SHA-512/256

Shay Gueron [1, 2], Simon Johnson [3], Jesse Walker[4]
[1] *Department of Mathematics, University of Haifa, Israel*
[2] *Mobility Group, Intel Corporation, Israel Development Center, Haifa, Israel*
[3] *Intel Architecture Group, Intel Corporation, USA*
[4] *Security Research Lab, Intel Labs, Intel Corporation, USA*

Abstract

With the emergence of pervasive 64 bit computing we observe that it is more cost effective to compute a SHA-512 than it is to compute a SHA-256 over a given size of data. We propose a standard way to use SHA-512 and truncate its output to 256 bits. For 64 bit architectures, this would yield a more efficient 256 bit hashing algorithm, than the current SHA-256. We call this method SHA-512/256. We also provide a method for reducing the size of the SHA-512 constants table that an implementation will need to store.

Key Words: hash algorithms, SHA-512.

1. Introduction

Robust and fast security functionality is basic tenant for secure computer transactions. Hashing algorithms have long been the poor-man of the community, with their security receiving less attention than standard encryption algorithms and with little attention paid to their speed. The attacks against SHA-1 reversed this situation and there are many new proposals being evaluated in response to the NIST SHA-3 competition. In the aftermath of the SHA-1 attacks the advice NIST produced was to move to SHA-256 [1]. As a result, many standards and products have started to move towards larger hash sizes, although this may be a somewhat protracted process as the SHA-3 competition now adds additional dimensions to feature selection and future supportability issues. This movement does not come without its costs as SHA-256 is about 2.2 times slower than SHA-1.

The reason why SHA-512 is faster than SHA-256 on 64-bit machines is that has 37.5% less rounds per byte (80 rounds operating on 128 byte blocks) compared to SHA-256 (64 rounds operating on 64 byte blocks), where the operations use 64-bit integer arithmetic. The adoption across the breadth of our product range of 64 bit ALU's make it possible to achieve better security using SHA-512 in less time than it takes to compute a SHA-256 hash.

However storing a SHA-512 bit hash is expensive, especially in a constrained hardware environment, such as a state-of-the-art processor. SHA-384 does reduce this storage requirement somewhat by truncating the final result of a SHA-512 to 384 bits. But by truncating the result of SHA-512 operation to 256bits it is possible to balance the cost of providing the necessary additional security/storage against the performance cost of calculating the hash.

We believe that adding SHA-512/256 to the SHA portfolio would provide implementers with performance/cost characteristics hitherto unavailable to them.

2. Performance of SHA-512 and SHA-256

The performance of SHA-256 and SHA-512 depends on the length of the hashed message. Here we provide a summary.

Generally, SHA-256 and SHA-512 can be viewed as a single invocation of an *_init()* function (that initializes the eight 64bit variable h0, h1, h2, h3, h4, h5, h6, h7), followed by a sequence of invocations of an *_update()* function, and an invocation a *_finalize()* function.

The *_finalize()* function itself consists of one or two invocations of *_update()*, depending on the message's length. In addition, there are some operations to create a formatted "last block(s)" (also called "padding").

The *_update()* functions for SHA-256 and SHA-512 are different and, even more importantly, operate on different block sizes: 64 bytes for SHA-256 and 128 bytes for SHA-512.

From a performance standpoint the contribution of the *_init()* function and the last block padding are negligible. Therefore, the performance of SHA-256 and SHA-512 can be quite accurately approximated from the performance of their respective *_update()* functions, and the number of invocations.

The number invocations of the *_update()* function depends on the message length as follows.

SHA-256:

Let M be a message of x bytes, $x = 64n + r$, $0 \le m < 64$.

If $r \le 55$, the number of calls to *_update()* is (n+1)

If $r > 55$, the number of calls to *_update()* is (n+2)

Denote n = floor (x/64), r = x mod 64, and the cost (in CPU cycles) of one SHA-256 *_update()* function by UPDATE256. The number of cycles for computing the SHA-256 of M is approximated by

$$\text{UPDATE256} \cdot (n + 1 + \text{floor}(r/55)) \quad (1)$$

SHA-512:

Let M be a message of y bytes, $y = 128m + s$, $0 \le s < 128$.

