HAS-V: A New Hash Function with Variable Output Length

Nan Kyoung Park*, Joon Ho Hwang, and Pil Joong Lee

Pohang University of Science and Technology(POSTECH),
Pohang, 790-784, Korea
nkpark@mail.posdata.co.kr, jhhwang@oberon.postech.ac.kr,
pjl@postech.ac.kr

Abstract. Hash functions play an essential role in many areas of cryptographic applications such as digital signature, authentication, and key derivation. In this paper, we propose a new hash function with variable output length, namely HAS-V, to meet the needs of various security levels desired among different applications. A great deal of attention was paid to balance the characteristics of security and performance. The use of message expansion, 4-variable Boolean functions, variable and fixed amounts of shifts, and interrelated parallel lines provide a high level of security for HAS-V. Experiments show that HAS-V is about 19% faster than SHA-1, 31% faster than RIPEMD-160, and 26% faster than HAVAL on a Pentium PC.

1 Introduction

A hash function is a function that maps an input with an arbitrary length to an output with a specific length, referred to as a *hash-code*. A *one-way* hash function must obey the preimage and second preimage resistance properties. Furthermore, most cryptographic applications require the hash function to satisfy the collision resistance property, which is a stronger constraint than the former two properties.

The collision of a hash function can be found by the birthday paradox or square root attack with $2^{n/2}$ operations where n is the length of the hash-code [18]. In order to prevent such attacks, the length of the hash-code should be no less than 128 bits. However, the works of van Oorschot and Wiener [12], on special-purpose hardware design for parallel collision search, suggest that the minimum length of the hash-code should be 160 bits. Ever since Damgård [6] established the design principles of a hash function, which included the fact that the collision resistance of the compression function is sufficient for the collision resistance of the hash function, almost all hash functions follow these principles.

There are three main categories of hash functions, namely hash functions based on block ciphers, hash functions based on modular arithmetic, and dedicated hash functions [13]. Most early hash functions were based on block ciphers. However, the modification of block ciphers into hash functions resulted

* Nan Kyoung Park is now with POSDATA, Pun-Dang Gu, Sung-Nam, 463-050, Korea

D.R. Stinson and S. Tavares (Eds.): SAC 2000, LNCS 2012, pp. 202–216, 2001.

in security weaknesses in collision search as well as performance deterioration. The slow performance problem is even more serious for hash functions based on modular arithmetic and serious doubts have been raised about their security. Consequently, the need for fast and secure hash functions resulted in dedicated hash functions. These functions are custom designed to achieve the goals of a cryptographic hash function. Among the numerous dedicated hash functions that are in use today, the MD4-family hash functions are the most widely used and analyzed family of hash functions. MD4 [14], MD5 [15], and RIPEMD-160 [9] are popular examples of MD4-family hash functions.

There have been several attacks on the MD4-family hash functions [1,2,5,7]. Among these attacks, a series of Dobbertin's attacks are becoming a real threat on practical applications. Fortunately, SHA-1 [11] and RIPEMD-160 are considered to be secure against these attacks [8]. The main distinction of SHA-1 is the message expansion process, where the message words used in the different rounds are computed as the sum of the previous message words and circular shift by 1-bit. This prevents making local changes, which is confined to a few bits, and accordingly individual message bits influence the calculations at large number of places. RIPEMD-160 is an enhanced version, in a way to be resistant against Dobbertin's attacks, of RIPEMD. Its main improvements are the increase in the number of rounds from 3 to 5 and the two parallel lines were modified to have a different message ordering, Boolean functions, and shift amounts.

Recently, much progress has been made in the software implementation of MD4-family hash functions [3,4]. Analyses show that the structures of MD4-family hash functions possess a higher instruction-level parallelism than current general-purpose computer architecture can provide. Among the MD4-family hash functions, it is known that the critical path to compute the step function of SHA-1 is shorter than any other MD4-family hash functions and the organization of RIPEMD-160 in two independent lines will become much useful in the near future.

In the remainder of this paper, we propose a new MD4-family hash function that produces a variable length hash-code, namely HAS-V. In Section 2, we present details on why a hash function with variable length hash-code is needed. In Section 3, the terminologies and notations used are defined. In Section 4, a description of the newly proposed hash function is given. In Section 5, we discuss the underlying design principles of HAS-V based on performance and security aspects. Performance comparison is given in Section 6, and concluding remarks are given in Section 7. The pseudo-code and the test values of HAS-V are given in the Appendix.

2 Motivation

The length of the hash-code is an important factor directly connected to the security of the hash function. Assuming that there are no unexpected design flaws known in a hash function, the complexity of finding a collision is heavily dependent on the length of the hash-code. The length of the hash-code must

be long enough to provide explicit security, but it must not be unnecessarily long to sacrifice efficiency of the entire system. Consequently, the length of the hash-code is a relative factor to the computing power possessed by the opponent. Therefore, the length of the hash-code is meant to vary from time to time as technology advances and information security broadens its area of application. It seems inappropriate to fix the length of the hash-code when various levels of security are desired among different applications.

