SAND93-2591

Unlimited Release
First Printed October 1992
Revised October 29, 1993
Revised June 22, 1994

Distribution
Category UC-705

A Fast Portable Implementation of the Secure Hash Algorithm, III*†

Kevin McCurley
Organization 1423
Sandia National Laboratories
Albuquerque, NM 87185.

July 11, 1994

1 Introduction

In 1992, NIST announced a proposed standard for a collision-free hash function. The algorithm for producing the hash value is known as the Secure Hash Algorithm (SHA), and the standard using the algorithm in known as the Secure Hash Standard (SHS). Later, an announcement was made that a scientist at NSA had discovered a weakness in the original algorithm. A revision to this standard was then announced as FIPS 180-1, and includes a slight change to the algorithm that eliminates the weakness. This new algorithm is called SHA-1. In this report we describe a portable and efficient implementation of SHA-1 in the C language. Performance information is given, as well as tips for porting the code to other architectures. We conclude with some observations on the efficiency of the algorithm, and a discussion of how the efficiency of SHA might be improved.

2 Background

The concept of a one-way function was first described in connection with storing passwords for computer logins [7, p. 91]. Since then the importance of one-way functions has grown to include other topics. In particular, in order for digital signature schemes to gain widespread acceptance, much more efficient one-way functions are required.

The concept of a one-way function has appeared in various forms, and is known under several names. They are related to the notion of a checksum, but not exactly the same. The purpose of such a checksum is to provide a short string of bits that gives some assurance of the accuracy of some much larger message. One property that such a checksum lacks is protection against deliberate tampering. By this I mean that it may be relatively simple to find another message that gives the same check sum. If a functions produces checksums that are hard to invert, then they are commonly referred to as collision-free hash functions. To be more precise, a collision-free hash function f is one with the property that it is infeasible to produce two messages $m_1 \neq m_2$ such that $f(m_1) = f(m_2)$. These are sometimes also known as Manipulation Detection Codes (MDC), a message digest, or fingerprints. By contrast with Message Authentication Codes, there is no secret information used in their construction.

*This work was performed under U.S. Department of Energy contract number DE-AC04-76DP00789.
†This report has been revised multiple times as more improvements have been found.

One early proposal for such a message digest was based on use of the Data Encryption Standard [3]. One serious disadvantage of this method involves the exportability of devices or software for performing encryption. Because of the use of DES, such a method was not freely exportable. In recent years there have been several proposals for collision-free hash functions that are based on methods other than encryption algorithms, with the goal of making them exportable. Included in this list are MD4 [5], MD5 (a later refinement of MD4 that is slower but presumably more secure), and Snefru [2]. The algorithm description for such functions is exportable under the GTDA license, but software for such hash functions still requires a license from the Department of Commerce For the code described here, it falls under the Export Control Commodity Number 5D95G, and is exportable under this license to everywhere except countries in category S and Z, along with a few others. There are additional restrictions if the exporter knows that it will be used for nuclear, chemical, or biological weapons. This information was provided to me by the Bureau of Export Administration, Department of Commerce (202) 377–0706.

In 1991, the National Institute of Standards and Technology (NIST) proposed a draft standard [6] for such a one-way function, known as the Secure Hash Standard (SHS). This standard later became FIPS 180. Since the adoption of this standard, a weakness was discovered by a scientist at NSA, and a fix for this weakness was announced in the form of a slightly modified algorithm called SHA-1 (the standard was modified to become FIPS 180-1). The Secure Hash Standard employs an algorithm known as the Secure Hash Algorithm (originally SHA, now SHA-1). A complete description of SHA-1 is beyond the scope of this document, but interested readers may contact NIST for the complete description [6]. Roughly speaking, the SHA-1 function takes an input consisting of a sequence of bits, and produces a 160-bit output (notice that 160 bits is 20 bytes, or 5 32-bit words).

One disadvantage of collision-free hash functions such as MD4, MD5, Snefru, and SHA-1 is that they are rather computationally demanding. Unfortunately, this restricts their widespread use, and it is therefore desirable to implement them as efficiently as possible. At the same time, the plethora of new computer architectures appearing on the scene suggests that a portable implementation would be very useful. Experience has shown that the very highest performance for a given computer can only be achieved by using assembly language to exploit the particular instruction set of the target architecture. On the other hand, the code described here can provide an excellent platform upon which to build a highly tuned implementation. Profiling information gathered on a Sun Sparcstation shows that approximately 88% of the time in a benchmark test is spent in a single routine (`do_block()`) that processes a 512-bit block. One attractive approach to tuning the code would be to use the C version to generate assembler code, and then tune the assembler to minimize the number of instructions required.