If $s \le 111$, the number of calls to *_update()* is (m+1)

If $s > 111$, the number of calls to *_update()* is (m+2)

Denote m = floor (x/64), s = y mod 64, and the cost (in CPU cycles) of one SHA-512 *_update()* function by UPDATE512. The number of cycles for computing the SHA-512 of M is approximated by

$$\text{UPDATE512} \cdot (m + 1 + \text{floor}(s/111)) \quad (2)$$

As an example, Figure 1 shows the SHA-512 flow of such a message, and it also illustrates why the overheads beyond the *_update()* invocations are negligible with respect to performance.

```
Input: a pointer to the hash string (8 *
64bit long words), a pointer to the message
whose byte length is a multiple of 128.
Output: The hash string holding the SHA-512
digest of the message.
Prototype:
void SHA-512_128byte_blocks(uint64_t
hash[8], uint8_t msg[256], int byte_length)
Flow:
SHA-512Init(hash)
last_block = zero_string
last_block[byte 0] = 0x80
last_block[qword 15] =
big_endian(byte_length*8)
append(msg, last_block)
for i=0 to byte_length/128
    SHA-512Update(hash, msg)
    msg = msg+128
end for
Output: The hash now  holds the digest of
the message
```

Figure 1: SHA-512 of a message whose length is a multiple of 128 bytes (pseudo code)

For comparison purposes we show the performance, in total cycles per block and in CPU cycles per byte, of the *_update()* functions for both SHA-256 and SHA-512, measured on the latest Intel architecture (micro-architecture codenamed "Westmere"), Xeon X5670 processor. We also show the total number of CPU cycles required for hashing a 1024 bytes message. The reported measurements were carried out on an Intel Xeon X5670 processor running at 2.67 GHz. The operating system was Linux (OpenSuse 11.1 64 bits). To isolate the performance of the functions that we measured, we disabled Intel® Turbo Boost Technology, Intel® Hyper-Threading Technology, and Enhanced Intel Speedstep® Technology. No X server and no network daemon were running. This data is shown in Tables 1 and 2.

As an example for optimized code (unrolled assembler), we used OpenSSL version 1.0.0a [3]. For "compact" code, we used home brewed straightforward C code.

Table 1: SHA-256 Performance

	Cycles	
	Total	**Per Byte**
SHA-256 Update (Compact C code)	1,863	29.11
SHA-256 Update (OpenSSL unrolled asm code)	1,166	18.22
SHA-256 of a 1024 bytes message (Compact C code)	33,757	32.97
SHA-256 of a 1024 bytes message (OpenSSL unrolled asm code)	19,769	19.30

Table 2: SHA-512 Performance

	Cycles	
	Total	**Per Byte**
SHA-512 Update (Compact C code)	2,473	19.32
SHA-512 Update (OpenSSL unrolled asm code)	1,483	11.58
SHA-512 on 1024 bytes message (Compact C code)	20,928	20.43
SHA-512 on 1024 bytes message (OpenSSL unrolled asm code)	13,392	13.07

In both examples we see that unrolling the code to carefully take advantage of the parallelism in the CPU micro-architecture, results in a performance improvement of ~37% (for SHA-256) and 37-40% (for SHA-512).

To illustrate the accuracy of the performance approximations, apply Equation (2) to a 1024 byte

message (m=8, s=0), with UPDATE512 = 1483 (Table 2, unrolled code). The approximated cycles count for SHA-512 is 13,347 cycles, which indeed closely approximates the actually measured 13,392 cycles. The small differences can be attributed to the overheads.

When comparing apples-to-apples implementations, SHA-512 performs ~50% more efficiently SHA-256. Even when comparing "best" against "worst" implementations, the SHA-512 performance is within ~6% of SHA-256 implementation.

3. The SHA-512/256 Truncation

In this section we will show how to truncate SHA-512 to 256 bits. The result of this process we refer to as SHA-512/256.

SHA-384 [2] already provides an existing example for truncation of SHA-512 to a shorter digest size. The computations of SHA-384 are exactly the same as SHA-512, and in order to signify that a hash was performed by a truncated form of SHA-512, the initial hash values are set to different constants. In other words, the only difference between SHA-512 and SHA-384 is in the *_init()* function, and of course the truncation itself.

We propose to apply the same technique to the truncation of SHA-512 to 256 bits digest.

In SHA-512 the *init()* function sets the initial state to the first 64 bits of the fractional parts of the square roots of the first 8 prime numbers. In SHA-384 these constants are replaced with the fractional parts of the square roots of the ninth through sixteenth prime numbers.

