(d,a,b,c,2,9,30)\\
c &= q(c,d,a,b,7,14,31)\\
b &= q(b,c,d,a,12,20,32)
\end{aligned}$$

5. The following are computed.

$$\begin{aligned}
a &= r(a,b,c,d,5,4,33)\\
d &= r(d,a,b,c,8,11,34)\\
c &= r(c,d,a,b,11,16,35)\\
b &= r(b,c,d,a,14,23,36)\\
a &= r(a,b,c,d,1,4,37)\\
d &= r(d,a,b,c,4,11,38)\\
c &= r(c,d,a,b,7,16,39)\\
b &= r(b,c,d,a,10,23,40)\\
a &= r(a,b,c,d,13,4,41)\\
d &= r(d,a,b,c,0,11,42)\\
c &= r(c,d,a,b,3,16,43)\\
b &= r(b,c,d,a,6,23,44)\\
a &= r(a,b,c,d,9,4,45)\\
d &= r(d,a,b,c,0,11,46)\\
c &= r(c,d,a,b,15,16,47)\\
b &= r(b,c,d,a,2,23,48)
\end{aligned}$$

6. The following are computed.

$$\begin{aligned}
a &= s(a, b, c, d, 0, 6, 49) \\
d &= s(d, a, b, c, 7, 10, 50) \\
c &= s(c, d, a, b, 14, 15, 51) \\
b &= s(b, c, d, a, 5, 21, 52) \\
a &= s(a, b, c, d, 12, 6, 53) \\
d &= s(d, a, b, c, 3, 10, 54) \\
c &= s(c, d, a, b, 10, 15, 55) \\
b &= s(b, c, d, a, 1, 21, 56) \\
a &= s(a, b, c, d, 8, 6, 57) \\
d &= s(d, a, b, c, 15, 10, 58) \\
c &= s(c, d, a, b, 6, 15, 59) \\
b &= s(b, c, d, a, 13, 21, 60) \\
a &= s(a, b, c, d, 4, 6, 61) \\
d &= s(d, a, b, c, 11, 10, 62) \\
c &= s(c, d, a, b, 2, 15, 63) \\
b &= s(b, c, d, a, 9, 21, 64)
\end{aligned}$$

7. All four variables are incremented by their original values.

$$\begin{aligned}
a &= a + T_1 \\
b &= b + T_2 \\
c &= c + T_3 \\
d &= d + T_4
\end{aligned}$$

}

The message digest, in low-order first, is

$$a \| b \| c \| d.$$

6.4 SHA-1

Another family of hash functions that has been developed from the MD family is the Secure Hash Algorithms or SHA for short. Hash functions of this family make up most of the security hash functions used nowadays, which will be discussed in the next chapters. However, the first member of the SHA family, SHA-1, is not considered secure anymore. It will therefore be discussed here rather than in chapter six alongside its current relatives. It should be noted that SHA-1 had a predecessor, SHA-0. However, since these two functions were extremely similar and SHA-0 was only changed in one aspect that was shown to be insecure, nowadays there is often no distinction made between SHA-0 and SHA-1. [1]

The description of the SHA-1 in this section follows the specification in [1].

SHA-1 is able to hash messages M of length l where $0 \leq l \leq 2^{64}$, splitting it into eighty 32-bit words $W_0, W_1, ..., W_{79}$. The result of this algorithm is always a 160-bit message digest. SHA-1 also uses five working variables labeled a, b, c, d and e as well as five words representing hash values $H_0^{(i)}, H_1^{(i)}, ..., H_5^{(i)}$ and a single temporary word T.

Similar to many other hash functions SHA-1 requires padding to create a message with a length that is a multiple of 512 bit. SHA-1 uses the same padding algorithm as the MD algorithms previously discussed. However, since it is common nowadays to use the big-endian convention, the message length being appended in the end would appear with its most significant bit in the left-most position. Even though many modern hash functions require multiples of 1024 bits, the padding principle of SHA-1 is still unchanged and in use today. The padding algorithm for SHA-1 is described as before.

Messages longer than 512 bits are then parsed into blocks of 512 bits $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Furthermore, each block is split into sixteen 32-bit words which are denoted as partial blocks $M_0^{(i)}, M_1^{(i)}, ..., M_{15}^{(i)}$, where $1 \leq i \leq N$.

As in previous sections SHA-1 starts by setting the initial hash value $H^{(0)}$. $H^{(0)}$ consists of the following five 32-bit words in hex.

$$
\begin{aligned}
H_0^{(0)} &= \texttt{67452301} \\
H_1^{(0)} &= \texttt{efcdab89} \\
H_2^{(0)} &= \texttt{98badcfe} \\
H_3^{(0)} &= \texttt{10325476} \\
H_4^{(0)} &= \texttt{c3d2e1f0}
\end{aligned}
$$

Calculating the SHA-1 hash value for a message M consists of the following four steps that are repeated cyclically once per 512-bit block, creating a total of N repetitions for the blocks $M^{(0)}, M^{(1)}, ..., M^{(N)}$. It should be noted at this point that the only aspect SHA-0 differed in is missing the left-rotate in this definition.

For $i = 1$ to N:
{

1. The so-called message schedule is prepared by feeding message blocks into their individual words, as well as appending M to a total of 80 words $W_0, W_1, ..., W_{79}$.

$$W_t = \begin{cases} M_t^{(i)} & \text{when } 0 \le t \le 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & \text{when } 16 \le t \le 79 \end{cases}$$

2. The five working variables $a, b, c, d,$ and e are initialized with the $(i-1)^{\text{st}}$ hash value.

$$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \end{aligned}$$

3. For $t = 0$ to 79:
   {
$$\begin{aligned} T &= ROTL^5(a) + f_t(b, c, d) + e + K_t + W_t \\ e &= d \\ d &= c \\ c &= ROTL^{30}(b) \\ b &= a \\ a &= T \end{aligned}$$
   }

4. The intermediate hash value $H^{(1)}$ is computed.

$$\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} \\ H_2^{(i)} &= c + H_2^{(i-1)} \\ H_3^{(i)} &= d + H_3^{(i-1)} \\ H_4^{(i)} &= e + H_4^{(i-1)} \end{aligned}$$

}

After these four steps have been repeated N times the final 160-bit message digest is given by

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)}.$$

7 Attacks on Insecure Hashes

As previously mentioned the algorithms defined in the previous chapter have all been proven to be insecure in the last years. In the following section, proofs are discussed which show vulnerability to attacks by all algorithms defined so far.

Generally speaking, there are commonly two types of attacks that are used in breaking hash functions, collision attacks and preimage attacks. While collision attacks attempt to find collisions of different inputs colliding in the same output hash, the pre-image attack, both in its standard form and the second variant, the second pre-image attack, aims to directly find the input given only the output hash. Formally, the different types are defined as follows. [18]

1. **Collision:** Find m and m' such that $m \neq m'$ and $H(m) = H(m')$.
2. **Pre-image:** Given $x = H(m)$, find m' such that $H(m') = x$.
3. 2^{nd} **Pre-image:** Given m and $x = H(m)$, find $m' \neq m$ such that $H(m') = x$.

It should be noted here, as previously mentioned, that for an attack to be considered effective, it usually needs to have a complexity lower than $2^{n/2}$, as this is the approximate complexity of the so-called birthday attack which can be applied to every hash function independent of its inner workings and hence provides the upper bound of achievable security. The birthday attack is derived from the principle of the birthday paradox. The underlying idea is that in a room full of people the probability of two people sharing the same birthday seems quite slim. This is because the probability of one fixed day being someone's birthday is quite slim. However, reversing the process by subtracting the probability of no occurring collisions from 1, quite easily shows a collision is far more likely than initially expected. Using this principle we can estimate the complexity of a n attack by finding a collision at random to be around $2^{n/2}$. [22]

7.1 MD2

For MD2 both different principles of attack, collision attack and pre-image attack, have been successfully produced. MD2 has been considered to be insecure since 2004 when Muller published a first pre-image attack. [18]

Of the first type of attack, the collision attack, a first partial attack was found by Rogier and Chauvaud in 1997. This attack finds collisions in the MD2 compression function and therefore does not constitute an attack on the MD2 hash function, since the full function includes a checksum block. Since then a generalization of this collision attack, as well as an extension to the MD2 hash function have been found. All types will be discussed in this section and follow [18].

In order to demonstrate the following attacks, [18] introduces a visualization of the MD2 hashing process using three rectangles, labeled $A, B,$ and C. The first block, A, starts with a row of zeros, while B, the middle block, begins with the message m. From that part downwards each line represents the next round with each next row being entered from the left with the last byte of the row above and the corresponding round constant. This illustration will be used in the following pages and is shown below as figure 1. It should be noted that h_{in}, as well as h_{out} correspond to $H^{(i-1)}$, the message leading into this block's computation, and $H^{(i)}$, the resulting hash of this block, respectively. Moreover, the indexing of the blocks

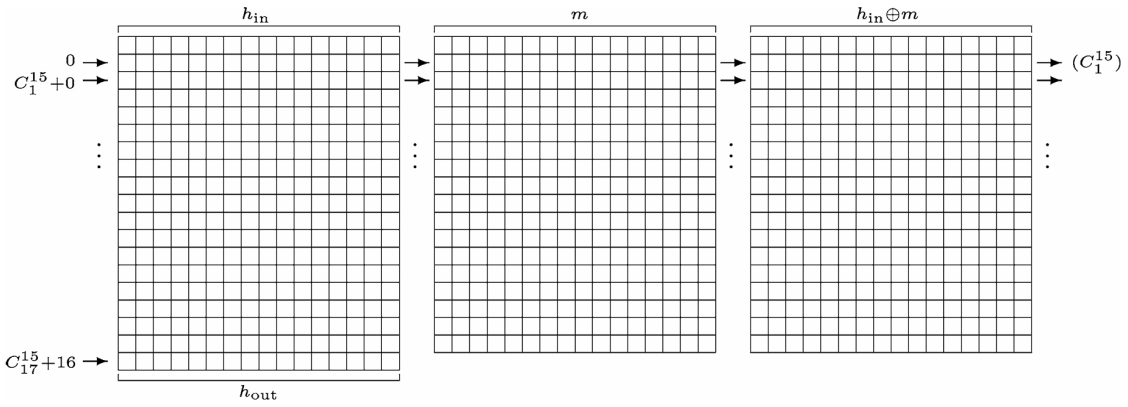

Figure 1: The MD2 hashing process as three separate blocks A, B, C by [18]

in this paper has been adapted to other conventions with i representing the column and j describing the index of the row for A_i^j. Therefore the descriptions C_1^{15} and C_{17}^{15} would correspond to C_{15}^1 and C_{15}^{17} in the following sections.

7.1.1 A First Collision Attack on the Compression Function

This collision attack targets an MD2 version without its checksum, reducing this attack to the MD2 compression function. Two prerequisites are required.

1. $A_t^0 = 0$ for $0 \leq t < 16$, meaning that the initialized buffer $H_0 = 0$ for the first 16 bytes. It should be noted that therefore, the first row in B and C are identical.

2. $A_{15}^j = B_{15}^j, = C_{15}^j$ for $1 \leq j \leq 14$, so the last columns of all three blocks are also identical.

Since each byte is computed as an XOR of two values, the one directly above and the output of a bijective function on the adjacent byte in the current block. Either value can be calculated, given the other two. Values are calculated as follows:

$$h_t^j = h_t^{j-1} \oplus S[h_{t-1}^j]$$
$$h_t^{j-1} = h_t^j \oplus S[h_{t-1}^j]$$
$$h_{t-1}^j = S[h_t^j \oplus h_t^{j-1}]$$

Using these equations, given the conditions 1 and 2, the entire block A down to row 15 can be computed. Note, that A_0^1 is solely computed from A_0^0 since the working variable p is initialized as 0. Considering this first calculation, it can be computed that all of A down to row 15 consists of 0-bytes. These values are independent from the message. Applying condition 2 a second time, parts of block B and C down to row 14 can also be computed. Selecting the missing values of B_0^{14} and B_1^{14}, the rest of block B down to row 14, including the first row equaling m, can be computed using column A_{15}. From these different choices 2^{16} different values of m can be found.
Since block A is independent from the message down to row 15, two messages collide if adjacent byte in the current block A_0^{16}, A_0^{17} and A_0^{18}. As each of those two messages can take 2^{16} different values, $2^{32}/2$ message pairs are possible. Assuming the bits are uniformly distributed, we can expect $2^{32}/2 * 1/2^{24} = 2^7$ pairs to collide.
Since this attack is in effect a birthday attack on 24 bits, a single collision can be found with

a time complexity of approximately 2^{12}.

This attack can be generalized to find multicollisions. Changing requirement 2 such that

$$A_{15}^i = B_{15}^i = C_{15}^i \text{ for } 1 \leq i \leq k \text{ where } k \text{ is some fixed value as long as } 1 \leq k \leq 14$$

allows us to vary the $16 - k$ bytes $B_0^k, ..., B_{15-k}^k$. Consequently, as k is decreased, there are more collisions as the messages now collide on the $17 - k$ bytes $A_0^{k+2}, ..., A_0^{18}$. Since the complexity of finding a collision is $2^{8(16-k)}$, the expected number of collisions is now $2^{2 \cdot 8(16-k)-1}/2^{8(17-k)} = 2^{8(15-k)-1}$.

To find an r-way collision, meaning r message blocks sharing the same hash, k may be reduced. Every set of r messages has probability $2^{-8(17-k)(r-1)}$ that all r messages have the same hash. Therefore, the number of different subsets of r out of $2^{8(16-k)}$ messages is

$$\binom{2^{8(16-k)}}{r} \approx 2^{8(16-k)r}/r!$$

and hence the expected number of r-way multicollisions is

$$2^{8(16-k)r}/r! \cdot 2^{-8(17-k)(r-1)} = 2^{8(17-k-r)}/r!.$$

7.1.2 Collision Attack with Arbitrary Chaining Input

Following the previously described attack, further collision attacks on the MD2 compression function have been formulated, including the following, which does not impose any restrictions on the so-called chaining input. The chaining input describes the state of the buffers leading into the current step. Given a certain initial buffer $h_0, ..., h_{15}$, we search for messages that, when processed from this state, yield the same output. $B_{15}^1, ..., B_{15}^k$ and $C_{15}^1, ..., C_{15}^k$ are chosen arbitrarily for some k with $1 \leq k < 18$. Using $C_{15}^1, ..., C_{15}^k$ and the known initial buffer the complete block A down to row A^{k-15} may be computed.

We now split the message block B in a left and a right half. Given the arbitrarily chosen $B_{15}^1, ..., B_{15}^k$, we can choose $l \in [0, 64]$. Using the equations from the previous collision attack, we can calculate the middle column down to row k in block B from both sides in the following way.

1. For 2^l different values of $m_0, ...m_7$, B_7^j and C_7^j for $1 \leq j \leq k$ can be computed. The message bytes are stored, indexed by the computed $2k$ bytes, in a table T_1.

2. For 2^l different values of $m_8, ...m_{15}$, B_7^j and C_7^j for $1 \leq j \leq k$ can be computed. The message bytes are stored, indexed by the computed $2k$ bytes, in a table T_2.

It should be noted here that collisions on the index within each of these tables must be handled, e.g. by storing them in a linked list.
As a next step, both tables are compared and all matches of indices are stored in another table T, indexed by the colliding indices. For each case, the complete message should be stored. Since there are 2^l pairs, each match has a probability of about 2^{-16k}. Therefore, the expected number of messages in T is 2^{2l-16k}.

The messages need to collide in $17 - k$ bytes. Consequently, they collide with a probability of about $2^{-8(17-k)}$, and hence, for a collision to occur with good probability, T needs to be at least $2^{4(17-k)}$. For any given value k, l should therefore be chosen such that

$$2l - 16k = 4(17 - k) \iff l = 6k + 34.$$

The complexity of this attack is dependent on the construction of T_1, T_2, and T. Computing the $2k$ bytes of columns 7 and 8 of B and C, T_1 and T_2 require $2 \cdot 2^l$ computations, each corresponding to about $2k$ 8-bit XORs, as using given values for $m^0, ...m^6$ part of column 6 can be pre-computed and looped through the values of m^7, which corresponds to only a single XOR for every byte in m^7. Assuming a 32-bit machine, which can compute 4 XORs in one operation, the computation can be estimated as equivalent to about $2/832 < 2^{-8}$ compression function evaluations. It should be noted, that this can only be reasonably assumed for values up to $k = 6$. Therefore, the construction and search through T_1 and T_2 each take about 2^{l-7} compression function evaluations, hence making up a complexity of about 2^{l-6} for the first part.

Adding the complexity of finding collisions in T, which requires 2^{2l-16k} compression function evaluations, the same number as the size of T, the total complexity of the attack is $2^{l-6} + 2^{2l-16k}$. Eliminating l as described above, we get a complexity of

$$2^{6k+28} + 2^{-4k+68}.$$

Since we want to minimize, observing that k must be an integer, the optimal values are $k = 4$ and consequently $l = 58$, which results in a complexity of around 2^{53}. For this attack, the probability of success is estimated at about 0.39. Increasing l e.g. $l = 58.6$ increases success probability already to about 0.93.
Using a meet-in-the-middle search memory requirements might be reduced to a negligible quantity at the cost of a slight increase in time complexity. $2^{4(17-k)}$ solutions to the meet-in-the-middle problem are still required, alongside their corresponding message blocks. To further minimize memory requirements, the attack can be repeated multiple times using a smaller value of k. A smaller k yields fewer candidate solutions per iteration, thereby reducing the memory load. However, since this approach drastically increases time complexity, assuming a one-to-one relationship between time complexity and memory requirement seems the most appropriate estimate.

7.1.3 Collision Attack on the Full MD2

The previously described attack can be extended to the full MD2 hash function. However, this attack faces two complications: (1) the padding of the message must be correct and (2) the checksum block needs to be correct. While complication (1) can easily be handled by simply adding an additional message block of 16 bytes of value 16 to every message, condition (2) is more complex.

To solve complication (2), a 2^{65}-way multicollision of messages of 65 blocks each must be constructed. This can be achieved by a method proposed by [16], using 65 collisions with chosen initial buffer values. According to this, one of these pairs will also collide on the checksum block with probability $p_1 = 1 - 1/e^2$.

In terms of time complexity, 2^{65} checksums need to be computed and stored, which takes about $16/832 \approx 2^{-5.7}$ of one compression function evaluation for updating one checksum. In total, computing 2^{65} checksums therefore takes $\Sigma_{i=0}^{65} 2^i \cdot 2^{-5.7} \approx 2^{60.3}$ compression function evaluations. Additionally, finding $65 \approx 2^6$ collisions requires about $2^{6+54} = 2^{60}$ time. Consequently, time complexity is about $2^{60.3} + 2^{60} \approx 2^{61.2}$. Factoring in the probability of finding a collision in the checksum block, the final estimation of the complexity is about $2^{61.2}/p_1 = 2^{61.4}$. As memory requirements are relatively high, with the necessity of storing 2^{65} checksums, accessing this memory may take longer than the estimated time complexity. This is also quite close to the complexity estimate for a birthday attack on MD2, which is

expected to take approximately $2^{65.5}$ time.

To reduce memory requirements, a cycle-finding method for finding collisions in the checksums might be used. It should be noted, however, that time complexity will be increased by this method. For example, as a trade-off 2^{49} checksums can be pre-computed from the first 49 message blocks and the cycle-finding algorithm can be used to iterate through the full checksums. Since each iteration requires computing $16 = 2^4$ "atomic" checksums, this takes about $2^{-1.7}$ compression function evaluations. The algorithm is estimated to repeat after $2^{64.6}$ checksums. Therefore, the time complexity is about $2^{62.9}$. Adding the complexity of finding the 2^{65}-way multicollision, total complexity is about $2^{63.3}$. Memory requirements in this compromise are dominated by the collision attack on the compression function which was estimated in the previous subsection to be about 2^{52}.

7.1.4 Pre-image Attacks

The following sections describe three different types of pre-image attacks on the MD2 compression function which can be extended to the full MD2 hash function. In the following discussion of these three types, as well as their detailed discussion below, $H^{(0)}$ will be used as notation for the first 16 bytes of the initial hash buffer. The three types include:

1. Type 1: Given $H^{(N)}$, find $H^{(0)}$ and m such that $f(H^{(0)}, m) = H^{(N)}$.
2. Type 2: Given $H^{(N)}$ and $H^{(0)}$, find m such that $f(H^{(0)}, m) = H^{(N)}$.
3. Type 3: Given $H^{(N)}$ and m, find $H^{(0)}$ such that $f(H^{(0)}, m) = H^{(N)}$.

7.1.5 Type 1 pre-image Attack on the Compression Function

Since in type 1 attacks only $H^{(N)}$ is given and the attacker can freely choose $H^{(0)}$ this attack type is always at least as efficient as type 2. When attacking MD2, this attack is more efficient than types 2 and 3. Given the final hash value, the lower right part of block A can be calculated up to the last byte of row A^3. Arbitrarily fixing A^1_{15} and A^2_{15}, two more diagonals can be computed, introducing a new condition for $H^{(0)}$ since A^1_{15} is computed only from $H^{(0)}$.

Note that with fixed $H^{(N)}$, A^1_{15}, and A^2_{15}, block B is independent of the input and depends only on the message m. We fix the last k bytes of the first row of block B arbitrarily. In the case of a longer message being hashed in several rounds of MD2, this step is required in each separate message block B. For each of these $2^{8(16-k)}$ values in B^0, B is computed and the last column stored in a table T_1.

Using the fixed A^1_{15}, we can choose all $2^{8(k+1)}$ different possible initial hash buffers $H^{(0)}$ and compute the rest of block A, as well as parts of blocks B and C. For each option, the last $k+1$ bytes of column B^{15}, as well as the uppermost left diagonal, which can be computed of block C starting from row C^1, are stored in a table T_2.

Lastly, we search for collisions in T_1, consisting of $2^{8(16-k)}$ values, and T_2, which contains $2^{8(k+1)}$ values. Since the collision should occur in the last $k+1$ bytes of block B, $2^{8(16-k)+8(k+1)-8(k+1)} = 2^{8(16-k)}$ collisions can be expected. Message pairs for each collision are then checked for agreement in the remaining $16-k$ bytes of block C, which has a probability of about $2^{-8(16-k)}$. Therefore, we can expect to find one collision within the message pairs.

When considering time complexity, step 1, computing the first bytes of A, can be ignored, since it only needs to be performed once. Since we can assume that comparison on $k+1$ bytes requires on average the same amount of work as computing one byte in the compression function, finding collisions using sorted tables can be expected to have a complexity of about $max(2^{8(16-k)}, 2^{8(k+1)})/832$. Hence, similarly to step 1, this can be left out in terms of optimizing time complexity by choice of k.