The operations that are used in SHA-1 are specifically designed with 32-bit microprocessors in mind, and consist of mainly of the following:

- addition modulo 2^{32},
- circular shifts of 32-bit quantities,
- exclusive or's of 32-bit quantities,
- logical and's of 32-bit quantities,
- logical complementation of 32-bit quantities.

The operations are chained together in such a way as to make it nonobvious how a person might invert the operations, thus making composed from these operations collision-free. These operations are all available from most 32-bit processors, and all but the circular shift are easy to access through the common computer language C (provided a 32-bit unsigned integer type is available). Unlike the design of MD4, SHA-1 favors big-endian architectures such as the Sparc standard rather than the Intel 80x86 processor family.

It should be pointed out that the code described here is *not* a certified implementation of SHA-1. The standard provides for the hash function to be applied to bit strings, whereas this implementation assumes that the message consists of a sequence of 8-bit bytes. This should not be a serious limitation in practice,

but nonetheless it does not support the full standard. Furthermore, the standard allows the length of the message to be a 64-bit integer, whereas we only accept strings of up to 2^{31} bytes (processing of the longest such string would take approximately 35 minutes on a Sparcstation II anyway).

3 Machines Tested

An earlier version of this paper reported on testing with a larger number of machines, but for the newer SHA-1 we must content ourselves with testing on a smaller number of machines than previously. In particular, all of the machines used in the testing run variations of UNIX, because these are the only machines available to the author. The machines used so far include models from DEC, Sun, Silicon Graphics, HP, NeXT, and Gateway. The necessary requirements are:

- support by the C compiler for a 32-bit unsigned integer data type,
- access to file size information through the equivalent of the `stat()` function of the UNIX operating system,
- for testing purposes, software access to a hardware clock for measuring performance.
- support for the `memcpy()` function (or a suitable replacement such as `bcopy()`).

The current implementation consists of a command line interface, and accepts several arguments. These arguments can be used to specify a filename, indicate to hash from standard input `stdin`, run a benchmark, run an example provided by NIST, or report timing information. Also included in the source code are separate functions to compute the hash value of a message provided in the form of a filename or a string (sequence of 8-bit bytes), as well as a separate function that reads from `stdin`. Prototypes for these functions are as follows:

```
uint32 shsstring(uchar8 *message, uint32 m_len, uint32 h[],uint32 options)
uint32 shsfile(char *filename, uint32 h[], uint32 options)
uint32 shsfilter(uint32 h[], uint32 options)
```

The standard distribution consists of the following files:

README	a short description
Makefile	for ease of compilation
main.c	a driver program
shsstring.c	hash a string
shsfile.c	hash a file
shsfilter.c	hash the input from stdin
do_block.c	internal routine to process a 512-bit block.
byteswap.c	for little-endian machines
shift.h	define circular shift operation
bs.h	for little-endian machines
timer.c	a timer

In addition, the following files are included for those machines that conform to various UNIX (non)standards:

timer1.c	another timer (POSIX standard)
timer2.c	yet another timer using clock()
getopt.c	option parser routine
junk.c	generate a benchmark file.

4 Performance Measurements

In this section we report the observed timings from the implementation, performed on a variety of machines using a variety of compilers. All of the timings were performed using an internal software timer provided as part of the standard library of a C compiler. In each case, the timings were from a hash of a message of approximately 2 megabytes. Since the algorithm speed does not depend in any observable way upon the actual sequence of bits in the message, it can be assumed that the results are representative of what should be observed for a random message.

In each case, different options were tried to determine the combination of compiler options that would produce the best timings. Timings are given below for hashing a string from memory and reading a file from disk. The latter can be considered somewhat unreliable, since they depend on the exact choice of disk system that is attached. Moreover, it might be possible to tune the code to a particular machine somewhat by changing the size of the default buffer that is read from 64 bytes to the natural size of a sector on the attached disk drive.