By analogy, we propose that the initialization constants for SHA-512/256 would be the fractional parts of the seventeenth through twenty-fourth prime numbers. These values are shown in Table 3. When the hashing has been completed, and a 512 bit result is obtained, the truncated digest would be defined the lower 256 bits of that result.

Table 3: SHA-512/256 Proposed Initial Constants

	Prime Number	Prime Value	The first 64bits of the fractional part of the square roots of the primes
h0	17	59	ae5f9156e7b6d99b
h1	18	61	cf6c85d39d1a1e15
h2	19	67	2f73477d6a4563ca
h3	20	71	6d1826cafd82e1ed
h4	21	73	8b43d4570a51b936
h5	22	79	e360b596dc380c3f
h6	23	83	1c456002ce13e9f8
h7	24	89	6f19633143a0af0e

Figure 2 provides a test vector example.

```
The 256 bytes message (represented as
a sequence of bytes; byte 0 is first,
byte 255 is last):
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff


The SHA-512/256 hash value (before
truncation):
h0 = 4ff7ecb3e7c23b55
h1 = 9974eba17a3d1a62
h2 = 0504f18be2e472ea
h3 = c5c5cbf75b3b7550
h4 = 27a8af7dc7dc9845
h5 = 8cfc76997dc50cfd
h6 = f4f500cc1830f561
h7 = bf2abd3732fdf66a


The SHA512/256 hash value (256 bits):
h0 = 4ff7ecb3e7c23b55
h1 = 9974eba17a3d1a62
h2 = 0504f18be2e472ea
h3 = c5c5cbf75b3b7550


For comparison, the SHA-512 hash value
of the same message is:
h0 = 1e7b80bc8edc552c
h1 = 8feeb2780e111477
h2 = e5bc70465fac1a77
h3 = b29b35980c3f0ce4
h4 = a036a6c946203682
h5 = 4bd56801e62af7e9
h6 = feba5c22ed8a5af8
h7 = 77bf7de117dcac6d
(note that completely different values
are obtained)
```

Figure 2: A SHA-512/256 example.

4. Calculating the SHA-512 constants with a smaller lookup table

The downside of implementing SHA-512 is that it requires a table of eighty 64 bit constants (a 640 bytes lookup table). For comparison, SHA-256 requires only sixty four 32 bit constant (a 256 bytes lookup table).

In some implementations the cost of storing data for the lookup table can be exceptionally high. In such cases, the storage requirement of SHA-512, namely for 384 more bytes than SHA-256, would be considered as a disadvantage.

For SHA-256, there is a way to reduce the storage requirement by computing the sixty four constants which are defined to be the first 32 bits of the fractional parts of the cube roots of the first sixty four prime numbers. One way to compute these cube roots is to use Newton-Raphson iterations, which, on 64 bit architectures, quickly converge to provide the first 32 bits of the result.

Unfortunately, this is not the case for SHA-512, because the SHA-512 constants are defined to be the first 64 bits of the fractional part of the cube roots of the first eighty primes. Performing simple numerical iterations on 64 bit architecture does not give the required precision.

We propose the following method for obtaining the SHA-512 constants. They can be approximated, using the Newton-Raphson Algorithm, up to the last two bytes (in no more than 14 iterations of the algorithm). Therefore, it is sufficient to store, for each constant, only the two bytes (pre-computed) difference between the result of the Newton-Raphson iterations and the exact constant. To avoid storing the first eighty primes, it is enough to store only the difference from the previous prime number. For example, the difference from the first prime is (2) is 0, the difference of the next prime (3) from the previous one (2) is 1, the next difference is 5-3=2 and so on. Since up to the first eighty primes, the largest difference is 24, it follows that all differences can be represented by 4 bits, so that pairs of differences can be stored in a single byte.