Therefore, creating tables T_1 and T_2, as well as checking message pairs for further collision on the remaining bytes, is expected to dominate the total time complexity. Balancing the creation of both tables by $16 - k = k + 1$, k would result in a non-integer. Since that complicates computation, k may be rounded to $k = 7$, resulting in a total complexity of about $2^{71} + 2^{63} + 2^{72} \cdot 9/832 = 2^{71}$. Similarly, rounding to $k = 8$ complexity can be estimated to be $2^{63} + 2^{71} + 2^{64} \cdot 8/832 = 2^{71}$. As both options require approximately 2^{72} memory access, both are equally ideal and total time complexity can be estimated at about 2^{72} in both cases.

7.1.6 Type 2 pre-image Attack on the Compression Function

In this type of attack, the initial hash value as well as the final hash value are given. Using both, most of block A can instantly be computed. In fact, fixing only A_0^2, A can be computed. Since the following attack checks every possible value of A_0^2, this attack requires a loop, which will increase time complexity. This loop over A_0^2 includes an additional loop over certain bytes of B. The idea is to find collisions in column 7 of B and C again. The attack is not guaranteed to have a solution but is expected to find one message which solves $f(H^{(0)}, m) = H^{(N)}$ in the following way.

For $A_0^2 = 0$ to 255:
{

1. Block A is computed.
2. For all possible values of $B_{15}^1, B_{15}^2, B_{15}^3,$ and B_{15}^4 do the following:
   {
   (a) For every value of $B_0^0, ..., B_7^0$ rows 1-4 of columns 0-7 of both B and C are computed. The values are stored in table T_1, indexed by the 8 bytes of column 7 of B and C. Collisions must be handled.
   (b) For every value of $B_8^0, ..., B_{15}^0$ rows 1-4 of columns 7-15 of both B and C are computed. The values are stored in table T_2, indexed again by the 8 bytes of column 7 of B and C. Collisions must be handled.
   (c) T_1 and T_2 both consist of 2^{64} values. We search for all matches between indices and store the complete message blocks in table T, indexed by the colliding indices. For 2^{128} messages on 2^{64} indices, we can expect 2^{64} candidates.
   (d) Every candidate message is checked. If it is a proper pre-image, it is added to the list M. M can be expected to have one element.
   }

}

As previously mentioned, due to the loop inside a loop, the time complexity of this attack is rather high. There are 2^8 values for the outer loop and 2^{32} different possibilities for the inner loop on each side. Hence, if we compute the inner loop once, 2^{64} individual elements are

computed and stored in the two lists. This can be estimated to be a workload similar to 2^{-8} compression function evaluations, since we can compute the 8 bytes using about two 32-bit XORs, avoiding repetitive work. Therefore, step (a) and (b) have a complexity of about 2^{57}. Since collisions between the tables can be found in linear time, step (c) takes about the same amount. Finally, candidates need to be checked. As only 32 bytes must be computed for each candidate on average, this step may be estimated to take about $2^{64} \cdot 32/832 \approx 2^{59.6}$ evaluations of the compression function. Therefore, the complexity of this attack can be estimated as $(2^{59.6} + 2 \cdot 2^{57}) \cdot 2^{40} \approx 2^{100}$ compression function evaluations. It should however be noted that memory access further adds to this estimation amounting to about 2^{105}. Memory requirements are about 2^{64} message blocks.

7.1.7 Type 3 pre-image Attack on the Compression Function

The type 3 attack is very similar to the type 2 attack and follows the same principle. Instead of A^0, B^0 is given. Still, the lower half of A can be computed. Guessing two bytes, A^1_{15} and A^2_{15}, A can still be computed. For the type 3 attack $C^1_{15}, C^2_{15}, C^3_{15},$ and C^4_{15} are guessed instead of the bytes of B. Accordingly, the first rows of A and C are computed, again using column 7 each as the meet-in-the-middle column.

Since two bytes in A need to be guessed in step 1 instead of just one, the complexity for this attack increases to about 2^{108} compression function evaluations. Again, it should be noted that including memory access, this estimate increases in this case to about 2^{113}.

7.1.8 Pre-image Attack on the Full MD2

The pre-image attack on the MD2 compression function can also be extended to the whole MD2. In this case, the most efficient type of attack is the Type 1 pre-image attack which will be discussed in this section. Instead of simply choosing the whole initial hash chaining input, however, only an initial value H_{iv} is given and used to compute the $2^{8(k+1)}$ chaining inputs $H^{(0)}$ that are required in the type 1 attack. We may do this by choosing message blocks m^0_i where $0 \leq i < 2^{8(k+2)}$ and computing $h^1_i = f(H_{iv}, m^0_i)$ for each i. Only 1 in each 256 will produce the correct value A^1_{15} that was chosen. Therefore, we end up with $2^{8(k+1)}$ chaining inputs, which is the required number as described above.

Even without the checksum, the time complexity is slightly higher to the original idea of the attack since the complexity of choosing $H^{(0)}$ is now $2^{8(k+2)}$ instead of $2^{8(k+1)-1}$. For the optimal value $k = 7$ total complexity is now about $2^{71} + 2^{72} + 2^{72} \cdot 9/832 \approx 2^{72.6}$.

The idea of this attack is to find two message blocks m_0 and m_1 such that $f(f(H^{(0)}, m_0), m_1) = H^{(N)}$ for some given output $H^{(N)}$. It should be noted that the probability of m_1 being the checksum of m_0 is only about 2^{-128}. Therefore, in order to achieve good probability of finding a pre-image with checksum m_1, 2^{128} pre-images are required in some way. Since calculating the attack 2^{128} times is not efficient, we use Joux's method of creating multicollisions again. [16] The attack follows the following five steps:

1. Using a given target hash value $H^{(N)}$ and the initial value of MD2 $H^{(0)}$, we produce a 2^{128}-way multicollision by the method of [16], starting from $H^{(0)}$. The common chaining output of those messages is called H_{128}.

2. In step 2, the main pre-image attack on the compression function as described previously is carried out using $H^{(0)} = H_{128}$. With good probability, this produces two message blocks m_0 and m_1 such that $f(H^{(0)}, m_0) = H_{129}$ and $f(H_{129}, m_1) = H^{(N)}$.

3. Using the checksum function g backwards, we can compute the checksum state $C*$ such that $g(C*, m_0) = m_1$.

4. The first 2^{64} checksums of the first 64 blocks of messages of the multicollision are computed. Furthermore, the last 2^{64} checksums of the last blocks of messages of the multicollision are calculated, this time backward by inverting the checksum. Those $2 \cdot 2^{64}$ values are then searched for a collision using a meet-in-the-middle search through the 2^{128}-way multicollision. The message found is called M.

5. $m* = M||m_0$ is a pre-image of $H^{(N)}$. It's checksum is m_1.

In terms of time complexity, step 1 requires finding 128 collisions in the compressions function. This has a complexity of about $2^{64+7} = 2^{71}$ using a Birthday attack, or 2^{60} using the method proposed by [16]. As discussed in previous sections step 2 has a complexity of about $2^{72.6}$ while step 3 has negligible complexity. Considering the tree structure of the multicollisions the complexity of step 4 can be estimated as 2^{66} evaluations of the checksum.

As step 2 dominates the attack's complexity, it can be estimated as around $2^{72.6}$, with complexity increasing to about 2^{73} if the collisions in step 1 are found using a Birthday attack. Since the meet-in-the-middle attack in step 4 can be carried out using memory-less techniques, memory requirements are 2^{72}, as described in the attack on the compression function.

Finding multiple pre-images can be achieved by more chaining inputs in the type 1 attack. In fact, a 2^t-way multicollision can be constructed in a time of about 2^{72+t} for $0 \leq t \leq 56$. For example, still using $k = 7$, choosing 2^{80} message blocks m_0^i, results in about 2^{72} valid chaining inputs and 2^8 expected pre-images, which would however increase time complexity to about 2^{80} compression function evaluations. These shorter multicollisions can also be used to produce shorter pre-images for the full hash function. For example using the 2^8-way multicollision and choosing $H_{32} = H^{(N)}$ and H_{30} arbitrarily and repeating for decreasing target hashes 16 times $(2^8)^{16} = 2^{128}$ pre-images of $H^{(N)}$ can be expected to be found, where one can be expected to have the right checksum. Complexity in this case is about $16 \cdot 2^{80} = 2^{84}$. However, the pre-images now have a length of $2 \cdot 16 - 1 = 31$ blocks, where the last block is again the checksum.

7.2 MD4

The first MD4 weaknesses have been found as early as 1991. The first complete collision attack, finding a collision with a probability of 2^{-22}, was published by Hans Dobbertin in 1996. Since then it has also been shown that the first two rounds of MD4 are not in fact one-way and therefore efficient pre-image and second pre-image attacks can be carried out. Furthermore, more efficient attacks have been proposed, two of which, one collision attack and one pre-image attack, will be presented in the following sections. [31]

The proof discussed in this section follows [31].

7.2.1 A Collision Attack on MD4

The collision attack works by selecting two messages that differ by a carefully chosen collision differential, and then deriving conditions to ensure that this differential is maintained throughout the compression process, ultimately resulting in a full collision. Finally, modifications can be made to any random message M ensuring that almost all the sufficient conditions hold, leading to a collision on any message chosen. This process can be performed in three steps.

Step 1 In this first step we choose a collision differential with specific properties as follows:

$$\Delta H^0 = 0 \longrightarrow \Delta H^N = 0$$

such that

$$\Delta M = M' - M = (\Delta M_0, \Delta M_1, ..., \Delta M_{15})$$
$$\Delta M_1 = 2^{31}, \Delta M_2 = 2^{31} - 2^{28}, \Delta M_{12} = -2^{16}$$
$$\Delta M_i = 0 \text{ for } 0 \leq i \leq 15, i \neq 1, 2, 12.$$

Following these properties and using the definition of MD4's hashing algorithm, all collision differentials are defined and all message word differences as well as chaining variable differences between M to M' for each step can be computed. The exact values for this can be found in table 7. Steps that are left out in the table have zero difference in all relevant columns. Hence, it can be seen that the collision differential consists of two internal collisions. Those collisions can be found in lines 2-25 and 36-41.

For all of these properties to hold, the conditions which have to be met are shown in table 8. These can easily be computed by using the following propositions.

1. For the nonlinear function $f(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$ the following properties can be derived:
$$f(x, y, z) = f(\neg x, y, z) \quad \text{if and only if } y = z$$
$$f(x, y, z) = f(x, \neg y, z) \quad \text{if and only if } x = 0$$
$$f(x, y, z) = f(x, y, \neg z) \quad \text{if and only if } x = 1$$

2. For the nonlinear function $g(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ the following properties can be derived:
$$g(x, y, z) = g(\neg x, y, z) \quad \text{if and only if } y = z$$
$$g(x, y, z) = g(x, \neg y, z) \quad \text{if and only if } x = z$$
$$g(x, y, z) = g(x, y, \neg z) \quad \text{if and only if } x = y$$

3. For the nonlinear function $h(x, y, z) = x \oplus y \oplus z$ the following properties can be derived:
$$h(x, y, z) = \neg h(\neg x, y, z) = \neg h(x, \neg y, z) = \neg h(x, y, \neg z)$$
$$h(x, y, z) = h(\neg x, \neg y, z) = h(x, \neg y, \neg z) = h(\neg x, y, \neg z)$$

Since these conditions are sufficient for all properties to hold, if the conditions in table 8 are met, M and M' consist of a collision.

Since there are 122 conditions messages need to fulfill in order to collide on our predefined collision differential, the probability of two messages M and M' colliding is only 2^{-122}. However, this probability can be improved in the next step by using message modification techniques.

Step 2 Messages are modified by single- or multi-step modification to increase the probability of collisions.

The **single-step modification** modifies the message at only one step of the MD4 hashing process according to the conditions given in table 8, thereby already increasing the collision probability efficiently. For example, M_1 could be modified as:

$$d_1 = d_1 \oplus (d_{1,7} \ll 6) \oplus ((d_{1,8} \oplus a_{1,8}) \ll 7) \oplus ((d_{1,11} \oplus a_{1,11}) \ll 10)$$
$$M_1 = (d_1 \gg 7) - d_0 - f(a_1, b_0, c_0)$$

In this example the probability of (M, M') colliding is already increased to 2^{-25}. However, the probability of a collision can be further increased by other modifications which are done in multiple steps.

The **multi-step modification** builds on the principle of further modifying already partially colliding messages to create full collisions. This is achieved by selecting two messages that collide in the first round, where all conditions of the first round hold, and changing single bits in certain messages, thereby already increasing the collision probability to efficiently satisfy the conditions of round 2 without influencing other aspects of the computation. Therefore, the messages M and M' need to partially collide in the first round and are then changed as follows:

1. Modifications for correcting the message to fulfill the conditions of a_5 are computed. In order to satisfy all five conditions of a_5, which can be found in table 8, messages $M_0, M_1, M_2, M_3,$ and M_4 need to be modified. For each condition, successively modify all message blocks using the same computational method as in the single-step modification. For the example of the conditions of a_5, table 1 shows the computations needed. For the first condition $a_{5,19} = c_{4,19}$, all five messages need to be modified using $i = 19$. As demonstrated in the table, the other conditions require $i = 26, 27, 29, 32$ correspondingly.

2. Modifications for satisfying the conditions of d_5 are computed in the same way, using the corresponding computations of each step. However, the last condition $d_{f,32} = b_{4,32}$ is not used in this case, since it would compromise the hash computation's internal state, altering the messages in multiple steps and rounds.

3. Further modifications in order to satisfy the conditions of $c_5, b_5, a_6, d_6,$ and c_6 are computed in the same manner. However, it must be considered that some conditions will need to be omitted again to keep the internal state intact. Furthermore, for some conditions, special properties might need to be considered. For example in this case, conditions for $c_{5,i}, i = 26, 27, 29, 32$ can be computed using the internal collision as is demonstrated in table 2.

It should be noted that the choice of modified bits, as well as the internal collision, are simply examples derived from the particular collision differential in this specific example

Step	Message block	Shift	Modified M_j	Changed Chaining Value
1	m_0	3	$m_0 \longleftarrow m_0 \pm 2^{i-4}$	${a_1}^{new} = a_1[\pm i], b_0, c_0, d_0$
2	m_1	7	$m_1 \longleftarrow (d_1 \gg 7) - d_0 - f(a'_1, b_0, c_0)$	$d_1, {a_1}^{new}, b_0, c_0$
3	m_2	11	$m_2 \longleftarrow (c_1 \gg 11) - c_0 - f(d_1, a'_1, b_0)$	$c_1, d_1, {a_1}^{new}, b_0$
4	m_3	19	$m_3 \longleftarrow (b_1 \gg 19) - b_0 - f(c_1, d_1, a'_1)$	$b_1, c_1, d_1, {a_1}^{new}$
5	m_4	3	$m_4 \longleftarrow (a_1 \gg 3) - a'_1 - f(b_1, c_1, d_1)$	a_2, b_1, c_1, d_1

Table 1: Example modifications for correcting the message to fulfill $a_{5,1}, i = 19, 26, 27, 29, 32$

Step	State value	Message block	Shift	Modified M_j	Changed Chaining Value	Extra condition in first round
6	d_2	m_5	7	$m_5 \longleftarrow m_5 + 2^{i-17}$	$d_2[i-9], a_2, b_1, c_1$	$d_{2,i-9} = 0$
7	c_2	m_6	11		$c_2, d_2[i-9], a_2, b_1$	$a_{2,i-9} = b_{1,i-9}$
8	b_2	m_7	19		$b_2, c_2, d_2[i-9], a_2$	$c_{2,i-9} = 0$
9	a_3	m_8	3	$m_8 \longleftarrow m_8 - 2^{i-10}$	$a_3, b_2, c_2, d_2[i-9]$	$b_{2,i-9} = 0$
10	d_3	m_9	11	$m_9 \longleftarrow m_9 - 2^{i-10}$	d_3, a_3, b_2, c_2	

Table 2: example of using an internal collision to modify a message to satisfy $c_{5,i}, i = 26, 27, 29, 32$

attack. Using a different collision differential leads to changed minimal sufficient conditions and hence yields different necessities and options of modification which need to be considered carefully for each condition.

Applying various modifications, as shown in the example above, yields a collision with a probability between approximately $2^{-6} \approx 2^{-2}$. Since a message can be selected by changing the last two words of the previously selected message, finding (M, M') takes one single-message modification for the first 14 words, which can be neglected. Each selected pair needs two single-message modifications for the last two words, as well as about twenty advanced modifications, each correcting one condition in the second round. In total, this amounts to about two MD4 computations. Therefore, using the lowest possible probability of about 2^{-6}, the total complexity of this MD4 attack does not exceed 2^8 MD4 computations.

7.3 MD5

Similar to the other MD-based hash algorithms, MD5 is considered insecure. The first partial attack to show this has been proposed by Den Boer and Bosselaers in 1993. This attack only finds a pseudo-collision. However, since then complete and faster attacks have been found and the use of MD5 as a secure hashing algorithm has stopped soon after. [32]

The attack described in this section is a differential attack by Wang and Yu in the year 2014. [32] The attack follows the same principle as the attack on MD4 in the previous section. A collision differential is defined, and then a pair of messages is found by computing sufficient conditions from the differential and using message modification on one of the messages to find collisions. However, in this attack instead of one message pair of 512 bits each, two message pairs $(M^{(0)}, M^{(1)})$ and $(M^{(0)'}, M^{(1)'})$ are used, where both messages consist of two blocks of 512 bits.

7.3.1 A Differential Collision on MD5

Step 1, similarly to MD4, describes the definition of the collision differential. For MD5 we select a collision differential with two iterations, which will be computed using the same techniques as for MD4, however including two separate iterations:

$$\Delta H_0 \longrightarrow \Delta H_1 \longrightarrow \Delta H = 0$$

such that

$$\Delta M^{(0)} = M^{(0)'} - M^{(0)} = (0, 0, 0, 0, 2^{31}, 0, 0, 0, 0, 0, 0, 2^{15}, 0, 0, 2^{31}, 0)$$
$$\Delta M^{(1)} = M^{(1)'} - M^{(1)} = (0, 0, 0, 0, 2^{31}, 0, 0, 0, 0, 0, 0, 2^{-15}, 0, 0, 2^{31}, 0)$$
$$\Delta H_1 = (2^{31}, 2^{31} + 2^{25}, 2^{31} + 2^{25}, 2^{31} + 2^{25}).$$

where $\Delta H_1 = (\Delta a, \Delta b, \Delta c, \Delta d)$ is the difference of the four chaining values in the first iteration. ΔM_0 is chosen to ensure a high probability of round 3 and 4 differentials to happen, Meanwhile, ΔM_1 ensures this requirement while also producing an output difference that can be canceled by the output difference of the variables of ΔH_1.

The differential for the hash values of each individual step can be found in table 9.

To efficiently work with the collision differential and to start modifying the message, a set of sufficient conditions has to be extracted from the collision differential which will then serve as the basis for modification. Using the equations and various step differences of the individual steps and their hash variables, the sufficient conditions for each chaining variable are computed step by step. The complete list of conditions that make up a sufficient set to hold the characteristics of the collision differential can be found in table 10.

As an example, the conditions for the five stages of chaining variables included in step 8 will be computed in the following paragraphs. The differential characteristic for step 8 is:

$$(\Delta c_2, \Delta d_2, \Delta a_2, \Delta b_1) \longrightarrow \Delta b_2.$$

For each of the chaining values, the corresponding equations they need to satisfy can be found in table 9. For step 8, the chaining variables need to satisfy the following equations:

$$b_1' = b_1$$
$$a_2' = a_2[7, ..., 22, -23]$$
$$d_2' = d_2[-7, 24, 32]$$
$$c_2' = c_2[7, 8, 9, 10, 11, -12, -24, -25, -26, 27, 28, 29, 30, 31, 32, 1, 2, 3, 4, 5, -6]$$
$$b_2' = b_2[1, 16, -17, 18, 19, 20, -21, -24]$$

The operation used in step 8, which will be used to compute the conditions in this example, is

$$b_2 = c_2 + (b_1 + f(c_2, d_2, a_2) + m_7 + t_7) \lll 22$$

and therefore of course simultaneously

$$b'_2 = c'_2 + (b'_1 + f(c'_2, d'_2, a'_2) + m'_7 + t_7) \lll 22.$$

Since parts of this can be left out, it is also useful to work just with the inner function

$$\phi_7 = f(c_2, d_2, a_2) = (c_2 \wedge d_2) \vee (\neg c_2 \wedge a_2).$$

We use the notation c_2^f for the c_2 which occurs inside of the function f and c_2^{nf} for the c_2 outside in order to distinguish these two and find that since $\Delta b_1 = 0$ and $\Delta m_7 = 0$, we know that $\Delta b_2 = \Delta c_2^{nf} + (\Delta\phi_7 \lll 22)$.

Using this equation and fixing one or two of the variables in f, we can reduce f to a single variable and compute the following set of sufficient conditions of the variables of step 8.

For example for $b_{2,1}$ we know that if $d_{2,11} = \overline{a_{2,11}} = 1$, then $\Delta\phi_{7,11} = 1$. Shifting by $\lll 22$, $\Delta\phi_{7,11}$ is now in the first position and therefore, since $\Delta c_{2,1}^{nf} = 0$, we know that $\Delta b_{2,1} = \Delta c_{2,1}^{nf} + \Delta\phi_{7,11} = 1$. The following conditions are computed using the same techniques:

1. Conditions for each non-zero bit in Δb_2
   (a) Conditions $b_{2,1} = 0$ and $d_{2,11} = 1$ ensure the change of the first bit of b_2.
   (b) Conditions $d_{2,26} = \overline{a_{2,26}} = 0$, $b_{2,16} = 0$ and $b_{2,17} = 1$ ensure the change of the 16-th and 17-th bit of b_2.
   (c) Conditions $d_{2,28} = \overline{a_{2,28}} = 0$, $b_{2,i} = 0$ for $i = 18, 19, 20$ and $b_{2,21} = 1$ ensure the change of the 18-th to 21-st bit of b_2.
   (d) Conditions $d_{2,3} = \overline{a_{2,3}} = 0$ and $b_{2,24} = 1$ ensure the change of the 24-th bit of b_2.
2. Conditions for each zero bit in Δb_2
   (a) Condition $c_{2,17} = 0$ ensures that the changes made in the 7-th to 12-th bit in $c_2'^{nf}$ and the 17-th bit of a'_2 result in no bit change in b_2. This can for example be proven by the following equation:

$$\Delta c_2^{nf}[7, ..., 11, 12] + (\Delta\phi_7[17] \lll 22) = -2^6 + 2^6 = 0.$$

   (b) Conditions $d_{2,i} = a_{2,i}$ for $i \in \{1, 2, 4, 5, 25, 27, 29, 30, 31\}$ ensure that the change in the i-th bit of c_2^f results in no change in b_2.
   (c) Conditions $c_{2,i} = 1$ for $i \in \{13, 14, 15, 16, 18, 19, 20, 21, 22, 23\}$ ensure that the change in the i-th bit of c_2^f results in no change in b_2.
   (d) Condition $d_{2,6} = \overline{a_{2,6}} = 0$ ensures that the change in the 6-th bit of c_2^f results in no change in b_2.
   (e) Condition $a_{2,32} = 1$ ensures that the changes in the 32-nd bit of c_2^f and the 32-nd bit in d_2 result in no change in b_2.
   (f) Conditions $d_{2,i} = 0$ for $i \in \{8, 9, 10\}$ ensure that the changes in the i-th bit of a_2 and the i-th bit in c_2^f result in no change in b_2.