Machine	Operating system	compiler & flags	bytes/second from memory	bytes/sec from file[a]
SGI Onyx (R4400)	Ultrix 5.2	cc -O2	4,258,600	2,706,175
Sun Sparcserver 1000	Solaris 2.3	gcc 2.3.3 -O2	3,575,073	2,333,818
DEC Alpha 3000/400	OSF 2.0	gcc 2.5.8 -O2	3,039,210	2,383,684
DEC Alpha 3000/400	OSF 2.0	cc -O4[b]	[c] 2,845,988	2,184,393
Sun Sparcstation 10	SunOS 4.1.3_U1	gcc -O2	2,438,549	1,990,561
Sun Sparcstation 10	SunOS 4.1.3_U1	cc -O4	2,097,152	1,797,926
Gateway 2000 Pentium 66Mhz[d]	Linux 1.0	gcc -O2	2,207,528	-
Sun Sparcstation LX	SunOS 4.1.3	gcc -O2	657,414	-
Sun Sparcserver 1000	Solaris 2.3	none	[e] 0	0
Sparcstation 2	SunOS 4.1.3_U1	gcc 2.3.3 -O2	[f] 1,075,463	830,106

[a] The timings from a file are highly dependent on the speed of the filesystem. In all cases for UNIX systems, the file systems were local rather than NFS mounted. Input from /bin/cat typically runs faster because of better buffering of reads.

[b] This option requires compiling on a single line, without using the makefile.

[c] This used `BYTESWAP_MACRO`. The byteswap function slows it down tremendously due to inefficiencies in byte loads and stores. If it wasn't a little-endian processor, then it would run at 3.3 Megabytes/second.

[d] This is a little endian machine. If the hardware rotate instruction is used (which is possible within gcc) then it should run about 40% faster.

[e] Performance measured from installed compiler. If Sun wants me to buy another machine, they can start shipping machines with bundled compilers again.

[f] The standard compiler produced a value of 1,003,422 bytes per second.

For comparison, the RSA implementation of MD5 that was written by Dusse and Rivest, ran at about 1,115,000 characters per second on my Sparc II, or about 11% faster than this implementation of SHS. (This code is available by anonymous ftp to rsa.com, as file `/pub/md5.doc`). The two programs were in a virtual dead heat on the DECStation 3100. Thus it appears that in spite of the somewhat more complicated operations used by SHA-1, the degradation in performance is only very slight.

5 Comments on performance

The standard states that computations are to be done in a certain order, but clearly we are free to rearrange their order and rewrite them so long as the resulting answer is unchanged. The code employs several such tricks, and some are described in this section.

Unrolling of loops Rather than using

```
for (t=16;t<80;t++)  {
  W[t] = W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16];
}
```

I wrote out the code explicitly for each of the loop operations. This makes ugly code, but eliminates loop overhead and index calculation.

Keep W[] in registers Rather than perform the entire computation of the `W` array, we computed the values as we need them, so that they can still be in registers after they are computed. If there is a processor with a lot of registers (about 40), then the useful part of the `W` array could be held in registers, preventing the processor from having to write them out to memory at all. Then the bus would be used only for feeding the data to the registers, and nothing else.

Redefine the registers The standard says that at each step you should shift the registers `A`,`B`,`C`,`D`, and `E`:

```
E = D;  D = C;  C = S(30,B);  B = A; A = TEMP;
```

Instead, we simply changed their meanings throughout the code.

Rewrite the mixing functions For $0 \leq t \leq 19$, the standard says to write `f(t,x,y,z)` as

```
f(t,x,y,z) = (x AND y) OR (~x AND z)
```

Instead we can rewrite this as the mathematically equivalent expression

```
f(t,x,y,z) = (z XOR (x AND (y XOR z)))              (0  <= t <= 19)
```

The latter uses three bitwise operations rather than four. Note also that the DEC Alpha 21064 microprocessor has instructions combining logical complementation with other logical operations (e.g., `A AND (NOT B)`). This can be used as an alternative, although it would require assembly language to do so.

Similarly, the standard says to use

```
f(t,x,y,z) = (x AND y) OR (x AND z) OR (y AND z)    (40 <= t <= 59)
```

but this can be rewritten as

```
f(t,x,y,z) = (x AND y) OR (z AND (x OR y))          (40 <= t <= 59)
```

The latter uses four operations rather than five.

It was observed in [4] that the latter may also be rewritten as

```
f(t,x,y,z) = x AND (y XOR z) XOR (z AND y)          (40 <= t <= 59)
```

They remark that this is faster for simple processors containing only an accumulator rather than general-purpose registers.