Altogether, this method uses only 2.5 bytes for each constant (instead of 8 bytes if the constants are stored), therefore, reducing the table size from 640 bytes to only 200 bytes. This method trades a reduced table size with the small cost of additional code and computations. The implementation and the associated constants are detailed in Figure 3 below.

```
Input: Pointers to three arrays.
Array 1 holds the differences between the first 80 primes (two differences per byte).
Array 2 holds the difference between the result of the Newton-Raphson iterations and the desired constant.
Array 3 a place to store the computed 80 SHA-512 constants.
Output: The SHA-512 constants
Prototype:
void calculate_SHA-512_constants (uint16_t deltas[80], uint8_t offsets[40], uint64_t K[80])
Constants:
offsets =
0x01, 0x22, 0x42, 0x42, 0x46, 0x26, 0x42,
0x46, 0x62, 0x64, 0x26, 0x46, 0x84, 0x24,
0x24, 0xe4, 0x62, 0xa2, 0x66, 0x46, 0x62,
0xa2, 0x42, 0xcc, 0x42, 0x46, 0x2a, 0x66,
0x62, 0x64, 0x2a, 0xe4, 0x24, 0xe6, 0xa2,
0x46, 0x86, 0x64, 0x68, 0x48

Deltas =
0xfe22, 0x05cd, 0xfb2f, 0xfbbc, 0xf538,
0xf019, 0xef9b, 0x0118, 0x0242, 0x0fbe,
0xf28c, 0xf4e2, 0x096f, 0xf6b1, 0xf235,
0x0694, 0x0ad2, 0x05e3, 0x15b5, 0x1c65,
0x0275, 0xe483, 0xfbd4, 0x13b5, 0xdfab,
0xf210, 0xe13f, 0x0ee4, 0x0fc2, 0xe725,
0x026f, 0xee70, 0xeffc, 0x0926, 0xeaed,
0xf3df, 0xe3de, 0xf2a8, 0xeee6, 0xf53b,
0x0364, 0xf001, 0x1791, 0xfe30, 0x1218,
0x2910, 0xe02a, 0xd1b8, 0x10c8, 0xeb53,
0xeb99, 0x08a8, 0x1a63, 0x0acb, 0xe373,
0xf8a3, 0xf2fc, 0xef60, 0x2b72, 0xf9ec,
0x1e28, 0xfde9, 0xf915, 0x132b, 0xe19c,
0x0207, 0xeb1e, 0x1178, 0xefba, 0x18a6,
0x0dae, 0x071b, 0xfd84, 0x2493, 0xfebc,
0x0d4c, 0x02b6, 0xfe2a, 0xfaec, 0x1817
Flow:
double p = 2   //the first prime
for i=0 to 79
    if (i%2 = 1)
        //offset is in the second nibble
        p = p + (offsets[i/2] & 0x0f)
    else
        //offset in the first nibble
        p = p + (offsets[i/2] >> 4)
    end if
    double n = p/3
    for j=0 to 13
        n = n - (n^3-p)/3n^2
    // correction to 64 bits accuracy
    K[i] = fraction(n) * 2^64 + deltas[i]
end for
Output:
The constants are now in the K array
```

Figure 3: Computing the SHA-512 constants.

5. SHA-512/t – Truncation to Other Output Lengths

It is conceivable that supporting digests of other lengths, less than 512, would be useful. A straightforward way to standardize such truncations would be to use another distinct set of eight primes, as SHA-384 and the proposed SHA-512/256 do. Such an approach does not scale gracefully because the initialization constants are not naturally related to the desired digest length. To avoid this situation we suggest another truncation method.

Let t be an integer satisfying $0 < t < 512$, $t \neq 256$, $t \neq 384$. Encode l as a Big-Endian 1024 bit integer T.

For the truncation of SHA-512 to t bits, namely SHA-512/t, we define the initial state (initialization constants) as follows:

$$IV_{512/t} = \text{SHA-512 (T)} \qquad (3)$$

In other words, $IV_{512/t}$ is the output of the SHA-512 compression function (C_{512} hereafter) operating on the encoded number T.

As above, SHA-512/t would use the initialization constants as in Equation (3), with all the remaining computations remaining the same as SHA-the standard 512. When the hashing has been completed, and a 512 bit result is obtained, the truncated digest would be defined the lower t bits of that result.

The initialization constants for SHA512/t for t=256 are shown in Table 4, and a test vector example is provided in Figure 4.