Step	Message Block	Shift	Modified M_j	Changed Chaining Value
2	m_1	12	$m_1 \longleftarrow m_1 + 2^{26}$	$(d_1^{new}, a_1, b_0, c_0)$
3	m_2	17	$m_2 \longleftarrow (c_1 - d_1^{new} \gg 17) - c_0 - \phi_2(d_1^{new}, a_1, b_0) - t_2)$	$(c_1, d_1^{new}, a_1, b_0)$
4	m_3	22	$m_3 \longleftarrow (b_1 - c_1 \gg 22) - b_0 - \phi_3(c_1, d_1^{new}, a_1) - t_3$	$(b_1, c_1, d_1^{new}, a_1)$
5	m_4	7	$m_4 \longleftarrow (a_2 - b_1 \gg 7) - a_1 - \phi_4(b_1, c_1, d_1^{new}) - t_3$	$(a_2, b_1, c_1, d_1^{new})$
6	m_5	12	$m_5 \longleftarrow (d_2 - a_2 \gg 12) - d_1^{new} - \phi_5(a_2, b_1, c_1) - t_5$	(d_2, a_2, b_1, c_1)

Table 3: Message Modifications for correcting $a_{5,32}$

Similarly to MD4, the next step modifies the message. Again we start with single-step modification to ensure that the conditions of round 1 hold. For each intermediate hash value in the first 16 steps, all conditions are considered and a new value is computed. Using the modified value the message is modified again. By this method, we can ensure that in round 1 all conditions hold with probability 1.

For example, the first intermediate value that holds conditions is c_1. This variable can be changed in the following way:

$$c_1^{new} \longleftarrow c_1^{old} - c_{1,7}^{old} \cdot 2^6 - c_{1,12}^{old} \cdot 2^{11} - c_{1,20}^{old} \cdot 2^{19}$$

$$m_2^{new} \longleftarrow ((c_1^{new} - c_1^{old}) \gg 17) + m_2^{old}.$$

After modifying each word of $M^{(0)}$ the probability of the first iteration differential holding is 2^{-43} and applying the same modifications to $M^{(1)}$, increases the probability of the second iteration holding to 2^{-37}.

In a second round of modification, multi-step modification is used to fulfill as many conditions as possible with minimal changes. For this part, different changes can be made which influence the time efficiency of computing the necessary changes as well as the probability of the iteration differentials holding. For example the message modifications for correcting $a_{5,32}$ can be computed according to table 3.

Regarding time complexity both iterations are considered separately. In the first part, a message M_0 is selected randomly and modified according to the previous description. With probability 2^{-37} this M_0 and M_0' generate the first iteration differential

$$\Delta M_0 \longrightarrow (\Delta H_1, \Delta M_1)$$

which is tested for its conditions by applying the compression function. Similarly, in the second iteration, for which a random message M_1 is selected and modified, M_1 and $M_1 + \Delta M_1$ produce the second iteration differential

$$(\Delta H_1, \Delta M_1) \longrightarrow \Delta H = 0$$

with probability 2^{-30}.

It should be noted at this point, that finding a message pair (M_0, M_0') usually requires only one single-step modification since we only need to change the last two words of the previously selected message.

In total, since the only remaining computational step of two single-step modifications and seven multi-step modifications, takes the time of approximately two MD5 operations, the time complexity of all computations concerning the first iteration is about 2^{39} while the second part does not exceed 2^{32}.

7.4 SHA 1

The first partial attack on SHA-1 was proposed in 2005 by Wang et. al. This attack was improved and further developed by De Cannière and Rechberger at ASIACRYPT 2006. The following section will follow this improved proof. [4] Since then, further attempts have been made to develop other attacks. Even though some claims have been made, no complete attacks have been published since. Since this attack still has a very high complexity, some still consider this hashing algorithm secure enough. However, NIST and other security institutions consider SHA-1 to be insecure since it has been proven to be vulnerable to attacks. Therefore, NIST recommends using SHA-1's followers SHA-2, which is its direct ascendant and shares most of its inner workings, or SHA-3, which is a completely new algorithm. Both algorithms will be discussed in the next chapter.

Similar to the attack on MD5 in the previous section, the collision attack on SHA-1 exploits the likelihood of certain so-called characteristics propagating through multiple rounds of the computation. Some properties, called *linear characteristics*, connect input and output of a given step. They may be described through linear equations, since they either directly translate through the computation or their probability for a certain input-output relationship is sufficiently probable to be described in a linear model. However, most changes cannot be this easily described. These more complex characteristics called *non-linear characteristics*, cannot be simply described by linear equations. However, they still have certain non-random properties that are useful for collision attacks. The underlying idea is that certain operations can be considered given, while the complex, non-linear characteristics connecting most parts of the computation can be used to gather information on the probability of certain outputs which is in turn used to create a more efficient set of conditions for finding collisions with high probability.

Since preparing the message by padding it and creating the message schedule is an easily reversible process, the focus of every attack found so far lies on the computation of the variables a, b, c, d, e, which start as the initial given hash values and end up making up the final hash. It should be noted that it is sufficient to only analyze and describe the variable a along with the word schedule, since the variables are cycled in each round, in the following way:

$$b_i = a_{i-1}, c_i = ROTL^{30}(a_{i-2}), d_i = ROTL^{30}(a_{i-3}), e_i = ROTL^{30}(a_{i-4})$$

7.4.1 Notation and Definitions

Throughout this proof differentials between certain words ΔX will be represented as a set within a pair of words X^2, containing the bit values satisfying the conditions. However, in order to represent the complex propagation of certain characteristics over several steps, a list of symbols, each describing the relationship between one pair of bits, will be used. The symbols are given in table 4.

For this proof, some further notation and definitions are required. These will be given here since they are only relevant to this section.

Def.1 Linear Characteristic: *A linear characteristic is the property of a certain differential, or sequence of differentials in a message schedule or variable which can be described by a linear equation or at least have its probability modeled linearly.*

Def.2 Non-Linear Characteristic: *A non-linear characteristic is the property of a cer-*

(x_i, x_i*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓
-	✓	×	×	✓
x	×	✓	✓	×
0	✓	×	×	×
u	×	✓	×	×
n	×	×	✓	×
1	×	×	×	✓
#	×	×	×	×
3	✓	✓	×	×
5	✓	×	✓	×
7	✓	✓	✓	×
A	×	✓	×	✓
B	✓	×	✓	✓
C	×	×	✓	✓
D	✓	×	✓	✓
E	×	✓	✓	✓

Table 4: Notation describing the relationship of word pairs [4]

tain differential, or sequence of differentials in a message schedule or variable, whose complex probability cannot be described by a linear equation.

Def.3 Message Freedom: *The message freedom $F_W(i)$ is the number of possibilities of W_i^2, given a fixed W_j^2, where $0 \leq j < i$, without violating any linear characteristics.*

It should be noted that in SHA-1 all words after the first 16 have $F_W(i) = 1$.

Def.4 Uncontrolled Probability: *The uncontrolled probability is the probability that an output follows a certain characteristic, given that all leading pairs follow it.*

For the variable a in step i of SHA-1 this is defined as follows:

$$P_u(i) = P(A_{i+1}^2 \in \Delta A_{i+1} | A_{i-j}^2 \text{ for } 0 \leq j < 5 \text{ and } W_i^2 \in \Delta W_i).$$

Def.5 Controlled Probability: *The controlled probability is the probability that there exists at least one pair of message words W_i^2 following the characteristic, such that the output follows the characteristic, given the leading pairs follow as well.*

For the variable a in step i of SHA-1 this is defined as follows:

$$P_c(i) = P(\exists W_i^2 \in \Delta W_i : A_{i+1}^2 | A_{i-j}^2 \in \Delta A_{i-j} \text{ for } 0 \leq j < 5).$$

It should be noted that, in definitions 4 and 5, the range for j as $0 \leq j < 5$ accounts for the variables b, c, d, e by considering the last 5 stages of a while ignoring repetitive information of steps further back.

We will use these definitions to calculate the total work factor of our collision search. In order to do so, tree structure is used to represent the state changes of each bit throughout the computation. If there is no information on the state of the given bit before and after a computational step, the tree splits into four branches at each node. Using these definitions, the number of nodes $N_s(i)$ visited in the collision search can be considered to be a weighted

probabilistic calculation. The average number of children of a node at step i can be computed as $F_W(i) \cdot P_u(i)$. However, considering that only a fraction $P_c(i)$ of the nodes at step i have any children, the following recursion, which describes the search process until its final step N, can be formulated:

$$N_s(i) = \begin{cases} 1 & \text{for } i = N \\ max\{N_s(i+1) \cdot F_W(i)^{-1} \cdot P_u^{-1}(i), P_c^{-1}(i)\} & \text{for } i < N \end{cases}$$

The sum of these nodes makes up the total work factor which can be written as follows:

$$N_W = \Sigma_{i=1}^{N} N_s(i)$$

The goal is to minimize the total work factor by introducing conditions that propagate with high efficiency. This can be achieved in three stages.

7.4.2 Stages of Constructing Characteristics

In stage 1, high probability differential characteristics are searched. Since the expanded message words after step 16 are entirely determined by the initial message in the first 16 words, the message freedom after step 16 is always 1. Therefore, it is most efficient to impose nonlinear conditions after this step as they propagate more effectively. In order to find suitable characteristics, the compression function is linearized and this linear variant is used to search for characteristics. These corresponding message differences are then determined and imposed onto the nonlinear variant, leaving out the most significant and the two least significant bits to avoid inconsistencies caused by the nonlinear function. This first stage creates a bottleneck for steps 1-16 which will be the focus of stage 2.

After stage 1, the probability of linear or high-probability characteristics is incredibly low. Hence, other conditions are required. Even though the probability in this stage is not as critical as in stage 1, ideally the characteristics found should still be as sparse as possible, since this enhances their probability of propagation. The probabilistic algorithm used in this stage is applied directly to the compression function and uses the fact that fortunately the sparser a characteristic is, the higher its probability of being found by imposing differences at random unrestricted bits and calculating how the condition propagates. Those operations are focused on steps 12 to 16 which form the bottleneck created in stage 1. This algorithm is used until there are no unrestricted positions left. If inconsistencies are found, the computation begins again. It should be noted that it has been found that picking `x`-bits and guessing their sign reveals inconsistencies considerably faster and is therefore a way of optimizing the algorithm.

Lastly, at this stage, all necessary restrictions have been imposed and the final, third stage is now used to further improve the total work factor. This is achieved by iterating through all bit positions of variables, expanding message words again, and trying out potential conditions by imposing them and checking for inconsistencies. These local optimizations are performed until the total work factor is satisfactory. It can be found that efficiency generally increases as long as the reduction in the size of the search tree is still higher than the expected number of surviving leaves. Since the expected number will result to be less than 1, we know that a message pair following the characteristic is only expected with a certain probability. This means that the search might need to be repeated, which aligns with the example computation.

7.4.3 Example

[4] also provide the first published example of a two-block collision characteristic with the standard initial hash values, along with a colliding message. Their characteristic is reduced

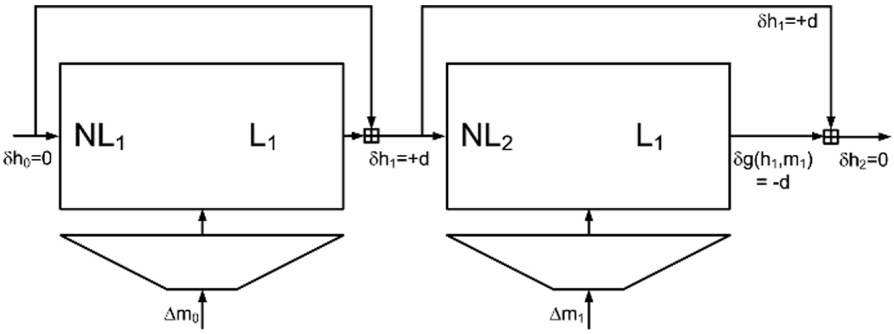

Figure 2: Two-block approach of producing collisions

to 64 steps, which suffices to find the collision. To calculate this characteristic, starting by choosing a message difference, they first compute high-probability characteristics which are treated as linear characteristics for the first block. These only allow XOR differences and the sign of each difference is determined in a later stage of the attack. It has been found that keeping so-called local collisions within the message difference aids in constructing the high-probability characteristics. These local collisions are comprised of a disturbing difference accompanied by a set of correcting differences It should be noted that any variant with fewer than 80 steps, such as the example given in this paper, can most likely use a shift of the same high-probability characteristics as used here for an efficient computation. The resulting output difference of the compression function is dealt with in the second block and can therefore be nonzero after the first block. Hence, no restrictions are imposed.

After describing a pseudo-near collision by fixing the conditions of linear or high-probability characteristics for the first block, the non-linear characteristics are computed to find a collision-producing characteristic. These feed the zero-difference chaining variables towards the linear characteristics, resulting in the previously mentioned modular difference of $+d$ at the end of block 1. Nevertheless, the same linear characteristics are used in block 2, which further feeds the difference. This will be used to check the resulting output in a final step since the created difference can be canceled out with the expected low-weight difference after the last step of the second block. To achieve this, a new non-linear characteristic is introduced into the second block, which takes into account the forwarded chaining input of the second block as well as the difference between the first and second block. Lastly, we check if the output difference of block 2, $\delta g(h_1, m_1)$, cancels out the output difference of the first block, δh_1, making sure that

$$\delta g(h_1, m_1) + \delta h_1 = 0.$$

Figure 2 by [4] visualizes this process while tables 13 and 14 show the concrete set of characteristics for block 1 and 2 of their example 2-block attack. Finally, table 5 shows two colliding messages found by this attack, along with their colliding hash value. Note that table 5 highlights that, for both message blocks, the bitwise differences between corresponding words in Message 1 and Message 2 remain consistent across the first 16 words.

i	Message 1 (m_0), first block	Message 1 (m_1), second block
1-4	63DAEFDD 30A0D167 52EDCDA4 90012F5F	3B2AB4E1 AAD112EF 669C9BAE 5DEA4D14
5-8	0DB4DFB5 E5A3F9AB AE66EE56 12A5663F	1DBE220E AB46A5E0 96E2D937 F3E58B63
9-12	D0320F85 8505C67C 756336DA DFFF4DB9	BE594F1C BD63F044 50C42AA5 8B793546
13-16	596D6A95 0855F129 429A41B3 ED5AE1CD	A9B24128 816FD53A D1B663DC B615DD01
i	Message 2 (m_0^*), first block	Message 2 (m_1^*), second block
1-4	63DAEFDE 70A0D135 12EDCDE4 70012F0D	3B2AB4E2 EAD112BD 269C9BEE BDEA4D46
5-8	ADB4DFB5 65A3F9EB 8E66EE57 32A5665F	BDBE220E 2B46A5A0 B6E2D936 D3E58B03
9-12	50320F84 C505C63E B5633699 9FFF4D9B	3E594F1D FD63F006 90C42AE6 CB793564
13-16	596D6A96 4855F16B 829A41F0 2D5AE1EF	A9B2412B C16FD578 11B6639F 7615DD23
i	XOR-difference are the same for both blocks	
1-4	00000003 40000052 40000040 E0000052	00000003 40000052 40000040 E0000052
5-8	A0000000 80000040 20000001 20000060	A0000000 80000040 20000001 20000060
9-12	80000001 40000042 C0000043 40000022	80000001 40000042 C0000043 40000022
13-16	00000003 40000042 C0000043 C0000022	00000003 40000042 C0000043 C0000022
i	The colliding hash values	
1-5	A750337B 55FFFDBB C08DB36C 0C6CFD97 A12EFFE0	

Table 5: Example of a 64-step collision using the standard IV [4]

8 State-of-the-Art Hash Functions

As shown in the last chapter, hash functions such as the MD-family as well as SHA-1 are not considered secure anymore. This does not however mean that they are not still in use. Hash functions that are proven to be vulnerable to attacks, such as SHA-1, are still often used to check data integrity. If the hash is used for example to check whether a message or string of data has been altered, SHA-1, or even lower levels of security, are sufficient. In that case, the hash that is calculated from the data is checked against the original hash value which has been stored or sent along with the message. Since the hash is not necessarily required to be secure, easier algorithms, that require less calculation, are preferred for some applications. For secure applications, however, only the following two should still be considered.

8.1 SHA-2

Similar to previously mentioned algorithms, SHA-2 requires padding and parsing the message in advance. For SHA-2, both of these steps are defined identically to SHA-1, which has already been discussed in section 4.4. Subsequent steps of the algorithm are also based on the same principles as its predecessors. The complete definitions of all variants of the second version of SHA in this chapter also follow the specifications in [1].

Secondly, the initial hash value also needs to be set before starting the algorithm. This value is one of two key elements that differentiate the different variants, SHA-256 and SHA-512, as well as truncated versions such as SHA-224, SHA-384, and SHA-512/t.

8.1.1 SHA-256

For SHA-256 the initial hash value consists of eight 32-bit words. However, in this case, these words are hex representations of the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

$$
\begin{aligned}
H_0^{(0)} &= \texttt{6a09e667} \\
H_1^{(0)} &= \texttt{bb67ae85} \\
H_2^{(0)} &= \texttt{3c6ef372} \\
H_3^{(0)} &= \texttt{a54ff53a} \\
H_4^{(0)} &= \texttt{510e527f} \\
H_5^{(0)} &= \texttt{9b05688c} \\
H_6^{(0)} &= \texttt{1f83d9ab} \\
H_7^{(0)} &= \texttt{5be0cd19}
\end{aligned}
$$

SHA-2 can hash any message M of length l, where $0 \leq l < 2^{64}$, using a message schedule of sixty-four 32-bit words $W_0, W_1, ..., W_{63}$ as well as eight 32-bit working variables $a, b, c, d, e, f, g,$ and h and two temporary words T_1 and T_2. The hash values are labeled $H_0^{(i)}, H_1^{(i)}, ...H_7^{(i)}$, holding the initial value, $H^{(0)}$ and resulting in the final hash value, $H^{(N)}$.

In SHA-2 computation, each message block $M^{(0)}, M^{(1)}, ..., M^{(N)}$ is processed in order, meaning the algorithm is used N times. Addition $(+)$ is performed modulo 2^{32}. The steps to the algorithm are as follows:

For $i = 1$ to N:
{

1. The message schedule, consisting of one 32-bit message block, is prepared and appended as follows:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \end{cases}$$

2. All eight working variables are initialized with the $(i-1)^{st}$ hash value.

$$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \\ f &= H_5^{(i-1)} \\ g &= H_6^{(i-1)} \\ h &= H_7^{(i-1)} \end{aligned}$$

3. For $t = 0$ to 63 :
   {

$$\begin{aligned} T_1 &= h + \Sigma_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t \\ T_2 &= \Sigma_0^{\{256\}}(a) + Maj(a,b,c) \\ h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2 \end{aligned}$$

   }

4. The i^{th} intermediate hash value $H^{(i)}$ is computed such that

$$\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} \\ H_2^{(i)} &= c + H_2^{(i-1)} \\ H_3^{(i)} &= d + H_3^{(i-1)} \\ H_4^{(i)} &= e + H_4^{(i-1)} \\ H_5^{(i)} &= f + H_5^{(i-1)} \\ H_6^{(i)} &= g + H_6^{(i-1)} \\ H_7^{(i)} &= h + H_7^{(i-1)} \end{aligned}$$

}

Lastly, after the steps have been repeated N times, once for each message block $M^{(N)}$, the message digest can be directly read from the final hash values. In this last step lies the second key difference between SHA-224 and SHA-256.

For SHA-256, the message digest is 256 bits long and defined by all eight final hash values as follows:

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}.$$

8.1.2 SHA-224

As mentioned, SHA-224 uses the same algorithm as previously described. However, this version requires different initial hash values. Furthermore, in SHA-224 the message digest is a truncated version of the final hash value. For SHA-224, the initial hash value consists of the following eight 32-bit words:

$$\begin{aligned} H_0^{(0)} &= \texttt{c1059ed8} \\ H_1^{(0)} &= \texttt{367cd507} \\ H_2^{(0)} &= \texttt{3070dd17} \\ H_3^{(0)} &= \texttt{f70e5939} \\ H_4^{(0)} &= \texttt{ffc00b31} \\ H_5^{(0)} &= \texttt{68581511} \\ H_6^{(0)} &= \texttt{64f98fa7} \\ H_7^{(0)} &= \texttt{befa4fa4} \end{aligned}$$

The 224-bit message digest is acquired from these final hash values $H^{(N)}$ by using only its leftmost 224 bits

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)}.$$

8.1.3 SHA-512

Using the same functions, as well as constants, all SHA-512 versions use the same algorithm, with only minor definition differences in the final hash value, which is also very similar to SHA-2. The main difference, which is also the main reason for increased security in SHA-512, is the length of the message and words being used, since SHA-256 uses 32-bit words, whereas SHA-512 uses 64-bit words. Within this algorithm, addition (+) is therefore performed modulo 2^{64}.

SHA-512 is a group of algorithms containing SHA-512, SHA-384, SHA-512/224, SHA-512/256 and other truncated versions. In general, all of these versions are computed the same way, the only difference between them being the length of the final hash value, as some get truncated in the last step. This assures that SHA-512, the base form of this algorithm, which outputs a 512-bit final hash, can be adapted for a variety of situations by generating shorter hashes.

In its base form, SHA-512 is able to hash messages M with length l, where $0 \leq l \leq 2^{128}$ and results in a 512-bit message digest. It uses a message schedule of eighty 64-bit words $W_0, W_1, ..., W_{79}$, eight working variables $a, b, c, d, e, f, g,$ and h, and two temporary words T_1 and T_2. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, ..., H_7^{(i)}$, holding the initial hash value $H^{(0)}$ and resulting in the final hash value $H^{(N)}$.

Along with using the same functions throughout the algorithm, padding for SHA-512 follows the same principle as previous SHA-versions. However, it is adapted to the message length. To the original message M of length l the bit 1 is appended, followed by k zero bits, where k is the smallest non-negative solution to the equation $l + 1 + k \equiv 896 \bmod 1024$. Lastly, a 128-bit binary representation of the length l of the message is appended to the message, resulting in a padded message of length l, where $1024|l$.