Hardware operation for circular shift Another speedup can be obtained when the hardware has a 32-bit circular shift (rotate) instruction. This exists on the 80x86 and 68000 series of processors, and may exist on others. This would allow you to replace the macro

```
#define S(n,x) (((x)<<(n)) | ((x)>>(32-(n))))
```

by a single inline assembly instruction. As an example of the type of savings that can be found in assembler, consider the code generated by the Borland C compiler for the 80x86 (instruction timings for a 486 are in the right column):

```
;
; B = S(30,B);
;
     mov     eax,edi   ; 1 clock
     shl     eax,30   ; 2 clocks
     mov     edx,edi   ; 1 clock
     shr     edx,2    ; 2 clocks
     or      eax,edx   ; 1 clock
     mov     edi,eax   ; 1 clock
```

for a total of 8 clock cycles, but since B is not reused, this could immediately be replaced with

```
     mov     eax,edi   ; 1 clock
     shl     eax,30   ; 2 clocks
     shr     edi,2    ; 2 clocks
     or      edi,eax   ; 1 clocks
```

for a total 6 clock cycles, and it could further be improved to

```
     rol     edi,30   ; 2 clocks
```

to bring it down to 2 clock cycles.

Register allocation It appears that except for the `W[]` array, all of the data in `do_block()` can be held in six registers. In order to avoid writing the `W[i]` values back to memory, it appears that we would require another 16 different registers, for a total of 22 registers capable of holding 32-bit integers. On a machine such as the 80x86 architecture, there are far too few registers for an efficient implementation (their use is also generally restricted). Other machines can benefit from efficient register management.

6 Comments on portability

Endian-ness As was mentioned before, SHA-1 favors big-endian machines, since little-endian machines need to swap the bytes inside 32-bit integers before processing. This can be done in one of two ways, depending on whether `BYTESWAP_MACRO` is defined in `shs.h`. If `BYTESWAP_MACRO` is defined, then it includes a macro from `bs.h` to swap bytes. Otherwise it uses a function `byteswap()`. Whichever is faster depends on the instruction set for the machine (for example, byte operations on the DEC alpha are very slow). In tests using djgpp, I found that it ran approximately 24% faster without the `byteswap` routine (of course this gave bogus hash values on the 80386). It would be faster to rewrite the entire code so that they `byteswap()` calls occured when the data was in a register rather than in memory, but unfortunately this would require a major restructuring since it does not reside in `do_block()`.

Portability of timing method Several versions of timing function is provided, since there appears to be no portable and reliable way to get running times of a process within the ANSI C standard. Care needs to be taken when timing programs on multi-user systems, so that the time used by the process is measured rather than the wall clock time.

Irritating details When I tried compiling this code with the standard `cc` compiler supplied with Ultrix on a Decstation model 3100, it compiled without complaining but produced incorrect answers. I managed to isolate the bug to the `byteswap()` routine, involving the value of the argument that is passed to it, which was changed by the call. I was unable to diagnose this problem, but I suspect a compiler bug involving the stack pointer. The code compiled correctly on the machine using the Gnu compiler (version 2.0).

The Sequent was missing a few things like `memcpy()` and `getopt()`. I also renamed `string.h` to `strings.h`.

The `stat()` function supplied with the libraries for Turbo C and djgpp libraries did not seem to work in the same fashion as UNIX machines. In particular, the file size returned by `stat` was different from that shown by the DOS command `dir`. This seems to be an artifact of the DOS file system that is machine-specific, and I elected to ignore it. You might need to seek to the end of the file to find the actual size.

A few other changes were required to get it to compile with Turbo C:

- a change of `memory.h` to `mem.h` in `shsstring.c`.
- replacement of `timer.c` by `timer2.c`, with a change of 1000000 to `CLK_TCK`.
- replacement of `sys/time.h` by `time.h`.

How much can we improve SHA?

The code described in this paper comes close to what I think is the fastest possible portable microprocessor implementation of SHA-1. For some applications, this may be sufficiently fast, but for others it may be far off of the mark. The world is moving toward ubiquitous electronic communication, and much of it is going to require authentication and digital signatures. For the gigabit per second networks that exist already, SHA-1 is going to be of limited use, and this situation is likely to become more pronounced as technology advances. Networking speed can be improved by using multiple paths, but the SHA is inherently serial.

One may ask how close to optimal this implementation is. Examination of the assembler code produced for the `do_block()` function on a variety of machines shows that processing of a 512-bit block generally requres something like 1600 individual instructions with this C implementation. If we examine the algorithm, it appears that the following 32-bit operations are required at a minimum:

- 3 instructions for each invocation of `f1`.
- 2 instructions for each invocation of `f2` (and for `f4`).
- 4 instructions for each invocation of `f3`.
- 1 load instruction for each 32-bit block of input data.