KCDSA [10], Korea Certificate-based Digital Signature Algorithm, is an example of a cryptographic application where a variable length of hash-code is needed. KCDSA employs variable length domain parameters in order to fulfill the various security needs in different applications. In the case of KCDSA, there is a need for a hash function that can produce a variable hash-code of up to 256 bits in order to fully utilize the flexible security level of KCDSA.

However, most conventional hash algorithms are designed to produce a specific length of hash-code, such as 128 bits for MD4 and MD5, and 160 bits for SHA-1 and RIPEMD-160. Among the well-known hash functions, HAVAL [19] is the only hash function that can produce a variable length hash-code. Although HAVAL is still considered to be secure, there are some concerns that a suitable modification of MD4 attack could be applied to HAVAL with 3 passes. Furthermore, HAVAL suffers from performance deterioration in CISC processors due to the excessive number of chaining variables used. There exists an optional extension of RIPEMD-128 and RIPEMD-160 to produce 256-bit and 320-bit hash-code. However, these methods do not provide any increase in security level, merely an increase in the length of the hash-code. This gives a clear motivation to design a new hash function with variable length hash-code, which is both efficient and secure.

Information security is becoming an inevitable part of our society, and therefore information technology must provide services to fulfill the needs of various people. The need for variable length hash-code can be explained in an analogous way. Moreover, the ever-increasing nature of computing power will eventually threaten the length of the hash-codes used in many applications today. Instead of redesigning a new hash function in such events, the use of a single hash function with a variable length hash-code seems to be a cost-effective and convenient way of increasing the security level.

3 Terminology and Notations

The use of *byte* in this paper implies an 8-bit quantity, *word* implies a 32-bit quantity, and *block* implies a 1024-bit quantity, which is the input size of the compression function. We assume a byte with the most significant bit of each byte listed first and a block with the least significant byte of each block given first. Throughout this paper, the following notations will be used:

- $+$: addition of words, i.e. addition by modulo-2^{32}.
- $X^{\ll s}$: the circular left shift of X by s bit positions.

Table 1. Characteristics of HAS-V

Length of Input Block (bits)	1024
Length of Output (bits)	$128 \sim 320$
Number of Rounds	10
Number of Chaining Variables	10
Number of Steps	200

Table 2. Initial values

A	B	C	D	E
67452301	$efcdab89$	$98badcfe$	10325476	$c3d2e1f0$
F	G	H	I	J
$8796a5b4$	$4b5a6978$	$0f1e2d3c$	$a0b1c2d3$	$68794e5f$

- $\neg$: the bitwise complement operation.
- $\vee, \wedge, \oplus$: the bitwise OR, AND, and XOR operation. ($X \wedge Y$ is also denoted as XY for simplicity)

4 Description of HAS-V Algorithm

The basic structure of the compression function of HAS-V is two parallel lines, denoted as the X-line and the Y-line, consisting of 100 steps each. Each line is composed of 5 rounds, where each round consists of 20 steps, and maintains 5 words of chaining variables, a total of 10 chaining variables for the entire compression function. The two parallel lines are interrelated by swapping the contents of the entire chaining variables in the X-line and the Y-line after each round. The message words used in the compression function are 32 words, or a 1024 bit block, of the input message and 8 additionally generated words each round by message expansion, a total of 40 words for the entire compression function. The characteristics of the structure of HAS-V are summarized in Table 1.

Append Padding Bits and Length: The message is padded so that its length is congruent to 952 modulo 1024. Padding is performed by appending a single "1" bit and necessary zero bits to satisfy the above constraints. The remaining 72 bits, in order to be a multiple of 1024 bits, is filled by appending the desired length of the hash-code represented in *bytes* and the length of the input coded in modulo 2^{64} represented in *bits*.

Initial Value of the Chaining Variables: The initial values of the chaining variables used in HAS-V are given in Table 2. A, B, C, D, E are the chaining variables of the X-line and F, G, H, I, J are the chaining variables of the Y-line.

