Performance Analysis of MD5

Joseph D. Touch[1]
USC / Information Sciences Institute
(touch@isi.edu)

Abstract

MD5 is an authentication algorithm proposed as the required implementation of the authentication option in IPv6. This paper presents an analysis of the speed at which MD5 can be implemented in software and hardware, and discusses whether its use interferes with high bandwidth networking. The analysis indicates that MD5 software currently runs at 85 Mbps on a 190 Mhz RISC architecture, a rate that cannot be improved more than 20-40%. Because MD5 processes the entire body of a packet, this data rate is insufficient for current high bandwidth networks, including HiPPI and FiberChannel. Further analysis indicates that a 300 Mhz custom VLSI CMOS hardware implementation of MD5 may run as fast as 256 Mbps. The hardware rate cannot support existing IPv4 data rates on high bandwidth links (800 Mbps HiPPI). The use of MD5 as the default required authentication algorithm in IPv6 should therefore be reconsidered, and an alternative should be proposed. This paper includes a brief description of the properties of such an alternative, including a sample alternate hash algorithm.

1: Introduction

The current Internet Protocol (IP) is undergoing its first major revision in 14 years [24]. As part of that revision, the new IP (IPv6, [12]) proposes a number of required options that were not required in the previous IP (IPv4, [24]). This paper describes a performance analysis of MD5 [28], the proposed "required optional" authentication algorithm in IPv6 [1]. Analysis indicates that MD5 may not adhere to the performance criterion of IPv6 [23], and thus its mandate as the default for a required option in IPv6 should be reconsidered.

This paper is organized as follows:

- An overview of MD5 and its relevance to IPv6.
- Measurements of MD5's reference implementation.
- Manual optimizations to reference implementation.
- Analysis of the limits of a software MD5.
- Analysis of the limits of a hardware MD5.
- Recommendations for further work, and pending proposals.

1. This work is supported by the Advanced Research Projects Agency through Ft. Huachuca contract #DABT63-93-C-0062 entitled "Netstation Architecture and Advanced Atomic Network". The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Army, the Advanced Research Projects Agency, or the U.S. Government.

1.1: IPv6 performance criterion

The technical criteria for IPv6 (also known as IPng) includes an explicit performance criterion [23]:

> *A state of the art, commercial grade router must be able to process and forward IPng traffic at speeds capable of fully utilizing common, commercially available, high-speed media at the time. Furthermore, at a minimum, a host must be able to achieve data transfer rates with IPng comparable to rates achieved with IPv4, using similar levels of host resources.*

This criterion can be summarized as "first do no harm[2]". This criterion is specified in IPv6 without condition; IP header options, whether voluntary or required, are not excepted.

Several *voluntary options* in IPv4 have become so-called *required options* in IPv6, notably authentication. An *option* is a mechanism that can be enabled or disabled; *voluntary options* need not be implemented to conform to the standard, but *required options* must be implemented, but can still be enabled or disabled on individual packets. Authentication is a *required option* in IPv6.

The authentication option allows per-packet specification of the particular authentication algorithm [1]. In keeping with the "required option" spirit, one algorithm is required to be implemented. The IPv6 authentication mechanism proposes MD5 as its required algorithm.

Internet RFCs and Internet Drafts regarding authentication do not address performance issues (as a rule). Notable exceptions are the ESP encapsulation mechanism [19], a keyed MD5 [20], and the technical criteria of IPv6 [23]. The claim in other documents is that current implementations exhibit poor performance, but that software optimizations or custom hardware will overcome this limitation [10] [11] [28].