The padded message is then parsed into N 1024-bit blocks $M^{(0)}, M^{(1)}, ..., M^{(N)}$. Each of these blocks will be used as sixteen separate 64-bit words $M_0^{(i)}, M_1^{(i)}, ..., M_{15}^{(i)}$.

Lastly, before hashing the message, an initial hash value $H^{(0)}$ needs to be set. These initial hash values differ for the various versions of SHA-512. An algorithm for setting the hash values of different versions will be discussed at the end of this section, after defining the basic version of the SHA-512 algorithm itself. For SHA-512, the initial hash value is set as a hex representation of the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

$$
\begin{aligned}
H_0^{(0)} &= \texttt{6a09e667 f3bcc908} \\
H_1^{(0)} &= \texttt{bb67ae85 84caa73b} \\
H_2^{(0)} &= \texttt{3c6ef372 fe94f82b} \\
H_3^{(0)} &= \texttt{a54ff53a 5f1d36f1} \\
H_4^{(0)} &= \texttt{510e527f ade682d1} \\
H_5^{(0)} &= \texttt{9b05688c 2b3e6c1f} \\
H_6^{(0)} &= \texttt{1f83d9ab fb41bd6b} \\
H_7^{(0)} &= \texttt{5be0cd19 137e2179}
\end{aligned}
$$

Finally, SHA-512 is computed by processing each message block $M^{(0)}, M^{(1)}, ..., M^{(N)}$ using the following steps:
For $i = 1$ to N: {

1. Preparing the message schedule as follows:

$$
W_t = \begin{cases} M_t^{(i)} & \text{when } 0 \leq t \leq 15 \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} & \text{when } 16 \leq t \leq 79 \end{cases}
$$

2. Working variables $a, b, c, d, , e, f, g,$ and h are initialized with the $(i-1)^{st}$ hash value.

$$
\begin{aligned}
a &= H_0^{(i-1)} \\
b &= H_1^{(i-1)} \\
c &= H_2^{(i-1)} \\
d &= H_3^{(i-1)} \\
e &= H_4^{(i-1)} \\
f &= H_5^{(i-1)} \\
g &= H_6^{(i-1)} \\
h &= H_7^{(i-1)}
\end{aligned}
$$

3. For $t = 0$ to 79: {

$$
\begin{aligned}
T_1 &= h + \Sigma_1^{\{512\}}(e) + Ch(e, f, g) + K_t^{\{512\}} + W_t \\
T_2 &= \Sigma_0^{\{512\}}(a) + Maj(a, b, c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 + T_2
\end{aligned}
$$

}

4. The i^{th} hash value is computed

$$\begin{aligned}
H_0^{(i)} &= a + H_0^{(i-1)} \\
H_1^{(i)} &= b + H_1^{(i-1)} \\
H_2^{(i)} &= c + H_2^{(i-1)} \\
H_3^{(i)} &= d + H_3^{(i-1)} \\
H_4^{(i)} &= e + H_4^{(i-1)} \\
H_5^{(i)} &= f + H_5^{(i-1)} \\
H_6^{(i)} &= g + H_6^{(i-1)} \\
H_7^{(i)} &= h + H_7^{(i-1)}
\end{aligned}$$

}

Steps 1 to 4 are repeated for each block, so N times. The resulting 512-bit message digest of the message M is then

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}.$$

8.1.4 SHA-384

As previously mentioned, alternative truncated versions differ in two aspects, initial hash value and message digest. There are three commonly used versions alternating from the basic SHA-512, as well as an algorithm effectively allowing for every possible truncated variant.

SHA-384 is the most frequently used truncated variant of SHA-512 and is obtained by computing SHA-512 following the common algorithm and truncating the left-most 384 bits of the final hash value as a message digest, resulting in a message digest of the following form:

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)}.$$

Furthermore, it is the only specialized variant using pre-defined initial hash values. Other truncated SHA-512 variants using a specific algorithm to compute their initial hash values will be discussed in the next section. For SHA-384, the initial hash value is defined to be the first sixty-four bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers.

$$\begin{aligned}
H_0^{(0)} &= \texttt{cbbb9d5d}\,\texttt{c1059ed8} \\
H_1^{(0)} &= \texttt{629a292a}\,\texttt{367cd507} \\
H_2^{(0)} &= \texttt{9159015a}\,\texttt{3070dd17} \\
H_3^{(0)} &= \texttt{152fecd8}\,\texttt{f70e5939} \\
H_4^{(0)} &= \texttt{67332667}\,\texttt{ffc00b31} \\
H_5^{(0)} &= \texttt{8eb44a87}\,\texttt{68581511} \\
H_6^{(0)} &= \texttt{db0c2e0d}\,\texttt{64f98fa7} \\
H_7^{(0)} &= \texttt{47b5481d}\,\texttt{befa4fa4}
\end{aligned}$$

8.1.5 SHA-512/t

With the exception of SHA-384, all other truncated variants of SHA-512 follow the same pattern, starting with an additional step of generating their initial hash values, using the *SHA-512/t IV Generation Function*. This algorithm is used by all SHA-512/t variants, which

includes all t-bit algorithms using SHA-512 with an output truncated to its left-most t bits.

For SHA-512/t, t might be any positive integer without a leading zero such that $t < 512$ and $t \neq 384$. For example, t might be 256, and "SHA-512/t" might be "SHA-512/256", which is probably the most frequently used truncated variant of SHA-512. The initial hash value of SHA-512/t is computed in the following way:

1. Denote $H^{(0)'}$ to be the initial hash value of SHA-512 as above.
2. Denote $H^{(0)''}$ to be the initial hash value computed below.
3. $H^{(0)}$ will be the initial value for SHA-512/t.
4. For $i = 0$ to 7
   {
   $H_i^{(0)''} = H_i^{(0)'} \oplus$ `a5a5a5a5a5a5a5a5`
   }
5. $H^{(0)} = SHA - 512(\text{"}SHA - 512/t\text{"})$ using $H^{(0)''}$ as the initial value, where t is the specific truncation value.

Using $H^{(0)}$ as its initial hash value, a message M is then computed using the regular SHA-512, truncating the final hash value to its left-most t bits, obtaining a message digest of length t bit.

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
l	0	1	2	3	4	5	6

Table 6: Caption

8.2 SHA-3

The newest secure hash algorithm is SHA-3. This algorithm has only been in use since 2015 and fundamentally differs from previous secure hash algorithms. SHA-3 is based on the cryptographic family $KECCAK$ and uses a so-called sponge function instead of a Merkle-Damgård construction. In contrast to previous algorithms, SHA-3 was not introduced in order to replace its predecessor, SHA-2, which is still in use and is considered secure, but rather to provide another secure hash algorithm in the NIST-standards. SHA-3 was developed as part of a NIST hash function competition, which began in 2006. After several rounds of testing and slight changes within the process of the competition, SHA-3 was declared the winner among 51 contestants in 2015. The following definition follows [2].

In this chapter the basic permutation used for SHA-3, $KECCAK - p$ is defined, followed by further specifications and alternate versions. Generally, $KECCAK - p$ is specified according to two parameters: the *width* of the permutation, which defines the fixed length of the strings being permuted, and the number of *rounds*, which are the iterations of each internal transformation. Accordingly, the permutation used in all variants of SHA-3 is usually written as $KECCAK - p[\mathsf{b}, n_r]$.

Before defining the permutation itself, some notational aspects have to be considered. Instead of saving the message string as a bit string and operating bit-wise, SHA-3 uses a different approach by representing the message and operating on it as a three-dimensional array. The state array A is defined to be a 5-by-5-by-w array, representing the state. Its indices are integer triples (x, y, z), where $0 \leq x < 5, 0 \leq y < 5$ and $0 \leq z < \mathsf{w}$. Each bit is referenced by its corresponding indices in the following form: $A[x, y, z]$. The array along with the naming conventions of its parts are shown in figure 3.

Since the b-bit string is saved into the array's 5-by-5 slices, there are two more qualities related to the state's bit-length: the lane size $\mathsf{w} = \mathsf{b}/25$ and the binary logarithm of the lane size $\mathsf{l} = log_2(\mathsf{b}/25)$. The seven possible values for these variables are given in the table below.

Before taking the form of a state array, the message is input as a b-bit string representing the state. The state is denoted by S with indices 0 to $\mathsf{b} - 1$ as follows:

$$S = S[0]||S[1]||...||S[\mathsf{b} - 1].$$

To convert the input string into an array, the state array is defined as follows. The labeling conventions for the indices of all three dimensions are illustrated in figure 4.

$$A[x, y, z] = S[\mathsf{w}(5y + x) + z]$$

$$\text{for all triples } (x, y, z) \text{ such that } 0 \leq x < 5, 0 \leq y < 5, \text{ and } 0 \leq z < w$$

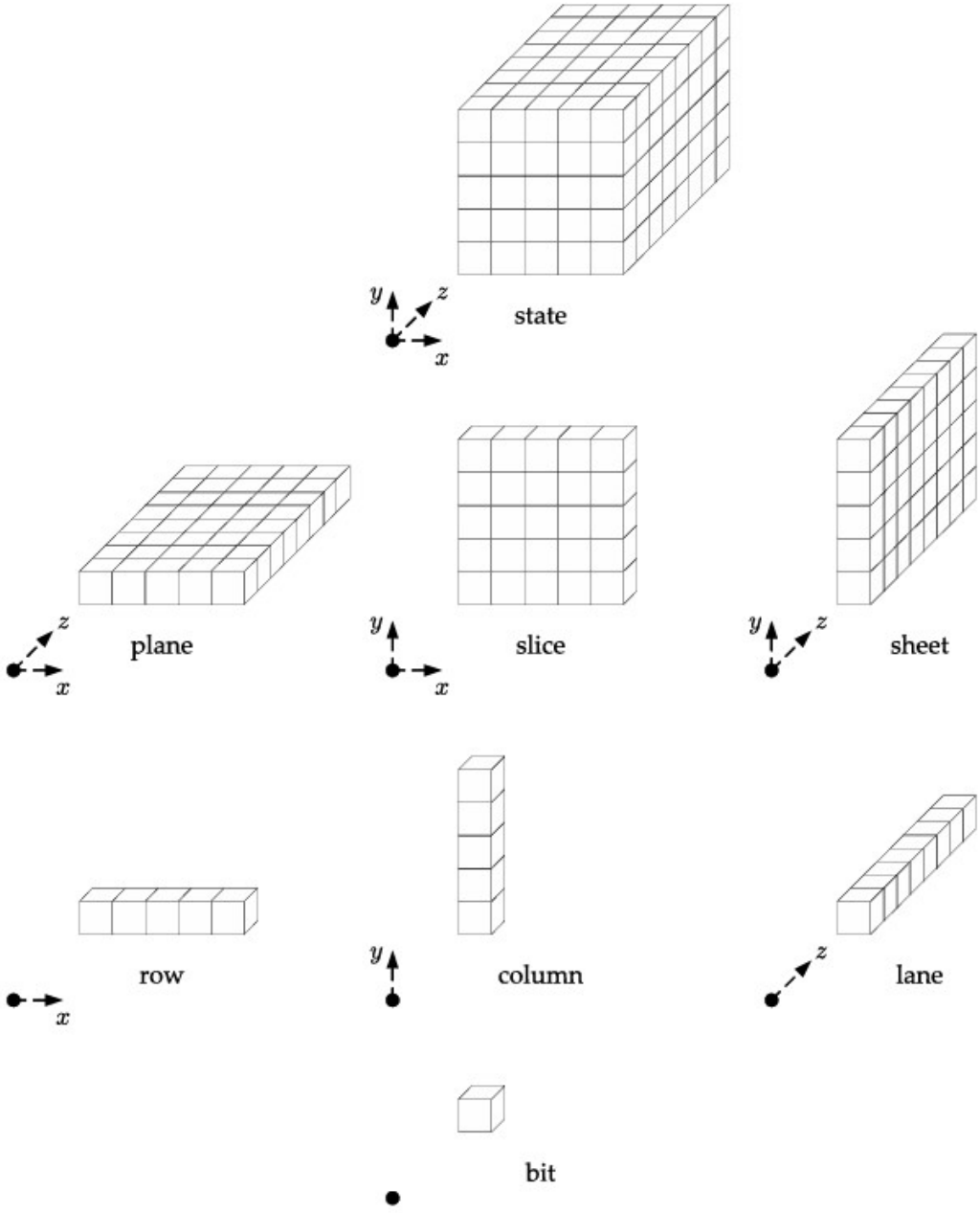

Figure 3: Parts of the SHA-3 State Array [2]

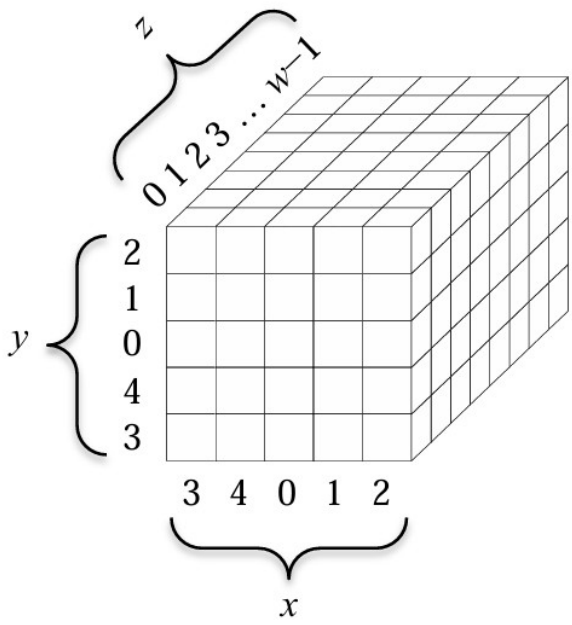

Figure 4: Labeling Convention for the SHA-3 State Array [2]

8.2.1 Padding

8.2.2 Algorithm 1: pad $10*1$(x,m)

Input:
positive integer x
non-negative integer m
Output:
string P such that $m + len(P)$ is a positive multiple of x

Steps:

1. Let $j = (-m - 2) \bmod x$.
2. Return $P = 1 \| 0^j \| 1$.

8.2.3 Step Mappings

Generally, the calculation of SHA-3 consists of five different algorithms θ, ρ, π, χ, and λ, which will be discussed separately in the following subsections.

8.2.4 Algorithm 2: $\theta(A)$

Input:
state array A
Output:
state array A'

Steps:

1. For all pairs (x, z) such that $0 \leq x < 5$ and $0 \leq z < \mathsf{w}$ let $C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$.

2. For all pairs (x, z) such that $0 \leq x < 5$ and $0 \leq z < \mathsf{w}$ let $D[x, z] = C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod \mathsf{w}]$.

3. For all triples (x, y, z) such that $0 \leq x < 5$ and $0 \leq z < \mathsf{w}$ let $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$

8.2.5 Algorithm 3: $\rho(A)$

Input:
state array A
Output:
state array A'

Steps:

1. For all z such that $0 \leq z < \mathsf{w}$ let $A'[0, 0, z] = A[0, 0, z]$.

2. Let $(x, y) = (1, 0)$.

3. For t from 0 to 23:

   (a) for all z such that $0 \leq z < \mathsf{w}$ let $A'[x, y, z] = A[x, y, (z - (t+1)(t+2)/2) \bmod \mathsf{w}]$

   (b) let $(x, y) = (y, (2x + 3y) \bmod 5)$

4. Return A'.

8.2.6 Algorithm 4: $\pi(A)$

Input:
state array A
Output:
state array A'

Steps:

1. For all triples (x, y, z) such that $0 \leq x < 5, 0 \leq y < 5,$ and $0 \leq z < \mathsf{w}$ let $A'[x, y, z] = A[(x + 3y) \bmod 5, x, z]$.

2. Return A'.

8.2.7 Algorithm 5: $\chi(A)$

Input:
state array A
Output:
state array A'

Steps:

1. For all triples (x, y, z) such that $0 \leq x < 5, 0 \leq y < 5,$ and $0 \leq z < \mathsf{w}$ let $A'[x, y, z] = A[x, y, z] \oplus ((A[(x + 1) \bmod 5, y, z] \oplus 1) \wedge A[(x + 2) \bmod 5, y, z])$.

2. Return A'.

8.2.8 Algorithm 6: $rc(t)$

Input:
integer t
Output:
bit $rc(\mathsf{t})$

Steps:

1. If $\mathsf{t} \bmod 225 = 0$: return 1.
2. Let $R = \mathsf{10000000}$.
3. For $i = 0$ to $\mathsf{t} \bmod 225$:
   (a) $R = 0 \| R$
   (b) $R[0] = R[0] \oplus R[8]$
   (c) $R[4] = R[4] \oplus R[8]$
   (d) $R[5] = R[5] \oplus R[8]$
   (e) $R[6] = R[6] \oplus R[8]$
   (f) $R = \text{Trunc}_8[R]$
4. Return $R[0]$.

It should be noted here that, technically, algorithm 6 is not a step mapping itself but rather a prerequisite to the step mapping ι, which will be described below.

8.2.9 Algorithm 7: $\iota(A, i_r)$

Input:
state array A
round index i_r
Output:
state array A'

Steps:

1. For all triples (x, y, z) such that $0 \leq x < 5, 0 \leq y < 5,$ and $0 \leq z < \mathsf{w}$ let $A'[x, y, z] = A[x, y, z]$.
2. Let $RC = 0^{\mathsf{w}}$.
3. For $j = 0$ to l let $RC[2j - 1] = rc(j + 7i_r)$.
4. For all z such that $0 \leq z < \mathsf{w}$ let $A'[0, 0, z] = A'[0, 0, z] \oplus RC[z]$.
5. Return A'.

8.2.10 KECCAK-$p[b, n_r]$

Finally, the standard variant of SHA-3, KECCAK-$p[b, n_r]$, consists of five step mappings used in the so-called round function Rnd which is defined as follows:

$$\text{Rnd}(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r).$$

8.2.11 Algorithm 8: KECCAK-$p[\mathsf{b}, n_r](S)$

Input:
string S of length b
number of rounds n_r
Output:
string S' of length b

Steps:

1. Convert S into a state array A.
2. For $i_r = 12 + 2\mathsf{l} - n_r$ to $12 + 2\mathsf{l} - 1$ let $A = \mathrm{Rnd}(A, i_r)$.
3. Convert A back into a string S'.
4. Return S'.

8.2.12 KECCAK-f

A specialized variant of SHA-3 is KECCAK-f a specified variant of the KECCAK-p family where $n_r = 12 + 2\mathsf{l}$. Therefore,

$$\text{KECCAK-}f[b] = \text{KECCAK-}p[\mathsf{b}, 12 + 2\mathsf{l}].$$

KECCAK-f rounds are indexed with 0 to $11 + 2\mathsf{l}$.

For example, KECCAK-$p[1600, 24]$ is equivalent to KECCAK-$p[1600]$. Accordingly, due to the indexing described in step 2 of algorithm 7, the rounds of KECCAK-$p[\mathsf{b}, n_r]$ exactly match the last rounds of KECCAK-$f[\mathsf{b}]$ and vice versa. In such cases, the preceding rounds are indexed by negative integers up to 0.

8.2.13 Sponge Functions

The term "*sponge construction*" refers to the process of constructing a function that operates on binary input and outputs a hash of arbitrary length using three particular components, namely:

1. an underlying function f, that operates on strings of a fixed length,
2. the parameter rate r, and
3. a padding rule pad.

Using this construction, the resulting function SPONGE[f, pad, r] is called a sponge function. Each sponge function takes two variables as input, a bit string N and the desired output length d, This is denoted in the following way

$$\text{SPONGE}[f, \text{pad}, \mathsf{r}](N, \mathsf{d}).$$

The construction is called a sponge function because it "absorbs" and "squeezes out" an arbitrary number of bits, which are stored in the state of the function.

8.2.14 Algorithm 9: SPONGE[f,pad,r](N, d)

Input:
string N
non-negative integer d
Output:
string Z such that len(Z)= d

Steps:

1. Let $P = \mathsf{N} \| pad(\mathsf{r}, \text{len}(\mathsf{N}))$.
2. Let $N = \text{len}(P)/\mathsf{r}$.
3. Let $\mathsf{c} = \mathsf{b} - \mathsf{r}$.
4. Let $P_0, ..., P_{N-1}$ be the unique sequence of stings of length r such that $P = P_0 \| ... \| P_{N-1}$.
5. Let $S = 0^{\mathsf{b}}$.
6. For i from 0 to $n-1$ let $S = f(S \oplus (P_i \| 0^{\mathsf{c}}))$.
7. Let Z be the empty string.
8. Let $Z = Z \| \text{Trunc}_{\mathsf{r}}(S)$.
9. If $\mathsf{d} \leq |Z|$, then return $\text{Trunc}_{\mathsf{d}}(Z)$; else continue.
10. Let $S = f(S)$ and continue with step 8.

The computation of SPONGE[f,pad,r](N, d) is halted as soon as the desired number of output bits has been produced.

9 Examples

Despite their vulnerabilities, some older algorithms are still in use for non-cryptographic applications. The following sections will discuss the current use cases for insecure algorithms as well as secure algorithms. Additionally, there will be an extensive example for each algorithm. The representation of the message, buffers, and individual computations will alternate between string-, bit-, array-, and hex representation. Furthermore, leading zeros on the left may be truncated in bit-representations of incomplete messages.

The example of MD2 will be left out, since MD2 is not used in any official capacities at the current time, and therefore has become obsolete.

All examples are computed using an implementation in Python following the specifications of the sources cited along their definitions. All code files are provided as an appendix to this work. For SHA-3 the code closely resembles the implementation of [29] which was used as an orientation.

9.1 MD4

MD4 was proven to be vulnerable early on and it has been considered insecure from 1996 onwards. [30] officially designated MD4 historic in 2004. However, its significance faded years earlier. Therefore, nowadays MD4 has nearly vanished from use. However, there are still some older file-sharing applications such as *eMule* that use the eD2k hashing protocol which uses the MD4 algorithm. This protocol uses MD4 to compute a hash directly from the complete file being stored or shared, which in turn is used as a file identifier. This process cannot be considered secure against any attack. It can however be used for efficient data storage and retrieval using the MD4 hash as a unique identifier for any file type. [5] [12]

For this example, the easiest file format, a text file, will be considered. Since a text file only consists of the text it encompasses, this aims to show the concept of computing the hash in an accessible way. For the following example, the modern version of MD4 with big-endian buffer representation is used, without the extension.

9.1.1 Input

To hash our example text, the simple text-string *"This is an example for MD4."*, the message first needs to be translated into binary for it to be operated on by the hashing algorithm. The binary representation of our example text is
01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001 01101110 00100000 01100101 01111000 01100001 01101101 01110000 01101100 01100101 00100000 01100110 01101111 01110010 00100000 01001101 01000100 00110100 00101110.