Message Preparation and Expansion: The length of the input block used in each compression function is 1024 bits. The upper 512 bits consist of 16 words,

Table 3. Message combination to generate extra messages

Index	Round 1	Round 2	Round 3	Round 4	Round 5
16	0, 1, 2, 3	3, 6, 9, 12	12, 5, 14, 7	7, 2, 13, 8	15, 9, 5, 3
17	4, 5, 6, 7	15, 2, 5, 8	0, 9, 2, 11	3, 14, 9, 4	12, 8, 6, 2
18	8, 9, 10, 11	11, 14, 1, 4	4, 13, 6, 15	15, 10, 5, 0	13, 11, 7, 1
19	12, 13, 14, 15	7, 10, 13, 0	8, 1, 10, 3	11, 6, 1, 12	14, 10, 4, 0

$X[0]$, $X[1]$, ..., $X[15]$ that are used in the X-line and the remaining lower 512 bits consist of 16 words, $Y[0]$, $Y[1]$, ..., $Y[15]$ that are used in the Y-line. Each line of the compression function then additionally generates 4 message words in each round by message expansion. The extra messages are created by the XOR of 4-word combinations chosen from its present line. The word combinations used in message expansion differ for every round of the compression function and are given in Table 3.

For example, the message word $X[17]$ used in round 2 of the X-line is generated as follows:

$$X[17] = X[15] \oplus X[2] \oplus X[5] \oplus X[8].$$

The rest of the message words, $X[16]$, $X[18]$, $X[19]$, can be expanded in a similar way. The message words in the opposite Y-line, $Y[16]$, $Y[17]$, $Y[18]$, $Y[19]$ can be derived in an analogous way.

Ordering of the Message Words: Each message word among the 20 message words, 16 initial input message words and 4 expanded message words, is applied to a single step in each line. The order of message words used in both lines is equivalent. The ordering of the message words for each round is given in Table 4. The extra messages generated by message expansion, $X[16]$, $X[17]$, $X[18]$, $X[19]$, are applied to steps 10, 15, 0, and 5, respectively, in each round.

Step Operation: The operation in each step is equivalent in both the X-line and the Y-line. The step operation of the X-line is given below.

$$T \leftarrow A^{\ll s} + f(B, C, D, E) + X + K,$$

$$E \leftarrow D \ ; \ D \leftarrow C \ ; \ C \leftarrow B^{\ll 30} \ ; \ B \leftarrow A \ ; \ A \leftarrow T.$$

Here f, s, and K are the Boolean function, shift amount, and additive constant, respectively. The Boolean function and constant differ for every round of the compression function, whereas, the shift amount varies for every step within a single round of the compression function.

Boolean Function: The following 5 Boolean functions are used in HAS-V.

$$f_0(x, y, z, u) = xy \oplus \neg xz \oplus yu \oplus zu,$$

Table 4. Message ordering

Step	Round 1	Round 2	Round 3	Round 4	Round 5
0	18	18	18	18	18
1	0	3	12	7	15
2	1	6	5	2	9
3	2	9	14	13	5
4	3	12	7	8	3
5	19	19	19	19	19
6	4	15	0	3	12
7	5	2	9	14	8
8	6	5	2	9	6
9	7	8	11	4	2
10	16	16	16	16	16
11	8	11	4	15	13
12	9	14	13	10	11
13	10	1	6	5	7
14	11	4	15	0	1
15	17	17	17	17	17
16	12	7	8	11	14
17	13	10	1	6	10
18	14	13	10	1	4
19	15	0	3	12	0

Table 5. Order of Boolean function

Line	Round 1	Round 2	Round 3	Round 4	Round 5
X	f_0	f_1	f_2	f_3	f_4
Y	f_4	f_3	f_2	f_1	f_0

$$f_1(x, y, z, u) = xz \oplus y \oplus u,$$
$$f_2(x, y, z, u) = xy \oplus \neg xu \oplus z,$$
$$f_3(x, y, z, u) = x \oplus yz \oplus u (= f_1(y, x, z, u)),$$
$$f_4(x, y, z, u) = \neg xy \oplus xz \oplus yu \oplus zu (= f_0(x, z, y, u)).$$

The Boolean functions are applied, in each line, as in Table 5

Shifts: For both lines, the shift amount is given in Table 6. The period of the shift amount is 20 steps in the compression function.

Constants: Additive constants are taken as the integer parts of the numbers given in Table 7.

Swapping of the Chaining Variables: The contents of the chaining variables in the X-line and the Y-line are swapped after every round.

$$A \leftrightarrow F \ ; \ B \leftrightarrow G \ ; \ C \leftrightarrow H \ ; \ D \leftrightarrow I \ ; \ E \leftrightarrow J$$

Table 6. Shift amount

Step mod 20	0	1	2	3	4	5	6	7	8	9
s	5	11	7	13	15	6	13	9	5	11
Step mod 20	10	11	12	13	14	15	16	17	18	19
s	7	12	8	15	13	8	15	6	7	14

Table 7. Constants

Line	Round 1	Round 2	Round 3	Round 4	Round 5
X	0	$5a827999$ $\lfloor 2^{30}\sqrt{2}\rfloor$	$6ed9eba1$ $\lfloor 2^{30}\sqrt{3}\rfloor$	$8f1bbcdc$ $\lfloor 2^{30}\sqrt{5}\rfloor$	$a953fd4e$ $\lfloor 2^{30}\sqrt{7}\rfloor$
Y	$\lfloor 2^{30}\sqrt{7}\rfloor$	$\lfloor 2^{30}\sqrt{5}\rfloor$	0	$\lfloor 2^{30}\sqrt{2}\rfloor$	$\lfloor 2^{30}\sqrt{3}\rfloor$