Table 4: SHA-512/t (t=256) Initial Constants

Length	256
Length T, encoded as a 1024 bit (Big-Endian) number. Here, T=256. (byte 0 is first, byte 127 is the last)	000100000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 000000000000000000000000 0000000000000000
The initial constants $IV_{512/t}$ (t=256) (= SHA-512 (T))	h0 = 2b2b0a74439fba29 h1 = b0395e75cf517538 h2 = 2d56e63211d68a9a h3 = cd2e4f0e7f903a4b h4 = 1fa53c41cf466fe4 h5 = 60119e4c4bc5e6c6 h6 = b895a38bba334ca3 h7 = 68b7beb95a22e694

```
The 256 bytes message (represented as
a sequence of bytes; byte 0 is first,
byte 255 is last):
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff

The SHA-512/256 hash value (before
truncation):
h0 = 50a201a0449a0617
h1 = 43e22f5c48eff125
h2 = 3ef8380002ae5655
h3 = 2b8c57b60cce2f2e
h4 = 4e2280f2ad1c4d8a
h5 = 688eb8b073694f88
h6 = ad4d5b4cbe93c8f4
h7 = 442d3a450787b415

The SHA512/256 hash value (256 bits):
h0 = 50a201a0449a0617
h1 = 43e22f5c48eff125
h2 = 3ef8380002ae5655
h3 = 2b8c57b60cce2f2e

SHA-512 of the same message:
h0 = 1e7b80bc8edc552c
h1 = 8feeb2780e111477
h2 = e5bc70465fac1a77
h3 = b29b35980c3f0ce4
h4 = a036a6c946203682
h5 = 4bd56801e62af7e9
h6 = feba5c22ed8a5af8
h7 = 77bf7de117dcac6d
(different values are obtained)
```

Figure 4: A SHA-512/t for t=256; Example.

This construction is different the method used by NIST for the SHA-384 truncation (and therefore different from the truncation we proposed in Section 3). On the other hand, this construction enjoys the following properties:

1. SHA-512/t (M) = SHA-512 (T || M), where T is as above, "||" denotes string concatenation, and M is any bit string (message) whose length in bits does not exceed $2^{64} - 1024$.
2. In particular, all of the values $IV_{512\text{-}t}$ are distinct if C_{512} is collision resistant.
3. If t and t′ are two distinct positive integers less than 512, then SHA-512/t (M) ≠ SHA-512/t′ (M) for any message M, since by SHA-512's collision resistant they are unequal before truncation.
4. SHA-512/t is collision resistant, pre-image resistant, and second pre-image resistant if SHA-512 also has these properties.

6. Conclusion

From our performance analysis, and experimentation on 64 bit Intel Architecture, we have shown that the cost of implementing a SHA-512 algorithm delivers a 50% performance improvement over similar implementations of SHA-256. We also showed that the storage costs for implementing SHA-512 can be reduced by adding a small amount of one-off computation to compute the SHA-512 constants - which we believe will be useful for constrained implementation environment.

In order for users to be able to distinguish between a SHA-512 digest which has been truncated and a SHA512/256 digest, we also offer new initialization constants, analogous to those used in SHA-384. We also follow the standard truncation method in the SHA standard [2] which can be extended to truncations to other lengths by choosing the next set of 8 primes.

When the NSA designed the SHA family of algorithms their design rationale was never published (this is one of the two major motivations for the SHA-3 competition; the other one being Wang's attack on SHA-1). To the best of our knowledge, from observing the SHA standard, and in particular, the method used for defining SHA-384, the actual values of the initialization constants are immaterial. They only need to be unique per hash function, and this is what we used for our SHA-512/256 truncation.

So, in addition, we propose an alternative method to define SHA-512/t, a truncation of SHA-512 to t bits long digests (for any positive t not equal to 256 or 384). This construction is very similar to the mechanism used in the Skein proposal for SHA-3 [4].

In either case, given SHA-512's performance on 64 bit architectures, we believe that SHA-512/256 removes a performance obstacle for the adoption of wider hash values, and that a truncated version of SHA-512 to 256 bits is a viable alternative to SHA-256, for 64 bit architectures.

6. References

[1] NIST, "NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and Continued Security Provided by SHA-1", 25th August 2004, http://csrc.nist.gov/groups/ST/toolkit/documents/shs/hash_standards_comments.pdf

[2] Federal Information Processing Standards Publication 180-3, "SECURE HASH STANDARD", October 2008, http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[3] OpenSSL Source Code, http://www.openssl.org/source

[4] N. Ferguson *et al*, "The Skein Hash Function Family", Version 1.3, 1st October 2010, http://www.schneier.com/skein1.3.pdf