9.1.2 Padding

To pad this message, we start by adding a 1-bit, followed by $k = 231$ 0-bits, as well as the length of the message, 216, which can be represented in binary as the number `11011000 00000000 00000000 00000000 00000000 00000000 00000000 00000000`. It should be noted here that we are still using the little-endian convention for MD4, meaning the least-significant bit is at the left-most position and that the remaining 64 bits are filled with zeros.

Adding all of these padding parts together and appending them to the original binary string, the state is now 10101000 11010000 11010010 11100110 01000000 11010010 11100110

01000000 11000010 11011100 01000000 11001010 11110000 11000010 11011010 11100000
11011000 11001010 01000000 11001100 11011110 11100100 01000000 10011010 10001000
01101000 01011101 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001 10110000 00000000 00000000 00000000 00000000
00000000 00000000 0000000

9.1.3 Computation

After parsing the padded message into the algorithm and initializing our buffers, which are now used in the high-order first convention, we can begin to compute block by block. Since our message is shorter than 512 bits, and therefore has been padded to exactly 512 bits, we only need to compute one block.

For each round, the buffers are computed using the algorithm's defined equations. The first equation of round one will be done as an example.

$$a = (a + f(b, c, d) + \texttt{10101000 11010000 11010010 11100110}) << 3$$

which results in $a = \texttt{10011011010010110100001010011011}$ or $a = \texttt{9b4b429b}$ in hex representation.
Similarly, the other buffers result in $b = \texttt{efcdab89}, c = \texttt{39afd549}$, and $d = \texttt{4fae41de}$ respectively after the first steps of round 1.

Computing the rest of rounds 1 to 3, the final results are as follows.

$$a = \texttt{475adc38}, b = \texttt{551259b}, c = \texttt{21b66634}, \text{ and } d = \texttt{32cedad5}.$$

Lastly, the final additions can be computed as follows.

$$\begin{aligned}
&\texttt{10001110101101011011100000111000} + \texttt{11001110100010100100011000000001} \\
&= \texttt{10101110100111111111111100111001} = a \\
&\texttt{11000101010100010010010110011011} + \texttt{11101111110011011010101110001001} \\
&= \texttt{10101000110110000110111110101111} = b \\
&\texttt{10000110110110011001100011010100} + \texttt{10011000101110101101110011111110} \\
&= \texttt{10111010011100010100001100110010} = c \\
&\texttt{11001011001110110110101101010101} + \texttt{10000001100100101010001110110} \\
&= \texttt{10000110000000100101111010010111} = d
\end{aligned}$$

Representing these as hexadecimals and concatenating them one after another we get the final hash value of

$$a||b||c||d = \texttt{39ff 9fae af6f d8a8 3243 71ba 4b2f 0143}$$

9.2 MD5

Similar to MD4, MD5 has been considered insecure for several years now. Following initial attacks and proof of vulnerabilities, it was officially declared "cryptographically broken" in 2008. However, some databases and applications still use MD5 in hashing their passwords, even though it is highly discouraged. There are also still instances of MD5 being used in verifying and authenticating digital signatures and file integrity checks, which is not a secure method, since colliding messages can be produced reliably nowadays. [21]

Another common use for MD5 is checking data or file integrity against accidental corruption. This is still considered viable since the algorithm outputs fairly evenly distributed hashes, keeping the chance of an accidental collision slim. A third use case for MD5 that is still acceptable is using it for load balancing or de-duplication systems. While load-balancing describes the process of storing information balanced over different servers in order to gain efficiency, the de-duplication system merely aims to eliminate redundant data to improve efficiency. In both cases, MD5 hashes of stored data are used to efficiently search for and compare data as MD5 still remains considerably easier and faster to compute than newer algorithms. [14]

9.2.1 Input

Since the simplest data to be hashed in such a case is the text file, the following example will once again use text since this enables us to visualize the changes more easily. Our MD5 example message will be *"Let's try another text for MD5."* or 01001100 01100101 01110100 00100111 01110011 00100000 01110100 01110010 01111001 00100000 01100001 01101110 01101111 01110100 01101000 01100101 01110010 00100000 01110100 01100101 01111000 01110100 00100000 01100110 01101111 01110010 00100000 01001101 01000100 00110101 00101110 in binary.

9.2.2 Padding

Since padding works the same as for MD4, we can directly formulate the padded message as: 01001100 01100101 01110100 00100111 01110011 00100000 01110100 01110010 01111001 00100000 01100001 01101110 01101111 01110100 01101000 01100101 01110010 00100000 01110100 01100101 01111000 01110100 00100000 01100110 01101111 01110010 00100000 01001101 01000100 00110101 00101110 10000000 00000000 11111000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

9.2.3 Computation

Similar to MD4, we use the buffers in the now-common big-endian convention and only need to compute one 16-word block. Again, the first computation of round 1 is given as an example.

$$a = b + ((a + f(b, c, d) + h_0 + T[1] \ll 7) = \texttt{10101111101010010100011010001000}$$

Equivalently, the other variables are computed until their last step resulting to

$$
\begin{aligned}
a &= 10000110011011011111001110000101010011 \\
b &= 10001001000000110110001111000001010101 \\
c &= 10001000100101011010101001110100100011 \\
d &= 10000111111001010001111111000101010000
\end{aligned}
$$

Lastly, the final additions can be computed as follows.

$$
\begin{aligned}
&10000110011011011111001110000101010011 + 11001110100010100100011000000001 \\
&= 10110000100000010001010100 = a \\
&10001001000000110110001111000001010101 + 11101111110011011010101110001001 \\
&= 11000010100110100110111111011110 = b \\
&10001000100101011010101001110100100011 + 10011000101110101101110011111110 \\
&= 10111110001001010111101000100001 = c \\
&10000111111001010001111111000101010000 + 10000001100100101010001110110 \\
&= 10010111101001000010111000110 = d
\end{aligned}
$$

Representing these as hexadecimals and concatenating them after another we get the final hash value of

$$a||b||c||d = \texttt{5404 c202 de9b a630 217a 25be c645 7a09}$$

9.3 SHA-1

SHA-1, the first cryptographic hash standardized by NIST, was declared insecure in 2011. [20] However, in contrast to its predecessors, its vulnerabilities are not as easily exploitable, as can be seen in the proof provided in chapter 5. Therefore, SHA-1 is still widely used in different applications and systems. However, the limitations in cryptographic safety should be taken into account when choosing a hashing algorithm. Since SHA-1 is insecure against targeted attacks, larger companies that are targeted frequently, such as Google, Microsoft, or Apple, as well as government and financial institutions have stopped using SHA-1 years ago. [13]

Smaller companies, as well as lesser-known online services and internal data and security management systems of certain organizations, however, still use SHA-1 in a variety of applications. Similar to MD4 and MD5, SHA-1 is used for data and file integrity checks to ensure that no changes have been made to the given data. In the case of SHA-1, the data checked by this method can range from stored files to online content prone to interception, such as communication. [9]

Some software vendors also use the SHA-1 hash as an integrity check, frequently publishing it alongside installation downloads or update files to help protect against malicious code being embedded in their software. Users can compare the hash provided alongside the program to the hash computed directly from the installation file before using it. In some instances, especially during updates or internal version control, systems perform this step automatically instead of requiring the user to do it manually. [9]

Other uses of SHA-1 today include digital signatures, which aim to confirm the legitimacy of digital messages as well as digital documents. However, governmental digital signatures, which have become frequent in official documents, already use higher security standards, leaving SHA-1 limited to certain applications. [9]

One rather famous example of a service that used SHA-1 until recently is Git, which used it to distinguish changes in objects more efficiently. Since the most common use now, however, is simply data integrity checks across different files, the example will focus again on the simplest type of file, a text file. Similar to MD4 and MD5, only the text itself, without any metadata, is hashed. [10] [9]

9.3.1 Input

For the purposes of this example, the text to be hashed will be *"This is another example sentence for SHA-1."* or 01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001 01101110 01101111 01110100 01101000 01100101 01110010 00100000 01100101 01111000 01100001 01101101 01110000 01101100 01100101 00100000 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101 00100000 01100110 01101111 01110010 00100000 01010011 01001000 01000001 00101101 00110001 00101110 in binary.

9.3.2 Padding

Since padding works identically to MD4 and MD5, we can already determine the padded message as *10101000 11010000 11010010 11100110* 01000000 11010010 11100110 01000000 *11000010 11011100 11011110 11101000* 11010000 11001010 11100100 01000000 11001010

11110000 11000010 11011010 11100000 11011000 11001010 01000000 11100110 11001010 11011100 11101000 11001010 11011100 11000110 11001010 *01000000 11001100 11011110 11100100* 01000000 10100110 10010000 10000010 01011010 01100010 01011101 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 *00000000 00000000 00000000 00000000* 00000000 00000000 00000000 00000000 00000000 00000000 00000010 1011000.

Notably the length l of the message at the end of this padding step is appended in the big-endian convention, marking the only difference from our previous examples. Setting the initial hash values is identical to the step in MD4 and MD5.

9.3.3 Computation

As in the examples given so far, we only hash a message of one block, so we only need to use the individual steps once. Since our example sentence consists of only one block, and therefore exactly sixteen 32-bit words, the first part of preparing the message schedule only consists of feeding our padded sentence into the first sixteen words of the schedule. For inputting the remaining 64 words, the first, $t = 16$, will be done as an example.

Using $W_{16} = ROTL^1(W_{13} \oplus W_8 \oplus W_2 \oplus W_0)$, we can search the right words, which are highlighted in the padded sentence above, use bitwise XOR and rotate the resulting number by 1 and thereby compute $W_t = 10101011000000110101001011101010$. Filling in each word into the message schedule, the final result is given in hex representation below for better readability.

W_t = 54686973 20697320 616e6f74 68657220 6578616d 706c6520 73656e74 656e6365 20666f72 20534841 2d312e80 0 0 0 0 158 2ac0d2ea d0be9282 524e4382 65938bd4 8d473b37 8e988b8e 6d2115a4 90f42297 521e312d 3b999ecf df169f2b 6f1b75f2 6dbd4bf1 a31c2bfa 51f56578 5beff578 1319177f be83c5c3 b7e025cc 2e12c5af 66b14142 15214e5e 81b108bc 2c6610c5 1661ee4f d7604c84 525bc3a6 e8e8ebd1 1b32469f 7f12a6f5 576b22d4 a470cdc3 9b14f413 2134dc78 4ef4d5d6 919f2866 ba0da633 11425b70 a249d882 caf66f63 9cd90515 79eb475f 9ad67f19 3578c0af 1f7f0a4f a1ec9ebe b69d9dd4 a19ae92e d1ab74dd ffba5d2a 9f6bcba5 c81d8ec8 f10253a9 ca66c211 80219d16 c70b2577 3b859980 66108f98 bb933709 ce1892b2 7de09754 e3070510 521fcd5a c9960c4f.

Since the initial hash values have already been initialized, we only need to copy them into the buffers. In step 3, all computations have to be performed 80 times again. As an example, the first round of this loop will be computed before discussing the final output of this step. Once again, the results are shown in hex, which demonstrates that some of the hash values are not changed but rather shifted to the next value, since some of the initial hash values are still part of this first round. These values are highlighted below. Note also that a and T are the same in every step as T is merely used as a temporary variable transitioning values to the next round.

$$
\begin{aligned}
T = ROTL^5(a) + f_t(b,c,d) + e + K_t + W_t = &\ \texttt{f41d0226} \\
e = d = &\ \texttt{10325476} \\
d = c = &\ \texttt{98badcfe} \\
c = ROTL^{30}(b) = &\ \texttt{7bf36ae2} \\
b = a = &\ \texttt{67452301} \\
a = T = &\ \texttt{f41d0226}
\end{aligned}
$$

Repeating this step for 80 rounds, the final values are

$$\begin{aligned} T &= \texttt{fbea7e4f} \\ e &= \texttt{a4f9be66} \\ d &= \texttt{2338f187} \\ c &= \texttt{202e3a2e} \\ b &= \texttt{23d917cc} \\ a &= \texttt{fbea7e4f}. \end{aligned}$$

Finally, intermediate hash values are computed as follows.

$$\begin{aligned} H_0^{(i)} &= \texttt{fbea7e4f} + \texttt{67452301} = \texttt{632fa150} \\ H_1^{(i)} &= \texttt{23d917cc} + \texttt{efcdab89} = \texttt{13a6c355} \\ H_2^{(i)} &= \texttt{202e3a2e} + \texttt{98badcfe} = \texttt{b8e9172c} \\ H_3^{(i)} &= \texttt{2338f187} + \texttt{10325476} = \texttt{336b45fd} \\ H_4^{(i)} &= \texttt{a4f9be66} + \texttt{c3d2e1f0} = \texttt{68cca056} \end{aligned}$$

Therefore, concatenating these five values, the resulting hash for the example sentence *"This is another example sentence for SHA-1."* is `632fa150 13a6c355 b8e9172c 336b45fd 68cca056`.

9.4 SHA-2

Like all previously mentioned hash algorithms, SHA-2 is used in different fields, such as data and file integrity checks. Moreover, like SHA-1, it is used in digital signatures to validate the integrity of messages and documents. In the case of SHA-2, this includes governmental or financial data. Furthermore, since there is no known viable attack on this hashing algorithm, it is also frequently used for handling other sensitive data, such as hashing passwords or ensuring data integrity and security in blockchain applications such as *Bitcoin*. [25]

Since SHA-2 might currently be the most widely used hashing algorithm, there are a number of diverse examples that could be used to demonstrate this algorithm. However, the most significant differences from SHA-1 include its use for more sensitive data, its use in blockchain applications, and its use in cryptographic protocols, such as SSL, or its newer variant TLS, and related techniques like HMAC, which all aim to secure online data transfer and messaging. [25]

It should be noted that in many cryptographic applications of SHA-2 the hashing process is only the first step in encrypting data, often followed by other cryptographic functions and encryption methods such as RSA. It should furthermore be noted that the example computed in the following section will be the variant SHA-512.

9.4.1 Input

The most straightforward example of cryptographic uses of SHA-2 is probably the hashing of passwords since the data hashed is short and not highly complex in itself. Therefore, the following example will demonstrate the use of SHA-2 for hashing a common password. It should be noted that passwords are not usually hashed on their own, but are commonly combined with a so-called *salt* first. A salt is a random string of symbols, usually of a predefined length, that is appended to the password before hashing and stored alongside the hash. This provides another layer of security against brute force and pre-computed tables of common passwords. The way in which salts are implemented into the passwords varies across different applications. However, in this example, the salt is simply appended to the password, which is a common and simple way to use salts. [3]

To create a realistic example, the input used in this example will be one of the most used passwords worldwide: *"Password"*. Hashing without a salt would result in the same hash for everyone using this password, which would render the user virtually unprotected from databases of precomputed common passwords. Therefore, we will append the random salt `"*l!74p927oy1"` to the password, which results in the binary representation 01010000 01100001 01110011 01110011 01110111 01101111 01110010 01100100 00101010 01101100 00100001 00110111 00110100 01110000 00111001 00110010 00110111 01101111 01111001 00110001.

9.4.2 Padding

In our first step, since padding is identical to our previous algorithm SHA-1 except for its length, we can directly compute the padded binary representation of the string *"Password*l!74p927oy1"* to be
10100000 11000010 11100110 11100110 11101110 11011110 11100100 11001000 01010100 11011000 01000010 01101110 01101000 11100000 01110010 01100100 01101110 11011110 11110010 01100011 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 *00000000 00000000 00000000 00000000 00000000*
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000001 0100000.

The only differences to SHA-1 are the length of the padded message, which is 512 instead of 256 bits, and the number of bits representing the length at the end which is now fixed at 128.

9.4.3 Computation

Once again we initialize our hash values, which differ from previous algorithms, and prepare our message schedule by inputting our sixteen words, as we again have only one block. This time, however, fitting in words of length 64 bits, the resulting message schedule will be twice as long. We start by computing word 16 of the message schedule as an example, the words required for this computation are highlighted in the padded text above.

$$W_{16} = \sigma_1^{\{512\}}(W_{14}) + W_9 + \sigma_0^{\{512\}}(W_1) + W_0 = \texttt{50617373776f7264}$$

Note that the values are given in hex representation for better readability. The following computations will use the same representation. Filling in words 17 to 79 in the same way, the resulting message schedule is as follows.

W_t = 50617373776f7264 2a6c213734703932 376f793180000000 0 0 0 0 0 0 0 0 0 0 0 0 a0
77aa186c3ad3ff36 466e2e4ac6f03e34 fad81f26a5793115 35ae4222d9536984 f3092612b2b8b8c4
c0942eab89ba5aad 8c9d8a6cfbcd7333 c4a0b660a0f54c69 f268328a3b9715d 8a72286f395ed376
8cd164d04873e4de 7edf80a8e5d6c096 d8b7e136b7df3b16 bbb33f0b7901b63e 865fc38838f5bf5b
6255918cee8d5a4 a125f9b74fa50d43 e829d83408154e96 c244a21eb6324aff b6346782ab4adad4
533c0a79c30b9ace 23f0d12a67a6b0bb 8d607f13a71c32f 9670892a76120d8 a8ee6438b14d43fb
55bb65eed4f95b66 daa25a7123ac17b7 d52ebc31193f3cea b34c9eda5331ef2b abea4a5b11c2735b
dc2c424291972866 551bdeafa958747d e12c6eb08b684ee6 89360033343810fd 296ee5d4b684014f
9a77b03f17539cef f5be3eda5c8a4e61 4dd939d4387a98d8 22854490131e7c60 56071bf0f84e91a5
59741987aeb493e9 3dbd80c2711c04fd 4fba700302e34ca9 2be5ed80ee734ac8 a5a734d146918f5f
ebbe3a081eda3aa 27eaf0554549a0bf 7d5ef80dd5563e9 62ca1722ed172004 478a3d3cb1e33bb7
6cc53138117087ea 3519ec1ed8ffad98 7a1ceab80cd4b532 44915994b89b9fb3 80fe220e79657686
5491a54f021b4ae0 afb896891544e4d5 85140f0b1f7eead2 464418fe6e914292 2a2ec43522374408
b2f26bb2aed1b28d 75b6ece576727e05 8a6841fb2872c16c d0f1ede1002d893d

Similar to SHA-1, we copy the initial hash values into the buffers. As an example, the first round of step 3 will be computed before proceeding with the final result of this loop. Using our equations in the first round, we get the following. Note again that many of the buffer values remain unchanged at this point. These values are highlighted, just as in the previous example.

$$
\begin{aligned}
T_1 = h + \Sigma_1^{\{512\}}(e) + Ch(e,f,g) + K_t^{\{512\}} + W_t = &\quad \texttt{a27a1ced93075b04}\\
T_2 = \Sigma_0^{\{512\}}(a) + Maj(a,b,c) = &\quad \texttt{4334c1bea164f555}\\
h = g = &\quad \textit{1f83d9abfb41bd6b}\\
g = f = &\quad \textit{9b05688c2b3e6c1f}\\
f = e = &\quad \textit{510e527fade682d1}\\
e = d + T_1 = &\quad \texttt{47ca1227f22491f5}\\
d = c = &\quad \textit{3c6ef372fe94f82b}\\
c = b = &\quad \textit{bb67ae8584caa73b}\\
b = a = &\quad \textit{6a09e667f3bcc908}\\
a = T_1 + T_2 = &\quad \texttt{e5aedeac346c5059}
\end{aligned}
$$

Results after the final step of this stage are

$$
\begin{aligned}
T_1 &= \texttt{84481c7bf2025cc0}\\
T_2 &= \texttt{d23cb624f3d7c26d}\\
h &= \texttt{a06e17d155201f27}\\
g &= \texttt{8ca72925636277e8}\\
f &= \texttt{23ba9f7ef5e0a2f3}\\
e &= \texttt{751b8fba78b01e67}\\
d &= \texttt{7bd0ff300766b843}\\
c &= \texttt{6864bbc515dbe7}\\
b &= \texttt{73cd62a6ab9c4666}\\
a &= \texttt{5684d2a0e5da1f2d}.
\end{aligned}
$$

Finally, using values to update with the intermediate hash values, we get the following.

$$
\begin{aligned}
H_0^{(i)} &= \texttt{5684d2a0e5da1f2d} + \texttt{6a09e667f3bcc908} = \texttt{c08eb908d996e835}\\
H_1^{(i)} &= \texttt{73cd62a6ab9c4666} + \texttt{bb67ae8584caa73b} = \texttt{2f35112c3066eda1}\\
H_2^{(i)} &= \texttt{006864bbc515dbe7} + \texttt{3c6ef372fe94f82b} = \texttt{3cd7582ec3aad412}\\
H_3^{(i)} &= \texttt{7bd0ff300766b843} + \texttt{a54ff53a5f1d36f1} = \texttt{2120f46a6683ef34}\\
H_4^{(i)} &= \texttt{751b8fba78b01e67} + \texttt{510e527fade682d1} = \texttt{c629e23a2696a138}\\
H_5^{(i)} &= \texttt{23ba9f7ef5e0a2f3} + \texttt{9b05688c2b3e6c1f} = \texttt{bec0080b211f0f12}\\
H_6^{(i)} &= \texttt{8ca72925636277e8} + \texttt{1f83d9abfb41bd6b} = \texttt{ac2b02d15ea43553}\\
H_7^{(i)} &= \texttt{a06e17d155201f27} + \texttt{5be0cd19137e2179} = \texttt{fc4ee4ea689e40a0}
\end{aligned}
$$

Lastly, concatenating them one after another we obtain our final hash value c08eb908d996e835 2f35112c3066eda1 3cd7582ec3aad412 2120f46a6683ef34 c629e23a2696a138 bec0080b211f0f12 ac2b02d15ea43553 fc4ee4ea689e40a0.

9.5 SHA-3

Similar to its predecessors, SHA-3 can be used in a variety of different applications, such as checking and verifying data and file integrity in storage and data transfers, as well as digital signatures and certificates. However, since currently there are no known attacks on SHA-3 and it is therefore considered the most secure hashing algorithm developed so far, SHA-3 is the preferred choice for verifying and signing sensitive data as well as cryptographic applications. SHA-3 is already adopted in certain official contexts to ensure the highest possible security for sensitive data. [24]

Again, it is important to note that in cryptographic applications, all hashing algorithms might be combined with other cryptographic measures to maximize security.

One famous example of a blockchain application using SHA-3 is *Ethereum*. *Ethereum* does not use the standardized variant of SHA-3, but instead employs the original *KECCAK*-256 version submitted to the *NIST*-contest, which differs in the padding step. [8] However, since the definition given in the previous chapter is the now standardized version, we will use SHA3-256 in the example. Other applications use different variants and this provides for better comparability to previous examples. Therefore, the output will be different from the real output used in *Ethereum* due to slight changes in the variables used. [8]

9.5.1 Input

However, in order to effectively demonstrate a realistic application of SHA-3 being used to hash data, the example input will follow a common *Ethereum* transactions. All features of the message are as specified in [7]. Note that all samples and computations will be given in hex representation, as this is standard for this kind of application.