Final Feedforward Process of Chaining Variables: Let us assume that the contents of the chaining variables before the compression function are $A \sim J$, and let the contents of the chaining variables after the compression function be $AA \sim JJ$. Then the updated contents of the chaining variables, or the output of the compression function, are given as shown.

$$A+ = AA,\ B+ = BB,\ C+ = CC,\ D+ = DD,\ E+ = EE,$$
$$F+ = FF,\ G+ = GG,\ H+ = HH,\ \ I+ = II,\quad J+ = JJ.$$

Output Tailoring: In the case of 320-bit hash-code, the output is given as the contents of the 10 chaining variables concatenated, i.e. $A||B||C||D||E||F||G||H||I||J$. Otherwise, when the length of the hash-code is required to be shorter than 320 bits, it must be tailored into a string of specified length, denoted as $O_0||O_1||\ldots||O_t (t = 3, 4, \ldots, 8)$. The contents of O_i differ in each case of various lengths of hash-codes. Let us denote a t-bit string as $X^{[t]}$ to explicitly indicate the length of X.

- Case 1 (128-bit hash-code): The 32-bit chaining variables E and J can be divided as follows:

$$E = E_1^{[16]} E_0^{[16]},\ J = J_1^{[16]} J_0^{[16]}.$$

  O_i is calculated as follows:

$$O_0 = A + F + E_1^{[16]},\quad O_1 = B + G + E_0^{[16]},$$
$$O_2 = C + H + J_1^{[16]},\quad O_3 = D + I + J_0^{[16]}.$$

- Case 2 (160-bit hash-code): O_i is calculated as follow.

$$O_0 = A + F,\ O_1 = B + G,\ O_2 = C + H,\ O_3 = D + I,\ O_4 = E + J.$$

- Case 3 (192-bit hash-code): The 32-bit chaining variables D, E, I, and J can be divided as follows:

$$D = D_2^{[11]} D_1^{[11]} D_0^{[10]},\quad E = E_2^{[11]} E_1^{[11]} E_0^{[10]},$$
$$I = I_2^{[11]} I_1^{[11]} I_0^{[10]},\quad J = J_2^{[11]} J_1^{[11]} J_0^{[10]}.$$

O_i is calculated as follows:

$$\begin{aligned} O_0 &= A + (J_2^{[11]} || I_1^{[11]}), \quad O_1 = B + (J_1^{[11]} || I_0^{[10]}), \\ O_2 &= C + (J_0^{[10]} || I_2^{[11]}), \quad O_3 = F + (E_2^{[11]} || D_1^{[11]}), \\ O_4 &= G + (E_1^{[11]} || D_0^{[10]}), \quad O_5 = H + (E_0^{[10]} || D_2^{[11]}). \end{aligned}$$

– Case 4 (224-bit hash-code): The 32-bit chaining variables E, I, and J can be divided as follows:

$$E = E_2^{[11]} E_1^{[11]} E_0^{[10]},\ I = I_3^{[8]} I_2^{[8]} I_1^{[8]} I_0^{[8]},\ J = J_3^{[8]} J_2^{[8]} J_1^{[8]} J_0^{[8]}.$$

O_i is calculated as follows:

$$\begin{aligned} O_0 &= A + (J_3^{[8]} || I_2^{[8]}), \quad O_1 = B + (J_2^{[8]} || I_1^{[8]}), \\ O_2 &= C + (J_1^{[8]} || I_0^{[8]}), \quad O_3 = D + (J_0^{[8]} || I_3^{[8]}), \\ O_4 &= F + E_2^{[11]},\ O_5 = G + E_1^{[11]},\ O_6 = H + E_0^{[10]}. \end{aligned}$$

– Case 5 (256-bit hash-code): The 32-bit chaining variables E and J can be divided as follows:

$$E = E_3^{[8]} E_2^{[8]} E_1^{[8]} E_0^{[8]},\ J = J_3^{[8]} J_2^{[8]} J_1^{[8]} J_0^{[8]}.$$

O_i is calculated as follows:

$$\begin{aligned} O_0 &= A + J_3^{[8]}, \quad O_1 = B + J_2^{[8]}, \\ O_2 &= C + J_1^{[8]}, \quad O_3 = D + J_0^{[8]}, \\ O_4 &= F + E_3^{[8]}, \quad O_5 = G + E_2^{[8]}, \\ O_6 &= H + E_1^{[8]}, \quad O_7 = I + E_0^{[8]}. \end{aligned}$$