This leads to the following estimate for the number of 32-bit instructions required for the original SHA algorithm:

Round 1: 40 circular shifts, 80 32-bit adds, and 60 operations for `f1`,

Round 2: 40 circular shifts, 80 32-bit adds, and 40 instructions for `f2`,

Round 3: 40 circular shifts, 80 32-bit adds, and 80 instructions for `f3`,

Round 4: 40 circular shifts, 80 32-bit adds, and 40 instructions for `f4`,

`W[i]`**, all rounds** 192 32-bit xor instructions,

loads 16 32-bit loads,

for a total of 908 32-bit operations. This ignores a number of factors:

- extra load/stores from having too few registers.
- overhead of a function call (stack setup, etc)
- index and/or address calculation for the `W[]` array.

Based on these observations, it appears to be unlikely to achieve a speedup of more than 40% over the version written here using a 32-bit microprocessor, unless the processor has multiple integer pipelines.

On a Sparcstation, the compiler turns each 32-bit circular shift operation into three of four instructions rather than one. Since the Sparc chip is a RISC chip, each instruction takes a single clock cycle, and the 908 operations counted above would become $908+4\times160 = 1548$ operations. This is where the biggest inefficiency arises. Note also that SHA-1 adds 64 extra circular shifts, which inflates this to $1548 + 64 \times 4 = 1804$ operations. Based on this, we would expect a performance hit of approximately 16% from the SHA-1 modification, and we observed a hit of 14%. The difference is probably accounted for in moving data around. The SHA-1 modification makes it that much more important to use the hardware cyclic shift operation if at all possible.

This paper does not address the security of SHA-1, but only it's efficiency. Clearly an insecure but secure hash function is useless, but it is an interesting open question to design secure functions that operate efficiently on common microprocessors. Design of such a function should ideally observe the following contraints:

- For each "block" of input data, it should only be loaded moved once from memory to a register.
- The algorithm should scale to multiple processors operating on individual register sets and a shared communication medium. This communication medium may have low latency such as a shared memory bus, or some kind of network. The network may be assumed to have high bandwidth, but may also have relatively high latency (i.e., tens of microseconds, or even milliseconds).
- The algorithm should rely on as few operations as possible, and should avoid such unusual operations as a circular shift that may be missing from RISC processors. This may be unavoidable for security reasons.
- The algorithm should be be capable of operating on a $2n$ bit data type with only slightly more than half the number of operations required when using an n-bit data type.
- Chaining of data is generally used to enhance security, but this is at odds with processor pipeline design. This should be held to a minimum to take advantage of pipelines.

It is interesting to note that the operation used in SHA-1 are exactly of the type that are extremely difficult to vectorize, namely recurrences with short lags. Pipelining of instructions should easily be possible however. It remains an open question whether a secure hash function can be designed that lends itself to either vectorization or parallelization. Very high speed applications in the future will no doubt require this, and perhaps someday we will regret not having planned for this.

Contact point:

This code is provided without any explicit support, but the author would be happy to respond to questions related to portability or performance issues, time permitting. I can be reached at:

Kevin S. McCurley
Organization 1423
Sandia National Laboratories
Albuquerque, NM 87185
email: `mccurley@cs.sandia.gov`
phone: (505) 845–7378
fax: (505) 845-7442

References

[1] Kevin S. McCurley, "A Fast Portable Implementation of the Secure Hash Algorithm", Sandia National Laboratories Technical Report SAND92-1634. This report is no longer available.

[2] Ralph Merkle, "A Fast Software One-Way Hash Function," Journal of Cryptology **3** (1990), 43–58.

[3] Ralph Merkle, "One Way Hash Functions and DES,", Advances in Cryptology - Crypto '89, Lecture Notes in Computer Science, Volume 435, Springer-Verlag, New York, 1990, pp. 428–446.

[4] David Naccache, David M'Raïhi, Dan Raphaeli, and Serge Vaudenay, "Can D.S.A. be Improved? - Complexity Trade-offs with the Digital Signature Standard", Proceedings of Eurocrypt '94, Springer-Verlag, to appear.

[5] Ronald L. Rivest, "The MD4 Message Digest Algorithm," Advances in Cryptology - Crypto '90, Lecture Notes in Computer Science, Volume 537, Springer-Verlag, New York, 1991, pp. 304–311.

[6] Request for comments on a proposed Secure Hash Standard (SHS), Federal Register, Volume 57, No. 21, January 31, 1992, pp. 3707–3708. (Copies of the proposed standard are obtained by writing to: Standards Processing Coordinator (ADP), National Institute of Standards and Technology, Gaithersburg, MD 20899).

[7] M. V. Wilkes, *Time-sharing computer systems*, American Elsevier, New York, 1968.