According to *Ethereum* specifications our sample input consists of the following 9 parts. The sample code can be created using [11]

1. **Nonce**: A unique number representing the count of transactions sent from this specific adress. In our sample the nonce will be `04`.
2. **Gas Price**: Amount of gas the user is willing to pay per unit of gas transferred. In our sample the gas price will be `85174876e800`.
3. **Gas Limit**: The maximum amount of gas units that can be consumed by the transaction. We will use a gas limit of `830186a0`.
4. **To**: The recipient's Ethereum adress. The adress this example transaction will be sent to is `94a0b86991c6218b36c1d19d4a2e9eb0ce3606eb48`.
5. **Value**: The amount of *ETH* transferred in this transaction. The value sent here will be `80`.
6. **Data**: Optional field that can be used to include information for specific contracts. In this sample, it calls a specific function of a certain contract group and therefore is b844 a905 9cbb 0000 0000 0000 0000 0000 0000 b5e5 d0f8 c0cb a267 cd3d 7035 d6ad c8eb a7df 7cfd 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 06f0 5b59 d3b2 0000.
7. **Recovery Identifier**: A sender ID. Our sample ID is `a0`.

8. **Signature**: The signiture is composed of two halves and is computed from identifying information of the sender, such as the sender's Ethereum adress. The signature of our sample transaction is 48b5 5bfa 915a c795 c431 978d 8a6a 992b 628d 557d a5ff 759b 307d 495a 3664 9353 1fff d310 ac74 3f37 1de3 b9f7 f9cb 56c0 b28a d436 01b4 ab94 9f53 faa0 7bd2 c804.

We can now take these individual parts and create one complete string from them. However, it should be noted here that data in Ethereum is usually encoded according to RLP standards, which has already been done using [23]. Therefore, the RLP list prefix must be added to the front of our sample. [6]

f8ab 0485 1748 76e8 0083 0186 a094 a0b8 6991 c621 8b36 c1d1 9d4a 2e9e b0ce 3606 eb48 80b8 44a9 059c bb00 0000 0000 0000 0000 0000 00b5 e5d0 f8c0 cba2 67cd 3d70 35d6 adc8 eba7 df7c fd00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0006 f05b 59d3 b200 0025 a0a0 48b5 5bfa 915a c795 c431 978d 8a6a 992b 628d 557d a5ff 759b 307d 495a 3664 9353 a0a0 1fff d310 ac74 3f37 1de3 b9f7 f9cb 56c0 b28a d436 01b4 ab94 9f53 faa0 7bd2 c804

Before computing the final example note that KECCAK specification defines the algorithm to primarily work with bytes, instead of individual bits and to use little-endian convention within bytes. Even though NIST does not explicitly state this in their specifications, this convention will be used in the example below, as this version is commonly implemented in instances worldwide. For better comparability, we therefore stick to this convention.

9.5.2 Padding

As with the other algorithms, we start with the padding step. To compute SHA3-256, we use a width $\mathsf{b} = 1600$ and a capacity of $\mathsf{c} = 2\mathsf{d} = 512$. Therefore, our rate r used in padding is $r = 1088$ bits. To pad the 1392-bits message to a multiple of 1088, we need to append 784 bits in total. Starting with the suffix `01`, followed by `1`, a row of zeros filling up the message, and `1` at the end, we end up with $j = (-4 - 1392) \bmod 1088 = 780$ as the number of zeros appended. However, as previously mentioned, using little-endian byte representation, this translates to `06` for the suffix and the first padding byte, `80` for the final padding byte and a total of 96 zero-bytes in-between.

9.5.3 Feeding into the State Array

After splitting the resulting message into two blocks, we can XOR the first into the state array and process it through the whole KECCAK algorithm. The state array will be shown in sheets starting at index zero.

The input array look as follows.

```
1110100001110110010010000001011110000101000001001010101111111000
0000000000000000000000000000000000000000000000000000000010111011
0000000000000000000000000000000000000000000000000000000000000000
0111110101010101100011010110001000101011100110010110101010001010
0000000000000000000000000000000000000000000000000000000000000000

1011100010100000100101001010000010000110000000011000001100000000
1101000011100101101101010000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0101101001001001011111010011000010011011011101011111111110100101
```

```
0000000000000000000000000000000000000000000000000000000000000000

1101000111000001001101101000101100100001110001101001000101101001
0111000000111101110011010110011111010001011001011110000001111000
0000000010110010110100110101100101011011111100000000011000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

0000011000110110110011101011000010011110001011100100101010011101
0111110011011111101001111110101111001000101011011101011000110101
1111101001011011101101010100100010100000101000000010010100000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

1001110000000101101010010100010010111000100000000100100011101011
0000000000000000000000000000000000000000000000000000000011111101
1000110110010111001100011100010010010101110001110101101010010001
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

9.5.4 Theta Step

The Theta step shifts and reorders the bits, resulting in the following state array.

```
1001110111111100011010011011011110010010101010110100000000110101
0111010110001010001000011010000000010111101011111111010110111010
0111010110001010001000011010000000010111101011111111010111001101
0000100011011111101011001100001000111100001101101000000101000111
0111010110001010001000011010000000010111101011111111010111001101

0110111100011111000000001011111010011001011001101110110111101010
0000011101011010001000010001111000011111011001110110111011101010
1101011110111111100101000001111000011111011001110110111011101010
1000110111111011011101001001011101000010000010010100100010100111
1101011110111111100101000001111000011111011001110110111011101010

1110001010101000110100100011110011010000111101011001111010011101
0100001101010100001010011101000001010011111110001100111100001100
0011001111011011001101111111011101010101011000011000010011111010
0011001101101001111001001011011111110001001100110000111111110100
0011001101101001111001001011011111110001001100110000111111110100

1000010001011101110101110000010100011100010111010011100000000010
1111111010110100101111100101111001001010110111101010010010101010
0111100000110000101011001111110100100010110100110101011110011111
1000001001101011000110011011010110000010011100110111001010011111
1000001001101011000110011011010110000010011100110111001010011111

0011011011110000111111111011110000010011100110000111001011010000
1010101011111010101010110111111000101010110001100000111010110001 10
0010011101100010011001110011110000111110110111110110000010101010
```

```
1010101011110101010101101111100010101011000110000011101000111011
1010101011110101010101101111100010101011000110000011101000111011
```

9.5.5 Rho Step

After the Rho step shifts, the array looks such as below.

```
1001110111111100011010011011011110010010101010110100000000110101
0111101011111110101101110110011101011000101000100001101000000001
1010110001010001000011010000000010111101011111110101111001101011
0110110100000010100011100001000110111111010110011000010001111000
1000011010000000010111101011111110101111001101011101011000101000

1101111000111110000000010111110100110010110011011101101111010100
0111011011101110101000000111010110100010000100011110000111110110
1111111001010000011110000111110110011101101110111010101101011110
0101001000101001111100011011111011011101001001011101000010000010
0101111011111110010100000111100001111101100111011011101110101011

0111100010101010001101001000111100110100001111010110011110100111
1101010100001010011101000001010011111110001100111100001100010000
0001100001001111101000011001111011011001101111110111010101010110
1111001001011011111110001001100110000111111110100001100110110100
1000011001101101001111001001011011111110001001100110000111111110

0101000111000101110100111000000000101000010001011101110101110000
0101010101111111010110100101111100101111001001010110111101010010
1111101001000101101001101010111100111110111100000110000101011001
0011011010110000010011100110111001010011111100000100110101100011
1001111110000010011010110001100110110101100000100111001101110010

1110000010011100110000111001011010000001101101111000011111111101
0110111110001010101100011000001110101100011010101010111101010101
0110111110110000010101010001001110110001001100111001111000011111
1111010101010110111110001010101100011000001110100011101110101010
0101010110111110001010101100011000001110100011101110101010111101

0111011011101110101000000111010110100010000100011110000111110110
0110111110001010101100011000001110101100011010101010111101010101
1101010100001010011101000001010011111110001100111100001100010000
0111101011111110101101110110011101011000101000100001101000000001
```

9.5.6 Pi Step

The Pi step, which reorders the individual bits, results in the following array.

```
1001110111111100011010011011011110010010101010110100000000110101
0101000111000101110100111000000000101000010001011101110101110000
1101111000111110000000010111110100110010110011011101101111010100
1110000010011100110000111001011010000001101101111000011111111101
0111100010101010001101001000111100110100001111010110011110100111

0111011011101110101000000111010110100010000100011110000111110110
0110111110001010101100011000001110101100011010101010111101010101
1101010100001010011101000001010011111110001100111100001100010000
0111101011111110101101110110011101011000101000100001101000000001
```

```
0101010101111111010110100101111100101111001001010110111101010010

0001100001001111101000011001111011011001101111110111010101010110
1010110001010001000011010000000010111101011111110101111001101011
1111101001000101101001101010111100111110111100000110000101011001
1111111001010000011110000111111011001110110111011101010110101110
0110111110110000010101010001001110110001001100111001111000011111

0011011010110000010011100110111001010011111100000100110101100011
0101001000101001111100011011111011011101001001011101000010000010
1111010101010110111110001010101100011000001110100011101110101010
1111001001011011111110001001100110000111111110100001100110110100
0110110100000010100011100001000110111111010110011000010001111000

0101010110111110001010101100011000001110100011101110101010111101
1000011001101101001111001001011011111110001001100110000111111110
1000011010000000010111101011111110101111001101011101011000101000
1001111110000010011010110001100110110101100000100111001101110010
0101111011111110010100000111100001111101100111011011101110101011

0101010110111110001010101100011000001110100011101110101010111101
1000011001101101001111001001011011111110001001100110000111111110
1000011010000000010111101011111110101111001101011101011000101000
1001111110000010011010110001100110110101100000100111001101110010
0101111011111110010100000111100001111101100111011011101110101011

1001010111111101011010000011110111001011000001010101010000110101
1101000110010100110111111000000000111001010100001000110101011010
1111010001111011100000111101011000110010000011011111101110011101
0110010010011100100010111000111000000100101011100010011010100011
0101001000101010001100011000111110100100001011111111011110101010

0101000001011110111011100001010110100000010100011110100111010111
0011110110100010010000010011111011110110001101010001011111010101
1101000000011000001011000001010011111110001110011101100110110010
0111101011110101001101111110011101011010111000100000101010100001
0101010101111101110100000101111100100001011011010110111100110010

0101100101000001100000010001111011010101101100011101011111001010
0010100000010101000000010000000010011111011111010111111100010111
1111100011000101101000001011101110011001111110101101001010101001
1111001111010000011110110111111011010110110111011110010010011100
0111110101001100000001010111101111110001101101111010010110011100

1011111011110000000011110101111111000011110100010100110101100011
0000001110101001001100101011111011011101011001000100110010000010
1010110101101000111110011110101100001000111100100011001001111110
1001001001000111011110000001111110000111110011111001110100111001
0100110100000010101010101001011010111111011110011100000001111100

0011011110111100101010101000011000101110100111100100101101111111
1010100001100111000111001001010111101000001100010000111111011011
1000011110000000001010101011111101100011000001111101011000101000
```

9.5.7 Chi Step

Next, the Chi step performs another reordering, creating the following state array.

```
1001010111111101011010000011110111001011000001010101010000110101
1101000110010100110111111000000000111001010100001000110101011010
1111010001111011100000111101011000110010000011011111101110011101
0110010010011100100010111000111000000100101011100010011010100011
0101001000101010001100011000111110100100001011111111011110101010

0101000001011110111011100001010110100000010100011110100111010111
0011110110100010010000010011111011110110001101010001011111010101
1101000000011000001011000001010011111110001110011101100110110010
0111101011110101001101111110011101011010111000100000101010100001
0101010101111101110100000101111100100001011011010110111100110010

0101100101000001100000010001111011010101101100011101011111001010
0010100000010101000000010000000010011111011111010111111100010111
1111100011000101101000001011101110011001111110101101001010101001
1111001111010000011110110111111011010110110111011110010010011100
0111110101001100000001010111101111110001101101111010010110011100

1011111011110000000011110101111111000011110100010100110101100011
0000001110101001001100101011111011011101011001000100110010000010
1010110101101000111110011110101100001000111100100011001001111110
1001001001000111011110000001111110000111110011111001110100111001
0100110100000010101010101001011010111111011110011100000001111100

0011011110111100101010101000011000101110100111100100101101111111
1010100001100111000111001001010111101000001100010000111111011011
1000011110000000001010101011111101100011000001111101011000101000
```

```
1000010111100000010111110111100011101101100000100110101101110010
0101101110101011000110100010100001110110100111011011001111111011
```

9.5.8 Iota Step

Finally, after pre-computing the iota constants RC =0000000000000001 for the first round of SHA3-256, the final state array after the Iota step of the first round is as follows.

```
1001010111111101011010000011110111001011000001010101010000110100
1101000110010100110111111000000000111001010100001000110101011010
1111010001111011100000111101011000110010000011011111101110011101
0110010010011100100010111000111000000100101011100010011010100011
0101001000101010001100011000111110100100001011111111011110101010

0101000001011110111011100001010110100000010100011110100111010111
0011110110100010010000010011110111011000110101000101111111010101
1101000000011000001011000001010011111110001110011101100110110010
0111101011110101001101111110011101011010111000100000101010100001
0101010101111101110100000101111100100001011011010110111100110010

0101100101000001100000010001111011010101101100011101011111001010
0010100000010101000000010000000010011111011111010111111100010111
1111100011000101101000001011101110011001111101011010010101011001
1111001111010000011110110111110110101101110111011110010010011100
0111110101001100000001010111101111110001101101111010010110011100

1011111011110000000011110101111111000011110100010100110101100011
0000001110101001001100101011111011011101011001000100110010000010
1010110101101000111110011110101100001000111100100011001001111110
1001001001000111011110000001111110000111110011111001110100111001
0100110100000010101010101001011010111111011110011100000001111100

0011011110111100101010101000011000101110100111100100101101111111
1010100001100111000111001001010101111010000011000100001111111011
1000011110000000001010101011111101100011000001111101011000101000
1000010111100000010111110111100011101101100000100110101101110010
0101101110101011000110100010100001110110100111011011001111111011
```

9.5.9 KECCAK

These steps are repeated 24 times in the same manner, except for the change in the constants used in the Iota step. The RC values used in each round are represented by the following list.
$RC = \{$
```
0000000000000001, 0000000000008082, 800000000000808A, 8000000080008000,
000000000000808B, 0000000080000001, 8000000080008081, 8000000000008009,
000000000000008A, 0000000000000088, 0000000080008009, 000000008000000A,
000000008000808B, 800000000000008B, 8000000000008089, 8000000000008003,
8000000000008002, 8000000000000080, 000000000000800A, 800000008000000A,
8000000080008081, 8000000000008080, 0000000080000001, 8000000080008008}
```

The complete rounds will not be computed here, as they follow the same shifts as the first round.

After this first block, the second block, complete with padding, is XOR-ed into the current state array, leaving the following state array for round 2.

```
0000011011000010010000010000111011001100101011110100011000000110
0011011111100100001100000010001110101001010111000100100000111111
1010000111001111010100000100111100101000110110011011010010010011
1000000100010001101101100010010001010110001111111101110011010101
1000110010100111100001001111100010011111010100110000011100001110

0100101111101011010111101011001010100001110011101001110001001101
1011010001011001110111000001100100101010000101100100111111111010
0100111010011001001110000000000100001101110001011100011111110011
0010110010000111110101110010111101100111001001010011101110000011
0011100011111010110100010011000100111000011101101111011010000001

0001110100111001101011101011011111100010011101111001110011101101
0001110111000000010000010111010110111111100011010100100010110101
1011100100110001100111010100001001100011101100111001111101000101
0000000010100100010100110011010100000100011110000100110001111110
0101110111011011110010111001001101101101100101101100101010110110

1010100010011101101001101111110010100110010011111111001000000110
1000101111110000110100110100000001110111101010110010010010111010
1010011000100101110111111101001001111000011110010011001001001110
0110101001001001111100110000100100010100101111010001000000001101
0111001111011000111000010000101101101011100000110100000101010001