– Case 6 (288-bit hash-code): The 32-bit chaining variable J can be divided as follows:

$$J = J_4^{[7]} J_3^{[7]} J_2^{[6]} J_1^{[6]} J_0^{[6]}.$$

O_i is calculated as follow.

$$\begin{aligned} O_0 &= A + J_4^{[7]},\ O_1 = B + J_3^{[7]}, \\ O_2 &= C + J_2^{[6]},\ O_3 = D + J_1^{[6]},\ O_4 = E + J_0^{[6]}, \\ O_5 &= F,\ O_6 = G,\ O_7 = H,\ O_8 = I. \end{aligned}$$

5 Design Rationales and Security Aspects

In this section, we discuss the underlying principles that were considered in the process of designing HAS-V. A great deal of attention was paid to balance the characteristics of security and performance. Security matters are considered based on previous attacks on hash functions and employ firm design philosophies of the previous hash functions. Performance matters are considered in the area of hardware support and algorithmic parallelism.

Number of Chaining Variables in Step Operation: One of the big differences between CISC processors, including the Intel 80x86 family and the Motorola 680x0, and RISC processors, including SPARC, MIPS, PA-RISC, PowerPC, and Alpha, is the number of on-chip general-purpose registers [4]. Generally RISC processors have enough general-purpose registers to load all the chaining variables on the register. However, due to their complex instruction set, CISC processors usually suffer from a shortage of general-purpose registers. In the case of a Pentium processor, there are only 7 general-purpose registers on its chip. Assuming at least 1 register is needed for temporary storage, no more than 6 registers can be used in an iterative step operation to perform without deterioration. HAVAL uses 8 chaining variables in its step operation and could suffer from performance deterioration in CISC processors. Therefore, in the design of HAS-V, a twin structure was employed and the number of chaining variables used in the step operation was chosen to be 5, so that the entire set of chaining variables could be loaded on the processor during the iterative process. It may seem at first that the two lines should be processed simultaneously since the chaining variables are swapped after every round. However, the two lines of the compression function can be processed independently as the entire chaining variables are swapped instead of just a portion of it. In an implementation point of view, the chaining variables are not actually swapped. Instead, the message words, Boolean function, and constants used in the step operation of round 2 and round 4 are replaced by the co rresponding ones of the opposite line. This can be better understood by referring to the pseudo-code in Appendix A.

Process of Message Words: Early attacks on MD4 and MD5 were based on the weakness of the rather straightforward usage of the message words. An attack on the last two rounds of MD4 [1] and the cryptanalysis of MD4 [7] fall into this category of attack. A single message word of the input is only used once in every round of MD4 and MD5. This seems to provide vulnerability for inner collisions. A concept of message expansion was introduced in SHA-1, which provided a concrete security level against these sorts of attacks. This attractive property of message expansion was employed in the design of HAS-V. However, the generation of 64 message words from 16 message words of input seemed to load a heavy burden on the performance of SHA-1. In HAS-V, we have generated 20 message words from 16 message words for each line. This allows HAS-V to stay within a fairly good performance range, while providing enough diffusion from a single message word.

Step Operation: The structure of the step operation is an important factor in determining the performance factor. It is known that the step operation used in SHA-1 possesses a natural algorithmic parallelism in its compression function [4]. This arises from the fact that the updated chaining variable is not used in the Boolean function of the next step operation, which has the effect of reducing the critical path length. This mechanism has been employed in the step operation of HAS-V to provide a further advantage in performance. Other characteristics

in the step operation of HAS-V are the use of 4-variable Boolean functions and the use of a shift amount, which varies for each step within a single round of the compression function. The variable shift amount seems to provide better immunity against attacks such as differential collision in SHA-0 [5]. The generalization of inner collisions to a full compression function seemed to be harder with variable shift amounts.

Boolean Function: Previous attacks such as the collision attack on the compression function of MD5 [2] and differential attacks uses the linear approximation of Boolean functions. HAS-V employs 4-variable Boolean functions, whereas most MD4-family hash functions employ 3-variable Boolean functions. Having an extra variable in the Boolean function increases the complexity of a linear approximation and the computational cost of the Boolean function. Therefore, it is important to keep the balance between the needs for non-linearity and the loss of computational efficiency, while constructing a Boolean function. The computation of Boolean functions in HAS-V require about 3 or 4 unit operations[1], whereas the 3-variable Boolean functions used in other hash functions require about 2 or 3 unit operations. Therefore, by using 4-variable Boolean functions and omitting a single addition in the step function, we can improve the security aspects of HAS-V without performance deterioration. Among the numerous 4-variable Boolean functions, we have selected ones that are 0-1 balanced, satisfy SAC, and have a high non-linearity [16,17] to be used in HAS-V.

Output Tailoring: The output of HAS-V must provide a variable output from 128 bits to 320 bits, incrementing in multiples of 32 bits. We have modified the output tailoring method of HAVAL to produce the desired length of output, while providing a fair share to all of the chaining variables. Moreover, this process must not put unnecessary burden on the overall workload to deteriorate the performance.

Endianness: As with most of the MD4-family hash functions, the newly proposed hash function is optimized for 32-bit architecture processors. HAS-V favors 'little-endian' architectures. Processors with 'big-endian' architectures have to byte-reverse each word before processing, and since the big-endian processors are generally faster, it was decided to let them do the reversing it. This incurs a performance penalty of about 25%.

[1] The number of unit operations for Boolean functions can be defined by the least number of bit-wise operations such as $\neg$, $\wedge$, $\vee$, or $\oplus$, which are required to compute the Boolean functions. It can be regarded as a performance measure. If we modify the Boolean functions f_0, f_1, and f_2, of HAS-V, the number of unit operations can be found. For example, since the truth table of f_0 is equivalent to that of $(x \oplus u)(y \oplus z) \oplus z$, the number of unit operations of f_0 is less than 4. In a similar way, those of f_1 and f_2 are 3 and 4, respectively

Table 8. Comparison of speed performance

Algorithm	Performance(Mbits/sec)	
	Pentium III (600MHz)	Ultra 2 SPARC (300MHz)
	Microsoft Visual C++	GNU C
MD5	41.95	22.04
SHA-1	22.58	10.54
RIPEMD-160	19.38	8.94
HAVAL(5 PASS)	20.64	11.01
HAS-V	27.86	12.58

6 Performance Evaluation

In this section we compare the performance of MD5, SHA-1, RIPEMD-160, HAVAL, and HAS-V. Output tailoring was ignored in both HAVAL and HAS-V, since it only occupies a negligible amount of time. Implementations were written in the C language and there was no optimization done in any way. The implementation was done solely for comparative reasons. The performance results were extracted by hashing 64Mbytes of data using an 8Kbyte buffer. Table 8 shows the results of our experiment. The results show that HAS-V has better performance than SHA-1, RIPEMD-160, or HAVAL with 5 passes in both environments.

The step operation of HAS-V consists of 3 additions, 2 circular shifts, and a Boolean function. Since the Boolean function consists of 4 unit operations, a single step operation will consist of 9 unit operations, assuming both addition and circular shift to be equivalent to unit operation. The total number of unit operations for generating the extra messages is 2(lines)×5(rounds)×4(messages)× 3(unit operations) = 120(unit operations). Therefore the number of unit opeations to hash 1024 bit block is given below.

$$1(\text{block}) \times 200(\text{steps}) \times 9(\text{step operation}) + 120(\text{message expansion})$$

$$= 1920(\text{unit operations})$$

In the case of RIPEMD-160, the total number of unit operation to hash 1024 bit block is given below.

$$2(\text{block}) \times 160(\text{steps}) \times 9(\text{step operation})$$

$$= 2880(\text{unit operations})$$

This is about 33% more operation than HAS-V. This fact can also be seen in Table 8 where HAS-V is 31% faster than RIPEMD-160 on a Pentium PC.

7 Conclusion

We have proposed a new hash function with a variable length hash-code, namely HAS-V. The design was made such that it is both secure and efficient in most

computing environments. We expect that our results will broaden the use of KCDSA or any other cryptographic application that uses hash functions. We believe that the variable nature of the hash-code length will anticipate the needs of various practical applications.

Acknowledgements

Authors wish to express thanks to an anonymous referee of FSE2000 for pointing out a weakness of HAS-V in the previous version. This research was supported mainly by KISA(Korea Information Security Agency), partially by Brain Korea 21 and Com^{2}MaC(Combinatorial and Computational Mathematics Center, POSTECH).

A Pseudo-Code of HAS-V

A.1 Definition

Let the input message string be consisted of t 1024-bit blocks, represented as $\{X_i[j], Y_i[j]\}(0 \le i < t, 0 \le j < 16)$.

$$
\begin{array}{ll}
f_j(x,y,z,u) = xy \oplus \neg xz \oplus yu \oplus zu & (0 \le j \le 19) \\
f_j(x,y,z,u) = xz \oplus y \oplus u & (20 \le j \le 39) \\
f_j(x,y,z,u) = xy \oplus \neg xu \oplus z & (40 \le j \le 59) \\
f_j(x,y,z,u) = x \oplus yz \oplus u & (60 \le j \le 79) \\
f_j(x,y,z,u) = \neg xy \oplus xz \oplus yu \oplus zu & (80 \le j \le 99) \\
g_j(x,y,z,u) = \neg xy \oplus xz \oplus yu \oplus zu & (0 \le j \le 19) \\
g_j(x,y,z,u) = x \oplus yz \oplus u & (20 \le j \le 39) \\
g_j(x,y,z,u) = xy \oplus \neg xu \oplus z & (40 \le j \le 59) \\
g_j(x,y,z,u) = xz \oplus y \oplus u & (60 \le j \le 79) \\
g_j(x,y,z,u) = xy \oplus \neg xz \oplus yu \oplus zu & (80 \le j \le 99)
\end{array}
$$