1100001101010101000011111011011101100101000111101001110001010101
1011100000100111111000100001000100001111111011001111001110010100
0011101011001100101001000010010100000000010010011100100100011000
0101001001000010011111000001100010110011100100101010111101110111
0110001011010010110010001110001011010010100110100110011011101111
```

This is processed through a second round of the complete KECCAK algorithm and, afterwards, without the need for truncation, fed into the algorithm a third and final time. After a last repetition of rounds 1 to 24, the final output is converted into a string, which can be represented by the hex integer `d152d066 81d30d12 fa7019a2 63f5fa16 8a34cfff e6d718a6 6f91a818 7048e482`.

10 Appendix

Step	**Chaining Value for M**	w_j, i	**Shift**	Δm_i	**i-th Step Difference**	**i-th Output for $M^{(0)'}$**
1	a_1	m_0	3			a_1
2	d_1	m_1	7	2^{31}		d_1[7]
3	c_1	m_2	11	$-2^{28} + 2^{31}$	$-2^7 + 2^{10}$	c_1[-8, 11]
4	b_1	m_3	19		2^{25}	b_1[26]
5	a_2	m_4	3			a_2
6	d_2	m_5	7		2^{13}	d_2[14]
7	c_2	m_6	11		$-2^{18} + 2^{21}$	c_2[19, 20, -21, 22]
8	b_2	m_7	19		2^{12}	b_2[-13, -14, 15]
9	a_3	m_8	3		2^{16}	a_3[17]
10	d_3	m_9	7		$2^{19} + 2^{20} - 2^{25}$	d_3[20, -21, -22, 23, -26]
11	c_3	m_{10}	11		-2^{29}	c_3[-30]
12	b_3	m_{11}	19		2^{31}	b_3[32]
13	a_4	m_{12}	3	-2^{16}	$2^{22} + 2^{25}$	a_4[23, 26]
14	d_4	m_{13}	7		$-2^{26} + 2^{28}$	d_4[-27, -29, 30]
15	c_4	m_{14}	11			c_4
16	b_4	m_{15}	19		2^{18}	b_4[19]
17	a_5	m_0	3		$2^{25} - 2^{28} - 2^{31}$	a_5[-26, 27, -29, -32]
18	d_5	m_4	5			d_5
19	c_5	m_8	9			c_5
20	b_5	m_{12}	13	-2^{16}	$-2^{29} + 2^{31}$	b_5[-30, 32]
21	a_6	m_1	3	2^{31}	$2^{28} - 2^{31}$	a_6[-29, 30, -32]
22	d_6	m_5	5			d_6
23	c_6	m_9	9			c_6
24	b_6	m_{13}	13			b_6
25	a_7	m_2	3	$-2^{28} + 2^{31}$		a_7
...	...					...
36	b_9	m_{12}	15	-2^{16}	2^{31}	b_9[-32]
37	a_{10}	m_2	3	$-2^{28} + 2^{31}$	2^{31}	a_{10}[-32]
38	d_{10}	m_{10}	9			d_{10}
39	c_{10}	m_6	11			c_{10}
40	b_{10}	m_{14}	15			b_{10}
41	a_{11}	m_1	3	2^{31}		a_{11}

Table 7: Differential Characteristics in the Collision Differential of MD4 [31]

a_1	$a_{1,7} = b_{0,7}$
d_1	$d_{1,7} = 0, d_{1,8} = a_{1,8}, d_{1,11} = a_{1,11}$
c_1	$c_{1,7} = 1, c_{1,8} = 1, c_{1,11} = 0, c_{1,26} = d_{1,26}$
b_1	$b_{1,11} = 1, b_{1,8} = 0, b_{1,11} = 0, b_{1,26} = 0$
a_2	$a_{2,8} = 1, a_{2,11} = 1, a_{2,26} = 0, a_{2,14} = b_{1,14}$
d_2	$d_{2,14} = 0, d_{2,19} = a_{2,19}, d_{2,20} = a_{2,20}, d_{2,21} = a_{2,21}, d_{2,22} = a_{2,22}, d_{2,26} = 1$
c_2	$c_{2,13} = d_{2,13}, c_{2,14} = 0, c_{2,15} = d_{2,15}, c_{2,19} = 0, c_{2,20} = 0, c_{2,21} = 1, c_{2,22} = 0$
b_2	$b_{2,13} = 1, b_{2,14} = 1, b_{2,15} = 0, b_{2,17} = c_{2,17}, b_{2,19} = 0, b_{2,20} = 0, b_{2,21} = 0, b_{2,22} = 0$
a_3	$a_{3,13} = 1, a_{3,14} = 1, a_{3,15} = 1, a_{3,17} = 0, a_{3,19} = 0, a_{3,20} = 0, a_{3,21} = 0, a_{3,22} = 1, a_{3,23} = b_{2,23}, a_{3,26} = b_{2,26}$
d_3	$d_{3,13} = 1, d_{3,14} = 1, d_{3,15} = 1, d_{3,17} = 0, d_{3,20} = 0, d_{3,21} = 1, d_{3,22} = 1, d_{3,23} = 0, d_{3,26} = 1, d_{3,30} = a_{3,30}$
c_3	$c_{3,17} = 1, c_{3,20} = 0, c_{3,21} = 0, c_{3,22} = 0, c_{3,23} = 0, c_{3,26} = 0, c_{3,30} = 1, c_{3,32} = d_{3,32}$
b_3	$b_{3,20} = 0, b_{3,21} = 1, b_{3,22} = 1, b_{3,23} = c_{3,23}, b_{3,26} = 1, b_{3,30} = 0, b_{3,32} = 0$
a_4	$a_{4,23} = 0, a_{4,26} = 0, a_{4,27} = b_{3,27}, a_{4,29} = b_{4,29}, a_{4,30} = 1, a_{4,32} = 0$
d_4	$d_{4,23} = 0, d_{4,26} = 0, d_{4,27} = 1, d_{4,29} = 1, d_{4,30} = 0, d_{4,32} = 0$
c_4	$c_{4,19} = d_{4,19}, c_{4,23} = 1, c_{4,26} = 1, c_{4,27} = 0, c_{4,29} = 0, c_{4,30} = 0$
b_4	$b_{4,19} = 0, b_{4,26} = c_{4,26} = 1, b_{4,27} = 1, b_{4,29} = 1, b_{4,30} = 0$
a_5	$a_{5,19} = c_{4,19}, a_{5,26} = 1, a_{5,27} = 0, a_{5,29} = 1, a_{5,32} = 1$
d_5	$d_{5,19} = a_{5,19}, d_{5,26} = b_{4,26}, d_{5,27} = b_{4,27}, d_{5,29} = b_{4,29}, d_{5,32} = b_{4,32}$
c_5	$c_{5,26} = d_{5,26}, c_{5,27} = d_{5,27}, c_{5,29} = d_{5,29}, c_{5,30} = d_{5,30}, c_{5,32} = d_{5,32}$
b_5	$b_{5,29} = c_{5,29}, b_{5,30} = 1, b_{5,32} = 0$
a_6	$a_{6,21} = 1, a_{6,32} = 1$
d_6	$d_{6,29} = b_{5,29}$
c_6	$c_{6,29} = d_{6,29}, c_{6,30} = d_{6,30} + 1, c_{6,32} = d_{6,32} + 1$
b_9	$b_{9,32} = 1$
a_{10}	$a_{10,32} = 1$

Table 8: Set of sufficient conditions for collisions MD4 [31]

Step	**Chaining Value for M**	w_i	**Shift**	Δm_i	**i-th Step Difference**	**i-th Output for $M^{(0)'}$**
4	b_1	m_3	22			
5	a_2	m_4	7	2^{31}	-2^6	a_2[7, ..., 22, -23]
6	d_2	m_5	12		$-2^6 + 2^{23} + 2^{31}$	d_2[-7, 24, 32]
7	c_2	m_6	17		$-1 - 2^6 + 2^{23} - 2^{27}$	c_2[7, 8, 9, 10, 11, -12, -24, -25, -26, 27, 28, 29, 30, 31, 32, 1, 2, 3, 4, 5, -6]
8	b_2	m_7	22		$1 - 2^{15} - 2^{17} - 2^{23}$	b_2[1, 16, -17, 18, 19, 20, -21, -24]
9	a_3	m_8	7		$1 - 2^6 + 2^{31}$	a_3[-1, 2, 7, 8, -9, -32]
10	d_3	m_9	12		$2^{12} + 2^{31}$	d_3[-13, 14, 32]
11	c_3	m_{10}	17		$2^{30} + 2^{31}$	c_3[31, 32]
12	b_3	m_{11}	22	2^{15}	$-2^7 - 2^{13} + 2^{31}$	b_3[8, -9, 14, ..., 19, -20, 32]
13	a_4	m_{12}	7		$2^{24} + 2^{31}$	a_4[-25, 26, 32]
14	d_4	m_{13}	12		2^{31}	d_4[32]
15	c_4	m_{14}	17	2^{15}	$2^3 - 2^{15} + 2^{31}$	c_4[4, -16, 32]
16	b_4	m_{15}	22		$-2^{29} + 2^{31}$	b_4[-30, 32]
17	a_5	m_1	5		2^{31}	a_5[32]
18	d_5	m_6	9		2^{31}	d_5[32]
19	c_5	m_{11}	14	2^{15}	$2^{17} + 2^{31}$	c_5[18, 32]
20	b_5	m_0	20		2^{31}	b_5[32]
21	a_6	m_5	5		2^{31}	a_6[32]
22	d_6	m_{10}	9		2^{31}	d_6[32]
23	c_6	m_{15}	14			c_6
24	b_6	m_4	20	2^{31}		b_6
25	a_7	m_9	5			a_7
26	d_7	m_{14}	9	2^{31}		d_7
27	c_7	m_3	14			c_7
...	...	...	...	...	...	...
34	d_9	m_8	11			d_9
35	c_9	m_{11}	16	2^{15}	2^{31}	c_9[*32]
36	b_9	m_{14}	23	2^{31}	2^{31}	b_9[*32]
37	a_{10}	m_1	4		2^{31}	a_{10}[*32]
38	d_{10}	m_4	11	2^{31}	2^{31}	d_{10}[*32]
39	c_{10}	m_7	16		2^{31}	c_{10}[*32]
...	...	...	...	...	...	...
45	a_{12}	m_9	4		2^{31}	a_{12}[*32]
46	d_{12}	m_{12}	11		2^{31}	d_{12}[32]
47	c_{12}	m_{15}	16		2^{31}	c_{12}[32]
48	b_{12}	m_2	23		2^{31}	b_{12}[32]
49	a_{13}	m_0	6		2^{31}	a_{13}[32]
50	d_{13}	m_7	10		2^{31}	d_{13}[-32]
51	c_{13}	m_{14}	15	2^{31}	2^{31}	c_{13}[32]
52	b_{13}	m_5	21		2^{31}	b_{13}[-32]
...	...	...	...	...	...	...

Step	**Chaining Value for M**	w_i	**Shift**	Δm_i	**i-th Step Difference**	**i-th Output for $M^{(0)'}$**
58	d_{15}	m_{15}	10		2^{31}	d_{15}[-32]
59	c_{15}	m_6	15		2^{31}	c_{15}[32]
60	b_{15}	m_{13}	21		2^{31}	b_{15}[32]
61	$aa_0 = a_{16} + a_0$	m_4	6	2^{31}	2^{31}	$aa_0^{'} = aa_0$[32]
62	$dd_0 = d_{16} + d_0$	m_{11}	10	2^{15}	2^{31}	$dd_0^{'} = dd_0$[26, 32]
63	$cc_0 = c_{16} + c_0$	m_2	15		2^{31}	$cc_0^{'} = cc_0$[-26, 27, 32]
64	$bb_0 = b_{16} + b_0$	m_9	21		2^{31}	$bb_0^{'} = bb_0$[26, -32]

Table 9: Differential Characteristics in the First Iteration Differential of MD5 [32]

c_1	$c_{1,7}=0, c_{1,12}=0, c_{1,20}=0$
b_1	$b_{1,7}=0, b_{1,8}=c_{1,8}, b_{1,9}=c_{1,9}, b_{1,10}=c_{1,10}, b_{1,11}=c_{1,11}, b_{1,12}=1, b_{1,13}=c_{1,13}, b_{1,14}=c_{1,14}, b_{1,15}=c_{1,15}, b_{1,16}=c_{1,16}, b_{1,17}=c_{1,17}, b_{1,18}=c_{1,18}, b_{1,19}=c_{1,19}, b_{1,20}=1, b_{1,21}=c_{1,21}, b_{1,22}=c_{1,22}, b_{1,23}=c_{1,23}, b_{1,24}=0, b_{1,32}=1$
a_2	$a_{2,1}=1, a_{2,3}=1, a_{2,6}=1, a_{2,7}=0, a_{2,8}=0, a_{2,9}=0, a_{2,10}=0, a_{2,11}=0, a_{2,12}=0, a_{2,13}=0, a_{2,14}=0, a_{2,15}=0, a_{2,16}=0, a_{2,17}=0, a_{2,18}=0, a_{2,19}=0, a_{2,20}=0, a_{2,21}=0, a_{2,22}=0, a_{2,23}=1, a_{2,24}=0, a_{2,26}=0, a_{2,28}=1, a_{2,32}=1$
d_2	$d_{2,1}=1, d_{2,2}=a_{2,2}, d_{2,3}=0, d_{2,4}=a_{2,4}, d_{2,5}=a_{2,5}, d_{2,6}=0, d_{2,7}=1, d_{2,8}=0, d_{2,9}=0, d_{2,10}=0, d_{2,11}=1, d_{2,12}=1, d_{2,13}=1, d_{2,14}=1, d_{2,15}=0, d_{2,16}=1, d_{2,17}=1, d_{2,18}=1, d_{2,19}=1, d_{2,20}=1, d_{2,21}=1, d_{2,22}=1, d_{2,23}=1, d_{2,24}=0, d_{2,25}=a_{2,25}, d_{2,26}=1, d_{2,27}=a_{2,27}, d_{2,28}=0, d_{2,29}=a_{2,29}, d_{2,30}=a_{2,30}, d_{2,31}=a_{2,31}, d_{2,32}=0$
c_2	$c_{2,1}=0, c_{2,2}=0, c_{2,3}=0, c_{2,4}=0, c_{2,5}=0, c_{2,6}=1, c_{2,7}=0, c_{2,8}=0, c_{2,9}=0, c_{2,10}=0, c_{2,11}=0, c_{2,12}=1, c_{2,13}=1, c_{2,14}=1, c_{2,15}=1, c_{2,16}=1, c_{2,17}=0, c_{2,18}=1, c_{2,19}=1, c_{2,20}=1, c_{2,21}=1, c_{2,22}=1, c_{2,23}=1, c_{2,24}=1, c_{2,25}=1, c_{2,26}=1, c_{2,27}=0, c_{2,28}=0, c_{2,29}=0, c_{2,30}=0, c_{2,31}=0, c_{2,32}=0$
b_2	$b_{2,1}=0, b_{2,2}=0, b_{2,3}=0, b_{2,4}=0, b_{2,5}=0, b_{2,6}=0, b_{2,7}=1, b_{2,8}=0, b_{2,9}=1, b_{2,10}=0, b_{2,11}=0, b_{2,12}=0, b_{2,14}=0, b_{2,16}=0, b_{2,17}=1, b_{2,18}=0, b_{2,19}=0, b_{2,20}=0, b_{2,21}=1, b_{2,24}=1, b_{2,25}=1, b_{2,26}=0, b_{2,27}=0, b_{2,28}=0, b_{2,29}=0, b_{2,30}=0, b_{2,31}=0, b_{2,32}=0$
a_3	$a_{3,1}=1, a_{3,2}=0, a_{3,3}=1, a_{3,4}=1, a_{3,5}=1, a_{3,6}=1, a_{3,7}=0, a_{3,8}=0, a_{3,9}=1, a_{3,10}=1, a_{3,11}=1, a_{3,12}=1, a_{3,13}=b_{2,13}, a_{3,14}=1, a_{3,16}=0, a_{3,17}=0, a_{3,18}=0, a_{3,19}=0, a_{3,20}=0, a_{3,21}=1, a_{3,25}=1, a_{3,26}=1, a_{3,27}=0, a_{3,26}=1, a_{3,29}=1, a_{3,30}=1, a_{3,31}=1, a_{3,32}=1$
d_3	$d_{3,1}=0, d_{3,2}=0, d_{3,7}=1, d_{3,8}=0, d_{3,9}=0, d_{3,13}=1, d_{3,14}=0, d_{3,16}=1, d_{3,17}=1, d_{3,18}=1, d_{3,19}=1, d_{3,20}=1, d_{3,21}=1, d_{3,24}=0, d_{3,31}=1, d_{3,32}=0$
c_3	$c_{3,1}=0, c_{3,2}=1, c_{3,7}=1, c_{3,8}=0, c_{3,9}=0, c_{3,13}=0, c_{3,14}=0, c_{3,15}=d_{3,15}, c_{3,16}=1, c_{3,17}=1, c_{3,18}=0, c_{3,19}=0, c_{3,20}=0, c_{3,31}=0, c_{3,32}=0$
b_3	$b_{3,8}=0, b_{3,9}=1, b_{3,13}=1, b_{3,14}=0, b_{3,15}=0, b_{3,16}=0, b_{3,17}=0, b_{3,18}=0, b_{3,19}=0, b_{3,20}=1, b_{3,25}=c_{3,25}, b_{3,26}=c_{3,26}, b_{3,31}=0, b_{3,32}=0$
a_4	$a_{4,4}=1, a_{4,8}=0, a_{4,9}=0, a_{4,14}=1, a_{4,15}=1, a_{4,16}=1, a_{4,17}=1, a_{4,18}=1, a_{4,19}=1, a_{4,20}=1, a_{4,25}=1, a_{4,26}=0, a_{4,31}=1, a_{4,32}=0$
d_4	$d_{4,4}=1, d_{4,8}=1, d_{4,9}=1, d_{4,14}=1, d_{4,15}=1, d_{4,16}=1, d_{4,17}=1, d_{4,18}=1, d_{4,19}=0, d_{4,20}=1, d_{4,25}=0, d_{4,26}=0, d_{4,30}=0, d_{4,32}=0$
c_4	$c_{4,4}=0, c_{4,16}=1, c_{4,25}=1, c_{4,26}=0, c_{4,30}=1, c_{4,32}=0$
b_4	$b_{4,30}=1, b_{4,32}=0$
a_5	$a_{5,4}=b_{4,4}, a_{5,16}=b_{4,16}, a_{5,18}=0, a_{5,32}=0$
d_5	$d_{5,18}=1, d_{5,30}=a_{5,30}, d_{5,32}=0$

c_5	$c_{5,18} = 0, c_{5,32} = 0$
b_5	$b_{5,32} = 0$
$a_6 - b_6$	$a_{6,18} = b_{5,18}, a_{6,32} = 0, d_{6,32} = 0, c_{6,32} = 0, b_{6,32} = c_{6,32} + 1$
c_9, c_{12}	$\Phi_{34,32} = 0, b_{12,32} = d_{12,32}$
$a_{13} - b_{13}$	$a_{13,32} = c_{12,32}, d_{13,32} = b_{12,32} + 1, c_{13,32} = a_{13,32}, b_{13,32} = d_{13,32}$
$a_{14} - b_{14}$	$a_{14,32} = c_{13,32}, d_{14,32} = b_{13,32}, c_{14,32} = a_{14,32}, b_{14,32} = d_{14,32}$
a_{15}	$a_{15,32} = c_{14,32}$
d_{15}	$d_{15,32} = b_{14,32}$
c_{15}	$c_{15,32} = a_{15,32}$
b_{15}	$b_{15,26} = 0, b_{15,32} = d_{15,32} + 1$
$aa_0 = a_{16} + a_0$	$a_{16,26} = 1, a_{16,27} = 0, a_{16,32} = c_{15,32}$
$dd_0 = d_{16} + d_0$	$dd_{0,26} = 0, d_{16,32} = b_{15,32}$
$cc_0 = c_{16} + c_0$	$cc_{0,26} = 1, cc_{0,27} = 0, cc_{0,32} = dd_{0,32}, c_{16,32} = d_{16,32}$
$bb_{0,6} = 0, bb_0 = b_{16} + b_0$	$bb_{0,26} = 0, bb_{0,27} = 0, bb_{0,32} = cc_{0,32}$

Table 10: Set of Sufficient Conditions for the First Iteration Differential of MD5 [32]

Step	**Output of i-th step for M_1**	w_i	s_i	Δw_i	**Output Difference of i-th step**	**Output of i-th step for M_1'**
IV	aa_0, dd_0, cc_0, bb_0					aa_0[32], dd_0[26, 32], cc_0[-26, 27, 32], bb_0[26, -32]
1	a_1	m_0	7		$2^{25} + 2^{31}$	a_1[26, -32]
2	d_1	m_1	12		$2^5 + 2^{25} + 2^{31}$	d_1[6, 26, -32]
3	c_1	m_2	17		$2^5 + 2^{11} + 2^{16} + 2^{25} + 2^{31}$	c_1[-6, -7, 8, -12, 13, -17, ..., -21, 22, -26, ..., -30, 31, -32]
4	b_1	m_3	22		$-2 + 2^5 + 2^{25} + 2^{31}$	b_1[2, 3, 4, -5, 6, -26, 27, -32]
5	a_2	m_4	7	2^{31}	$1 + 2^6 + 2^8 + 2^9 + 2^{31}$	a_2[1, -7, 8, 9, -10, -11, -12, 13, 32]
6	d_2	m_5	12		$-2^{16} - 2^{20} + 2^{31}$	d_2[17, -18, 21, -22, 32]
7	c_2	m_6	17		$-2^6 - 2^{27} + 2^{31}$	c_2[7, 8, 9, -10, 28, -29, -32]
8	b_2	m_7	22		$2^{15} - 2^{17} - 2^{23} + 2^{31}$	b_2[-16, 17, -18, 24, 25, 26, -27, -32]
9	a_3	m_8	7		$1 + 2^6 + 2^{31}$	a_3[-1, 2, -7, -8, -9, 10, -32]
10	d_3	m_9	12		$2^{12} + 2^{31}$	d_3[13, -32]
11	c_3	m_{10}	17		2^{31}	c_3[-32]
12	b_3	m_{11}	22	-2^{15}	$-2^7 - 2^{13} + 2^{31}$	b_3[-8, 14, 15, 16, 17, 18, 19, -20, -32]
13	a_4	m_{12}	7		$2^{24} + 2^{31}$	a_4[-25, ..., -30, 31, 32]
14	d_4	m_{13}	12		2^{31}	d_4[32]
15	c_4	m_{14}	17	2^{31}	$2^3 + 2^{15} + 2^{31}$	c_4[4, 16, 32]
16	b_4	m_{15}	22		$-2^{29} + 2^{31}$	b_4[-30, 32]
17	a_5	m_1	5		2^{31}	a_5[32]
18	d_5	m_6	9		2^{31}	d_5[32]
19	c_5	m_{11}	14	-2^{15}	$2^{17} + 2^{31}$	c_5[18, 32]
20	b_5	m_0	20		2^{31}	b_5[32]
21	a_6	m_5	5		2^{31}	a_6[32]
22	d_6	m_{10}	9		2^{31}	d_6[32]
23	c_6	m_{15}	14			c_6[32]
24	b_6	m_4	20	2^{31}		b_6[32]
25	a_7	m_9	5			a_7
26	d_7	m_{14}	9	2^{31}		d_7
27	c_7	m_3	14			c_7
...	...					
34	d_9	m_8	11			d_9
35	c_9	m_{11}	16	-2^{15}	2^{31}	c_9[*32]
36	b_9	m_{14}	23	2^{31}	2^{31}	b_9[*32]
37	a_{10}	m_1	4		2^{31}	a_{10}[*32]
38	d_{10}	m_4	11	2^{31}	2^{31}	d_{10}[*32]
39	c_{10}	m_7	16		2^{31}	c_{10}[*32]
...	...					

Step	**Output of i-th step for M_1**	w_i	s_i	Δw_i	**Output Difference of i-th step**	**Output of i-th step for M_1'**
49	a_{13}	m_0	6		2^{31}	a_{13}[32]
50	d_{13}	m_7	10		2^{31}	d_{13}[-32]
51	c_{13}	m_{14}	15	2^{31}	2^{31}	c_{13}[32]
52	b_{13}	m_5	21		2^{31}	b_{13}[-32]
...						
59	c_{15}	m_6	15		2^{31}	c_{15}[32]
60	b_{15}	m_{13}	21		2^{31}	b_{15}[32]
61	$a_{16}+aa_0$	m_4	6	2^{31}		$a_{16} + aa_0 = a'_{16} + aa'_0$
62	$d_{16}+dd_0$	m_{11}	10	-2^{15}		$d_{16} + dd_0 = d'_{16} + dd'_0$
63	$c_{16} + cc_0$	m_2	15			$c_{16} + cc_0 = c'_{16} + cc'_0$
64	$b_{16} + bb_0$	m_9	21			$b_{16} + bb_0 = b'_{16} + bb'_0$

Table 11: Differential Characteristics in the Second Iteration Differential [32]

a_1	$a_{1,6}=0, a_{1,12}=0, a_{1,22}=1, a_{1,27}=0, a_{1,28}=0, a_{1,32}=1$
d_1	$d_{1,2}=0, d_{1,3}=0, d_{1,6}=0, d_{1,7}=a_{1,7}, d_{1,8}=a_{1,8}, d_{1,12}=1, d_{1,13}=a_{1,13}, d_{1,16}=0, d_{1,17}=a_{1,17}, d_{1,18}=a_{1,18}, d_{1,19}=a_{1,19}, d_{1,20}=a_{1,20}, d_{1,21}=a_{1,21}, d_{1,22}=0, d_{1,2}=0, d_{1,27}=1, d_{1,28}=1, d_{1,29}=a_{1,29}, d_{1,30}=a_{1,30}, d_{1,31}=a_{1,31}, d_{1,32}=1$
c_1	$c_{1,2}=1, c_{1,3}=1, c_{1,4}=d_{1,4}, c_{1,5}=d_{1,5}, c_{1,6}=0, c_{1,7}=1, c_{1,8}=0, c_{1,9}=1, c_{1,12}=1, c_{1,13}=0, c_{1,17}=1, c_{1,18}=1, c_{1,19}=1, c_{1,20}=1, c_{1,21}=1, c_{1,22}=0, c_{1,26}=1, c_{1,27}=1, c_{1,28}=1, c_{1,29}=1, c_{1,30}=1, c_{1,31}=0, c_{1,32}=1$
b_1	$b_{1,1}=c_{1,1}, b_{1,2}=0, b_{1,3}=0, b_{1,4}=0, b_{1,5}=1, b_{1,6}=0, b_{1,7}=0, b_{1,8}=0, b_{1,9}=0, b_{1,10}=c_{1,10}, b_{1,11}=c_{1,11}, b_{1,12}=0, b_{1,13}=0, b_{1,17}=0, b_{1,18}=0, b_{1,19}=1, b_{1,20}=0, b_{1,21}=0, b_{1,22}=0, b_{1,26}=1, b_{1,27}=0, b_{1,28}=1, b_{1,29}=1, b_{1,30}=1, b_{1,31}=0, b_{1,32}=1$
a_2	$a_{2,1}=0, a_{2,2}=0, a_{2,3}=0, a_{2,4}=0, a_{2,5}=1, a_{2,6}=0, a_{2,7}=1, a_{2,8}=0, a_{2,9}=0, a_{2,10}=1, a_{2,11}=1, a_{2,12}=1, a_{2,13}=0, a_{2,17}=0, a_{2,18}=1, a_{2,19}=1, a_{2,20}=1, a_{2,21}=0, a_{2,22}=1, a_{2,27}=0, a_{2,28}=1, a_{2,29}=0, a_{2,30}=0, a_{2,31}=1, a_{2,32}=0$
d_2	$d_{2,1}=0, d_{2,2}=1, d_{2,3}=1, d_{2,4}=0, d_{2,5}=1, d_{2,6}=0, d_{2,7}=1, d_{2,8}=0, d_{2,9}=0, d_{2,10}=0, d_{2,11}=1, d_{2,12}=1, d_{2,13}=0, d_{2,17}=0, d_{2,18}=1, d_{2,21}=0, d_{2,22}=1, d_{2,26}=0, d_{2,27}=1, d_{2,28}=0, d_{2,29}=0, d_{2,32}=0$
c_2	$c_{2,1}=1, c_{2,7}=0, c_{2,8}=0, c_{2,9}=0, c_{2,10}=1, c_{2,11}=1, c_{2,12}=1, c_{2,13}=1, c_{2,16}=d_{2,16}, c_{2,17}=1, c_{2,18}=0, c_{2,21}=0, c_{2,22}=0, c_{2,24}=d_{2,24}, c_{2,25}=d_{2,25}, c_{2,26}=1, c_{2,27}=1, c_{2,28}=0, c_{2,29}=1, c_{2,32}=1,$
b_2	$b_{2,1}=0, b_{2,2}=c_{2,2}, b_{2,7}=1, b_{2,8}=1, b_{2,9}=1, b_{2,10}=1, b_{2,16}=1, b_{2,17}=0, b_{2,18}=1, b_{2,21}=1, b_{2,22}=1, b_{2,24}=0, b_{2,25}=0, b_{2,26}=0, b_{2,27}=1, b_{2,28}=0, b_{2,29}=0, b_{2,32}=1$
a_3	$a_{3,1}=1, a_{3,2}=0, a_{3,7}=1, a_{3,8}=1, a_{3,9}=1, a_{3,10}=0, a_{3,13}=b_{2,13}, a_{3,16}=0, a_{3,17}=1, a_{3,18}=0, a_{3,24}=0, a_{3,25}=0, a_{3,26}=0, a_{3,27}=1, a_{3,28}=1, a_{3,29}=1, a_{3,32}=1$
d_3	$d_{3,1}=0, d_{3,2}=0, d_{3,7}=1, d_{3,8}=1, d_{3,9}=1, d_{3,10}=1, d_{3,13}=0, d_{3,16}=1, d_{3,17}=1, d_{3,18}=1, d_{3,19}=0, d_{3,24}=1, d_{3,25}=1, d_{3,26}=1, d_{3,27}=1, d_{3,32}=1$
c_3	$c_{3,1}=1, c_{3,2}=1, c_{3,7}=1, c_{3,8}=1, c_{3,9}=1, c_{3,10}=1, c_{3,13}=0, c_{3,14}=d_{3,14}, c_{3,15}=d_{3,15}, c_{3,16}=1, c_{3,17}=1, c_{3,18}=0, c_{3,19}=1, c_{3,20}=d_{3,20}, c_{3,32}=1,$