$$
\begin{array}{lll}
K_j = 00000000 & K'_j = a953fd4e & (0 \le j \le 19) \\
K_j = 5a827999 & K'_j = 8f1bbcdc & (20 \le j \le 39) \\
K_j = 6ed9eba1 & K'_j = 00000000 & (40 \le j \le 59) \\
K_j = 8f1bbcdc & K'_j = 5a827999 & (60 \le j \le 79) \\
K_j = a953fd4e & K'_j = 6ed9eba1 & (80 \le j \le 99)
\end{array}
$$

s(j) = 5,11,7,13,15,6,13,9,5,11,7,12,8,15,13,8,15,6,7,14

$$
\begin{array}{ll}
m(j) = 18,0,1,2,3,19,4,5,6,7,16,8,9,10,11,17,12,13,14,15 & (0 \le j \le 19) \\
m(j) = 18,3,6,9,12,19,15,2,5,8,16,11,14,1,4,17,7,10,13,0 & (20 \le j \le 39) \\
m(j) = 18,12,5,14,7,19,0,9,2,11,16,4,13,6,15,17,8,1,10,3 & (40 \le j \le 59) \\
m(j) = 18,7,2,13,8,19,3,14,9,4,16,15,10,5,0,17,11,6,1,12 & (60 \le j \le 79) \\
m(j) = 18,15,9,5,3,19,12,8,6,2,16,13,11,7,1,17,14,10,4,0 & (80 \le j \le 99)
\end{array}
$$

$a_j(k) = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$ $(j = 0)$
$a_j(k) = 3, 6, 9, 12, 15, 2, 5, 8, 11, 14, 1, 4, 7, 10, 13, 0$ $(j = 20)$
$a_j(k) = 12, 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3$ $(j = 40)$
$a_j(k) = 7, 2, 13, 8, 3, 14, 9, 4, 15, 10, 5, 0, 11, 6, 1, 12$ $(j = 60)$
$a_j(j) = 15, 9, 5, 3, 12, 8, 6, 2, 13, 11, 7, 1, 14, 10, 4, 0$ $(j = 80)$

A.2 Pseudo-Code

Initial values of the chaining variables
$h_0 = 67452301; h_1 = efcdab89; h_2 = 98badcfe; h_3 = 10325476; h_4 = c3d2e1f0;$
$h_5 = 8796a5b4; h_6 = 4b5a6978; h_7 = 0f1e2d3c; h_8 = a0b1c2d3; h_9 = 68794e5f;$

for $i = 0, \ldots, t-1$\{
  $A = h_0; B = h_1; C = h_2; D = h_3; E = h_4;$
  $F = h_5; G = h_6; H = h_7; I = h_8; J = h_9;$
  for $j = 0$ to 99\{
    if$(j = 0, 40, 80)$\{
      for $k = 0$ to 3\{
        $X_i[16+k] = X_i[a_j(4k)] \oplus X_i[a_j(4k+1)] \oplus X_i[a_j(4k+2)]$
        $\oplus X_i[a_j(4k+3)];$
      \}
    \}
    else if$(j = 20, 60)$\{
      for $k = 0$ to 3\{
        $Y_i[16+k] = Y_i[a_j(4k)] \oplus Y_i[a_j(4k+1)] \oplus Y_i[a_j(4k+2)]$
        $\oplus Y_i[a_j(4k+3)];$
      \}
    \}
    if$(round = 1, 3, 5)$\{
      $T = A^{\lll s(j\%20)} + f_j(B, C, D, E) + X_i[m(j)] + K_j;$
    \}
    else if$(round = 2, 4)$\{
      $T = A^{\lll s(j\%20)} + g_j(B, C, D, E) + Y_i[m(j)] + K'_j;$
    \}
    $E = D; D = C; C = B^{\lll 30}; B = A; A = T;$
  \}
  for $j = 0$ to 99\{
    if$(j = 0, 40, 80)$\{
      for $k = 0$ to 3\{
        $Y_i[16+k] = Y_i[a_j(4k)] \oplus Y_i[a_j(4k+1)] \oplus Y_i[a_j(4k+2)];$
        $\oplus Y_i[a_j(4k+3)];$
      \}
    \}
    else if$(j = 20, 60)$\{
      for $k = 0$ to 3\{
        $X_i[16+k] = X_i[a_j(4k)] \oplus X_i[a_j(4k+1)] \oplus X_i[a_j(4k+2)]$

$$\oplus X_i[a_j(4k+3)];$$

}

}

if($round = 1, 3, 5$){

$T = F^{\ll s(j\%20)} + g_j(G, H, I, J) + Y_i[m(j)] + K'_j;$

}

else if($round = 2, 4$){

$T = F^{\ll s(j\%20)} + f_j(G, H, I, J) + X_i[m(j)] + K_j;$

}

$J = I; I = H; H = G^{\ll 30}; G = F; F = T;$

}

$h_0 += F; h_1 += G; h_2 += H; h_3 += I; h_4 += J;$

$h_5 += A; h_6 += B; h_7 += C; h_8 += D; h_9 += E;$

}

B Test Values of HAS-V

The test values of HAS-V are given in the case of 320-bit hash-code with no output tailoring

```
HAS-V("")=475974be d7ea137d 982d1df5 b2583b1a c4d5941d
8d557bb3 03586742 d8891788 943a9668 a9da68c3
HAS-V("abc")=a70ab818 294865cf 9c9697d6 97152353 70381b83
3f8f1a42 e0150588 8b002e43 05fe6405 519f595c
```

References

1. B.den Boer and A.Bosselaers, An attack on the last two rounds of MD4, *Advances in Cryptology-Crypto'91*, LNCS 576, Springer-Verlag, 1992, pp.194-203.
2. B.den Boer and A.Bosselaers, Collisions for the compression function of MD5, *Advances in Cryptology-Eurocrypt'93*, LNCS 773, Springer-Verlag, 1994, pp.293-304.
3. A.Bosselaers, R.Govaerts and J.Vandewalle, Fast hashing on the Pentium, *Advances in Cryptology-Crypto'96*, LNCS 1109, Springer-Verlag, 1996, pp.298-312.
4. A.Bosselares, R.Govarerts and J.Vandewalle, SHA: a design for parallel architecture, *Advances in Cryptology-Eurocrypt'97*, 1997, pp.348-362.
5. F.Chabaud and A.Joux, Differential collisions in SHA-0, *Advances in Cryptology-Crypto'98*, LNCS 1462, Springer-Verlag, 1998, pp.56-71.
6. I.B.Damgård, A design principle for hash functions, *Proceedings of Crypto '89*, LNCS 435, Springer-Verlag, 1990, pp. 416-427.
7. H.Dobbertin, Cryptanalysis of MD4, *Fast Software Encryption*, LNCS 1039, Springer-Verlag, 1996, pp.53-69.
8. H.Dobbertin, The status of MD5 after a recent attack, *CryptoBytes*, 2(2), Sep. 1996, pp.1-6.
9. H.Dobbertin, A.Bosselaers and B.Preneel, RIPEMD160: A strengthened version of RIPEMD, *Fast Software Encryption*, LNCS1039, Springer-Verlag, 1996, pp.71-82. (An updated and corrected version is available at *ftp.esat.kuleuven.ac.be, /pub/COSIC/bosselae/ripemd/*.)

10. C.H.Lim and P.J.Lee, A study on the proposed Korean digital signature algorithm, *Advances in Cryptology-Asiacrypt'98*, LNCS 1514, Springer-Verlag, 1998, pp. 175-186.
11. NIST, Secure hash standard, *FIPS PUB 180-1*, Department of Commerce, Washington D.C., Apr. 1995.
12. P.C.van Oorchot and M.J.Wiener, Parallel collision search with applications to hash functions and discrete logarithms. *Proc. of 2nd ACM Conference on Computer and Communications Security,* ACM Press, 1994, pp.210-218.
13. B.Preneel, Analysis and design of crytpographic hash functions, *PHD thesis*, Katholiceke University Leuven, 1993.
14. R.Rivest, The MD4 message digest algorithm, *Advances in Cryptology-Crypto'90*, LNCS 537, Springer-Verlag, 1991, pp.303-311.
15. R.Rivest, The MD5 message digest algorithm, *RFC 1321*, Internet Activities Board, Internet Privacy Task Force, Apr. 1992.
16. J.Seberry and X.M.Zhang, Highly nonlinear 0-1 balanced boolean functions satisfying strict avalanche criterion, *Advances in Cryptology-Auscrypt'92*, LNCS 718, Springer-Verlag, 1993, pp.145-154.
17. J.Seberry, X.M.Zhang and Y.Zheng, Nonlinearity and propagation characteristics of balanced boolean functions, *Information and Computation*, 119, 1995, pp.1-13.
18. G.Yuval, How to swindle Rabin, *Cryptologia*, Vol. 3, No. 3, 1979, pp.187-189.
19. Y.Zheng, J.Pieprzyk and J.Sebberry, HAVAL - A one-way hashing algorithm with variable length of output, *Advances in Cryptology-Auscrypt'92*, LNCS 718, Springer-Verlag, 1993, pp.83-104.