b_3	$b_{3,8}=1, b_{3,13}=1, b_{3,14}=0, b_{3,15}=0, b_{3,16}=0, b_{3,17}=0, b_{3,18}=0, b_{3,19}=0, b_{3,20}=1, b_{3,25}=c_{3,25}, b_{3,26}=c_{3,26}, b_{3,27}=c_{3,27}, b_{3,28}=c_{3,28}, b_{3,29}=c_{3,29}, b_{3,30}=c_{3,30}, b_{3,32}=1$
a_4	$a_{4,4}=1, a_{4,8}=0, a_{4,14}=1, a_{4,15}=1, a_{4,16}=1, a_{4,17}=1, a_{4,18}=1, a_{4,19}=0, a_{4,20}=1, a_{4,25}=1, a_{4,26}=1, a_{4,27}=1, a_{4,28}=1, a_{4,29}=1, a_{4,30}=1, a_{4,31}=0, a_{4,32}=0$
d_4	$d_{4,4}=1, d_{4,8}=1, d_{4,14}=1, d_{4,15}=1, d_{4,16}=1, d_{4,17}=1, d_{4,18}=1, d_{4,19}=0, d_{4,20}=1, d_{4,25}=0, d_{4,26}=0, d_{4,27}=0, d_{4,28}=0, d_{4,29}=0, d_{4,30}=0, d_{4,31}=1, d_{4,32}=0$
c_4	$c_{4,4}=0, c_{4,16}=0, c_{4,25}=1, c_{4,26}=0, c_{4,27}=1, c_{4,28}=1, c_{4,29}=1, c_{4,30}=1, c_{4,31}=1, c_{4,32}=0$
b_4	$b_{4,30}=1, b_{4,32}=0$
a_5	$a_{5,4}=b_{4,4}, a_{5,16}=b_{4,16}, a_{5,18}=0, a_{5,32}=0$
d_5	$d_{5,18}=1, d_{5,30}=a_{5,30}, d_{5,32}=0$
c_5	$c_{5,18}=0, c_{5,32}=0$
b_5	$b_{5,32}=0$
a_6-b_6	$a_{6,18}=b_{5,18}, d_{6,32}=0, c_{6,32}=0, b_{6,32}=c_{6,32}+1$
c_9, b_{12}	$\phi_{34,32}=1, b_{12,32}=d_{12,32}$
$a_{13}-b_{13}$	$a_{13,32}=c_{12,32}, d_{13,32}=b_{12,32}+1, c_{13,32}=a_{13,32}, b_{13,32}=d_{13,32}$

$a_{14} - b_{14}$	$a_{14,32} = c_{13,32}, d_{14,32} = b_{13,32} + 1, c_{14,32} = a_{14,32}, b_{14,32} = d_{14,32}$
$a_{15} - b_{15}$	$a_{15,32} = c_{14,32}, d_{15,32} = b_{14,32} + 1, c_{15,32} = a_{14,32}, b_{15,32} = d_{15,32} + 1$
a_{16}	$a_{16,26} = 1, a_{16,32} = c_{15,32}$
d_{16}	$d_{16,26} = 1, d_{16,32} = b_{15,32}$
c_{16}	$c_{16,26} = 1, c_{16,32} = d_{16,32}$
b_{16}	$b_{16,26} = 1$

Table 12: Set of Sufficient Conditions for the Second Iteration Differential [32]

i	ΔA_i	ΔW_i	F_W	$P_u(i)$	$P_c(i)$	$N_s(i)$
-4	0000111101001011 1000011111000011					
-3	0100000011001001 0101000111011000					
-2	0110001011101011 0111001111111010					
-1	1110111111001101 1010101110001001					
0	0110011101000101 0010001100000001	0110001111011010 11101111110111nu	0	0.00	0.00	1.07
1	0000001110001111 100010001001000n	0n110000010100000 11010-010u1n01u1	1	0.00	0.00	1.07
2	0n00100101000010 10110-00011u0un0	0u01001011101101 --11011n100100	4	-3.00	0.00	2.07
3	1u10100001110010 100-1un110nuu110	unn10000000000-1 001--10u0u11u1	5	-4.00	0.00	3.07
4	1un0010110011110 un1100-0n1n11nu1	n0n0110110110100 1-01111-10110101	2	-2.00	0.00	4.07
5	n1u10110101un000 10nu10u111000010	u1100101101000011 1111--1n101011	4	-4.00	0.00	4.07
6	100u100u01111nu0 0u1110nu111u1un1	10u0111001100110 1-1---101011n	7	-5.00	0.00	4.07
7	nn1100101n110101 1-1111-11u1001u0	00n100101010-101 ---100nu11111	7	-5.00	0.00	6.07
8	011101110011100u0 0010-0n11110u11	u1010000001100- -00--11-000010u	7	-6.00	0.00	8.07
9	1n1u000101uuuu0u u1110-1010n110n0	1n00010100000101 -100-10-u1111n0	4	-3.00	0.00	9.07
10	1011000101n11111 n111u-01n00un100	nu1101010110001- -011--1u0110un	6	-5.00	0.00	10.07
11	nnnnnnnnnnnnnnnn nnnnnnn-nnnnn0n1	1u01111111111111 -----0u110n1	9	-9.00	0.00	11.07
12	0011010000001111 0110000110011000	0101100101101101 01101--1-0101nu	4	-3.00	0.00	11.07
13	0100000000001000 000111100-011000	0n001000010101- -----n1010n1	11	-4.00	0.00	12.07
14	1001100010001100 0-0---0110101	nu00001010011-- ----1n1100uu	11	-2.00	0.00	19.07

i	ΔA_i	ΔW_i	F_W	$P_u(i)$	$P_c(i)$	$N_s(i)$
15	1101101011111-1 -----00010n	uu101101010-1-1- ----1-1n011n1	11	−0.07	0.00	28.07
16	11111100---- -----0-0111	1101001010100-- ----1010101u	0	−1.00	−1.00	39.00
17	0000------ -----1-1111	1u0011100111-- ----111011u0	0	−1.00	−0.99	38.00
18	--0------ ------01u-	un00111011-0-0- ----0n0011nu	0	0.00	0.00	37.00
19	-------- --------n	1u1100011111-- ----1un011n0	0	0.00	0.00	37.00
20	-------- --------	n1101001100--- -----011000n	0	−1.00	−1.00	37.00
21	-------- -------n-	1u1000110-1-0-- ----0u1000n0	0	−2.00	−2.00	36.00
22	-------- -------n-	1n011010011--- ----0u0110n1	0	−2.00	−2.00	34.00
23	-------- --------	0n10011011--- ----011111n0	0	−1.00	−1.00	32.00
24	-------- --------	00101001-0-0-- ----001010n1	0	−1.00	−1.00	31.00
25	-------- -------n-	0001110111--- ----1u100100	0	0.00	0.00	30.00
26	-------- --------	n00010000---- ----0-11111n1	0	−1.00	−1.00	30.00
27	-------- --------	n001111-1-1--- ----11101010	0	0.00	0.00	29.00
28	-------- --------	u10111110---- -----11001n0	0	−1.00	−1.00	29.00
29	-------- -------n-	n0011100---- ----1u110010	0	0.00	0.00	28.00
30	-------- --------	001010-1-1--- ----101010110	0	−2.00	−2.00	28.00
31	-------- -------n-	u0110101---- ----0u110111	0	0.00	0.00	26.00
32	-------- --------	u101001----- ----011111010	0	−2.00	−2.00	26.00
33	-------- -------u-	00010-1-0---- ---110n100000	0	0.00	0.00	24.00
34	-------- --------	u011010----- ----001101110	0	−2.00	−2.00	24.00
35	-------- -------n-	101111----- ---010u111001	0	0.00	0.00	22.00
36	-------- --------	n111-1-1---- ----1010110u0	0	−1.00	−1.00	22.00
37	-------- --------	110110----- ----100000000	0	0.00	0.00	21.00
38	-------- --------	n1001------ ----010111110	0	0.00	0.00	21.00

i	ΔA_i	ΔW_i	F_W	$P_u(i)$	$P_c(i)$	$N_s(i)$
39	-------- --------	u11-0-1----- ----101101011	0	0.00	0.00	21.00
40	-------- --------	01010------ ----01011100	0	0.00	0.00	21.00
41	-------- --------	1011------ ----100100000	0	0.00	0.00	21.00
42	-------- --------	00-0-0----- ----100111001	0	0.00	0.00	21.00
43	-------- --------	1101------ ----001111011	0	0.00	0.00	21.00
44	-------- --------	011------- ----10010000	0	0.00	0.00	21.00
45	-------- --------	1-1-0------ ----101111000	0	0.00	0.00	21.00
46	-------- --------	110------- ---1011010010	0	0.00	0.00	21.00
47	-------- --------	01------- ----101011000	0	0.00	0.00	21.00
48	-------- --------	-0-0------ ----101100001	0	0.00	0.00	21.00
49	-------- --------	10------- ---1101010111	0	0.00	0.00	21.00
50	-------- --------	0-------- ---1010101n11	0	−1.00	−1.00	21.00
51	-------- -------n-	0-1------- ---10u100011-	0	0.00	0.00	20.00
52	-------- --------	1-------- ----001000u11	0	−1.00	−1.00	20.00
53	-------- --------	-------- ---110111n00u	0	−2.00	−2.00	19.00
54	-------- ------n--	-1------- ----u111011-u	0	−1.00	−1.00	17.00
55	-------- --------	-------- ---101010u00u	0	−1.00	−1.00	16.00
56	-------- --------	-------- ----0111n10u-	0	−2.00	−1.91	15.00
57	-------- ------n--	0-------- ---u111000-u-	0	−1.00	−1.00	13.00
58	-------- --------	-------- ---0-1010un1u-	0	−2.00	−1.83	12.00
59	-------- ------u--	-------- ---1n01n11u-	0	−2.00	−1.87	10.00
60	-------- -----n---	-------- ---u-11000xu-0	0	−2.00	−1.00	8.00
61	-------- --------	-------- ---0000n01ux-	0	−2.00	−1.00	6.00
62	-------- --------	-------- ---1000n00n-x-	0	−3.00	−1.89	4.00
63	-------- -----n---	-------- --u-10010-n-n-	0	−1.00	−1.00	1.00

i	ΔA_i	ΔW_i	F_W	$P_u(i)$	$P_c(i)$	$N_s(i)$
64	-------- --------					

Table 13: Characteristic used for the first block of the 64-step collision [4]

i	ΔA_i	ΔW_i	F_W	$P_u(i)$	$P_c(i)$	$N_s(i)$
-4	1111001111110001 0000010000n10011					
-3	0110111011100000 1010001110011101					
-2	1100101110110010 0011110111000100					
-1	1001011011110100 100111001n110101					
0	1010000000011110 1110010101101000	0011101100101010 101101-0111000nu	1	0.00	0.00	1.24
1	1111001001110010 110010-10000n1nu	1n1010101101000- -0100101u1n11u1	3	-3.00	0.00	2.24
2	uu10001001100001 000001nu01un01u0	0u10011010011100 10011-11n101110	2	-2.00	0.00	2.24
3	0u10010110111100 000nnnn01011u1nn	nun1110111101010 010011010n0u01n0	0	0.00	0.00	2.24
4	0nu1110110110n00 10uuuu0uunuu1u10	n0n1110110111110 0010001000001110	0	0.00	0.00	2.24
5	1000111nu111u000 1n11111100100001	u010101101000110 10-00101-u100000	2	-1.00	0.00	2.24
6	u11110un101n0u01 11-1011n010u1010	10n1011011100-10 110--10011011u	5	-2.00	0.00	3.24
7	u001u11nn0101011 100n--0u011n111	11u10011111001- -00-0-10uu00011	6	-4.00	0.00	6.24
8	1n010101001u01n1 0000-0-11000u011	u0111110010110- -0--1-001110n	9	-7.00	0.00	8.24
9	01001u1n10100110 100101-1-uu10100	1n11110101100-- -----u0001n0	12	-10.00	0.00	10.24
10	uuuuuuuuuuuuuuu uuuuuuu-1100u011	nu0100001100010 0-----n1001nu	9	-8.00	0.00	12.24
11	010011101111110 0011111un-0111100	1n0010110111100 1001---1n001u0	6	-6.00	0.00	13.24
12	110000001011111 1111111111111u110	101010011011001 001000--001010nn	3	-2.00	-1.00	13.24
13	011000010111111 1111111-0110110n	1n0000010110111 11----n1110u0	8	-2.24	0.00	14.24
14	010111111001101 0110----010u0	uu01000110110- -----1u0-11nn	12	-4.00	0.00	20.00
15	010100100100000 10------00nu	un110110000-0-0 ----1-0n000n1	11	-1.00	0.00	28.00
16	001001001011-- ------10010	1100010100000- -----1101001n	0	0.00	0.00	38.00

17	100000----- -------1000	0n1101111101-- -----11-001u1	0	−1.00	−0.99	38.00
18	--0----- -------0u1	nn11101111-0-1- -----0n0010nu	0	0.00	0.00	37.00
19	-------- --------n	0u1100011010-- ----1un000n1	0	−1.00	−1.00	37.00
20	-0------- --------	n0101010011-- ----11-10110n	0	−1.00	−1.00	36.00
21	-------- -------n-	1u0001000-0-0-- ----0u1000n1	0	−1.00	−1.00	35.00
22	-------- -------n-	0n010001010--- ----0u-011n1	0	−2.00	−2.00	34.00
23	-------- --------	1n10010111--- -----00101n1	0	−1.00	−1.00	32.00
24	-------- --------	11011111-0-1-- ----000101n1	0	−1.00	−1.00	31.00
25	-------- -------n-	0010000100--- ----0u010000	0	0.00	0.00	30.00
26	-------- --------	u10011101---- ----001000u0	0	−1.00	−1.00	30.00
27	-------- --------	n100100-0-0--- ----01010001	0	0.00	0.00	29.00
28	-------- --------	u11001101---- ----0-0-100n0	0	−1.00	−1.00	29.00
29	-------- -------n-	n1111011---- ----1u110000	0	0.00	0.00	28.00
30	-------- --------	100110-1-1--- ----00-00100	0	−2.00	−2.00	28.00
31	-------- -------u-	u0000101---- ----1n000111	0	0.00	0.00	26.00
32	-------- --------	u011010----- ---0001111100	0	−2.00	−2.00	26.00
33	-------- -------n-	11111-0-0---- ----0-u100101	0	0.00	0.00	24.00
34	-------- --------	u011010----- ----0-0000000	0	−2.00	−2.00	24.00
35	-------- -------u-	100100----- ---010n011010	0	0.00	0.00	22.00
36	-------- --------	n100-0-1---- ---0-1-11010n0	0	−1.00	−1.00	22.00
37	-------- --------	010111----- ----100001001	0	0.00	0.00	21.00
38	-------- --------	u0001------ ----0-0001101	0	0.00	0.00	21.00
39	-------- --------	u00-0-0----- ----101010100	0	0.00	0.00	21.00
40	-------- --------	11110------ ----010000101	0	0.00	0.00	21.00

41	-------- --------	0011------ ----011010010	0	0.00	0.00	21.00
42	-------- --------	00-1-0----- ----01-001100	0	0.00	0.00	21.00
43	-------- --------	1010------ ----001111100	0	0.00	0.00	21.00
44	-------- --------	000------- ---11-0011100	0	0.00	0.00	21.00
45	-------- --------	0-1-0------ ----1-1100101	0	0.00	0.00	21.00
46	-------- --------	000------- ---0--010010	0	0.00	0.00	21.00
47	-------- --------	11------- ----001010101	0	0.00	0.00	21.00
48	-------- --------	-0-0------ ---1100111001	0	0.00	0.00	21.00
49	-------- --------	10------- ----0-1111110	0	0.00	0.00	21.00
50	-------- --------	0-------- ---11-1100n10	0	−1.00	−1.00	21.00
51	-------- -------n-	1-0------- ---10u010110-	0	0.00	0.00	20.00
52	-------- --------	1------- ----0000001u10	0	−1.00	−1.00	20.00
53	-------- --------	-------- ---011011n10u	0	−2.00	−2.00	19.00
54	-------- ------n--	-1------- ----u-11011-u	0	−1.00	−1.00	17.00
55	-------- --------	-------- ---111011u01u	0	−1.00	−1.00	16.00
56	-------- --------	-------- ---01-00n10u-	0	−2.00	−1.91	15.00
57	-------- ------n--	1------- ----u101111-u-	0	−1.00	−1.00	13.00
58	-------- --------	-------- ---10-00un0u-	0	−2.00	−1.83	12.00
59	-------- ------u--	-------- ---0n01u11u-	0	−2.00	−1.87	10.00
60	-------- -----u---	-------- ---n-0-111xu-0	0	−2.00	−1.00	8.00
61	-------- --------	-------- ---0100u01ux-	0	−2.00	−1.00	6.00
62	-------- --------	-------- ----0-u11n-x-	0	−3.00	−1.89	4.00
63	-------- -----u---	-------- --n-10110-u-n-	0	−1.00	−1.00	1.00
64	-------- --------					

Table 14: Third characteristic used for the second block of the 64-step collision [4]

References

[1] Secure hash standard (SHS). Federal information processing standards publication; FIPS PUB 180-4. U.S. Dept. of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, 2012.

[2] SHA-3 standard : permutation-based hash and extendable-output functions. Federal information processing standards publication; 202. U.S. Dept. of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, 2015.

[3] Auth0. Adding salt to hashing: A better way to store passwords, 2025.

[4] Christophe De Cannière and Christian Rechberger. Finding sha-1 characteristics: General results and applications. In Xuejia Lai and Kefei Chen, editors, Advances in Cryptology – ASIACRYPT 2006, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] eMule Project Team. eMule Project. `https://www.emule-project.com/home/perl/general.cgi?l=1`, 2025. Accessed: 14 January 2025.

[6] Ethereum Community. Rlp - ethereum wiki. `https://github.com/ethereum/wiki/wiki/RLP/026ed2e8ef5514e47558ec99a336f2f65c45116a`, 2025. Accessed: February 05, 2025.

[7] Ethereum Foundation. Transactions - ethereum. `https://ethereum.org/en/developers/docs/transactions/`, 2024. Accessed: February 05, 2025.

[8] Ethereum Foundation. Ethash, 2025.

[9] GeeksforGeeks. SHA-1 Hash in Java, 2024. Last Updated: 18 Jul, 2024.

[10] Git Project. Hash function transition, 2025. Accessed on 21 January 2025.

[11] Go Ethereum Book. Create raw transaction - ethereum development with go, 2025. Accessed: February 05, 2025.

[12] Ian G. Gosling. eDonkey/ed2k: Study of A Young File Sharing Protocol. Giac paper, SANS Institute Information Security Reading Room, 2003.

[13] Roger A. Grimes. Why aren't we using SHA-3?, 2025. Accessed on 21 January 2025.

[14] Deepak Gupta. Md5: Understanding its uses, vulnerabilities, and why it's still around. `https://guptadeepak.com/md5-understanding-its-uses-vulnerabilities-and-why-its-still-around/`, 2025. Accessed: January 14, 2025.

[15] IBM. Hashing functions. `https://www.ibm.com/docs/en/psfa/7.2.1?topic=toolkit-hashing-functions`, 2025. Accessed: 14 January 2025.

[16] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, Advances in Cryptology—CRYPTO 2004, volume 3152 of Lecture Notes in Computer Science, pages 306–316, Berlin, 2004. Springer.

[17] Burt Kaliski. The MD2 Message-Digest Algorithm. RFC 1319, April 1992.

[18] Lars R. Knudsen, John Erik Mathiassen, Frédéric Muller, and Søren S. Thomsen. Cryptanalysis of md2. Journal of cryptology, 23(1):72–90, 2010.

[19] Spencer Miller. From scratch: An introduction to hash functions. 2020. REU Paper.

[20] National Institute of Standards and Technology. Recommendation for transitioning the use of cryptographic algorithms and key lengths. Special Publication NIST Special Publication 800-131A, National Institute of Standards and Technology, January 2011.

[21] Okta. Md5: A brief overview. `https://www.okta.com/identity-101/md5/`, 2025. Accessed: January 14, 2025.

[22] Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2010.

[23] pyrlp contributors. Tutorial — pyrlp 4.0.1 documentation, 2025. Accessed: February 05, 2025.

[24] Restack. Secure hashing techniques: Knowledge answer sha-3, 2023.

[25] Restack. Secure hashing techniques: Sha 256 vs sha 512, 2025.

[26] Ronald L. Rivest. The md4 message digest algorithm. In Alfred J. Menezes and Scott A. Vanstone, editors, Advances in Cryptology-CRYPTO' 90, pages 303–311, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[27] Ronald L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[28] Shashwat Sharma. A comprehensive study of cryptographic hash functions. In Conference Paper, June 2024. Supervised by Ayush Mishra, Scientist (B), DMSRDE, DRDO.

[29] TheLeopardsH. SHA3-Python-3-implementation, 2024. Accessed: 2025-03-21.

[30] S. Turner and L. Chen. Use of the MD4 Algorithm is Discouraged. RFC 6150, RFC Editor, March 2011.

[31] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions md4 and ripemd. In ADVANCES IN CRYPTOLOGY - EUROCRYPT 2005,PROCEEDINGS, volume 3494 of Lecture Notes in Computer Science, pages 1–18. Springer Nature, BERLIN, 2005.

[32] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In ADVANCES IN CRYPTOLOGY - EUROCRYPT 2005,PROCEEDINGS, volume 3494 of Lecture Notes in Computer Science, pages 19–35. Springer Nature, BERLIN, 2005.