MD5 can be implemented in software on a 190 Mhz RISC processor at 85 Mbps. On a Sun SPARC 20/71, MD5 executes at 57-59 Mbps, but IPv4 executes at 120 Mbps (via Fore SBA-200 IP over ATM). MD5 can also be implemented in 300 Mhz VLSI CMOS at up to 256 Mbps. By contrast, the Internet Checksum of IPv4 can be implemented in software at very high speeds (260 Mbps on a Sun SPARC 20/61, vs. 38 Mbps for MD5), and has been implemented at 1.23 Gbps in a single inexpensive programmable chip (32 bit parallel at 26 ns in a $40 programmable logic device chip [31]). MD5 does not keep pace with available link rates (800 Mbps HiPPI), or available IPv4 implementations, in violation of the performance criterion of IPv6.

1.2: MD5

MD5 is a message digest authentication algorithm developed by RSA, Inc. [28]. It is an augmented version of the MD4 algo-

2. This phrase commonly refers to the treatment of a patient by a medical doctor, as part of the Hippocratic Oath.

rithm [27]. The authentication algorithm computes a digest of the entire data of the message, used for authentication. Typically, the message digest is registered with a trusted third-party, or encrypted via other means [20]. The digest is used by the receiver to verify the contents of a message. It can also be used to encrypt the contents of a message, via a second pass over the data by another algorithm. MD5 requires that both the sender and receiver compute the digest of the entire body of a message.

MD5 is used for authentication in a number of protocols. It is also included as an encapsulation mechanism in SIPP, IPv6, and IPv4 [19]. The following is a partial list of protocols or protocol options using MD5. Some protocols on this list chose MD5 explicitly because of its use in SNMP V2, thus its implementation in many routers (indicated by a star '*'):

- SNMP V2 [10]
- IPv6 * / IPng* [1] [12]
- IPng ESH* (uses DES) [3]
- IPv4 [19] [20]
- SIPP* (progenitor of IPng) [7]
- OSPF* [4]
- RIP-II* [5]
- RIPng* [17]
- TCP [11]
- SOCKS V5 [16]
- WWW's Secure HyperText Transfer Protocol [25]
- WWW's SimpleMD5 [13]

There is some concern that the performance of MD5 may not keep up with these other protocols. SNMP V2 is a control protocol, not intended for high bandwidth continuous stream operation, so protocol performance is not critical. Even so, SNMP V2 uses several authentication mechanisms to optimize security vs. performance, including one (DES-CBC) noted for its implementation at 1 Gbps in hardware (as indicated in [19]).

1.2.1 Overview of the MD5 algorithm

MD5 is a block-chained digest algorithm, computed over the data in phases of 512-byte blocks organized as little-endian 32-bit words (Figure 1). The first block is processed with an initial seed, resulting in a digest that becomes the seed for the next block. When the last block is computed, its digest is the digest for the entire stream. This chained seeding prohibits parallel processing of the blocks.

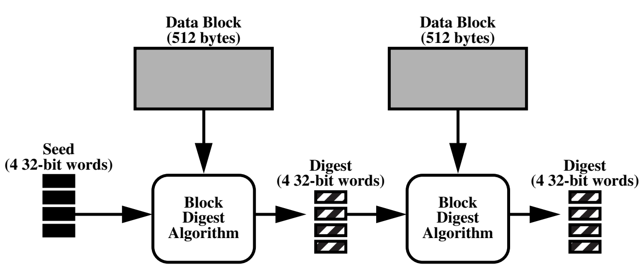

FIGURE 1. MD5 block-chained digest algorithm

Each 512-byte block is digested in 4 phases. Each phase consists of 16 basic steps, for a total of 64 basic steps. Each step updates one word of a 4-word accumulated digest, using the entire intermediate digest as well as block data and constants. In general, each basic step depends on the output of the prior step, defeating simple parallelization of the steps. The basic structure of the steps is shown below (<<< denotes *rotate*). The accumulated digest is denoted by {A,B,C,D}, as in RFC-1321 [28]:

```
A = B+((A+ F(B,C,D) + X[i++] + k1)<<<k2)
D = A+((D+ F(A,B,C) + X[i++] + k1)<<<k2)
C = D+((C+ F(D,A,B) + X[i++] + k1)<<<k2)
B = C+((B+ F(C,D,A) + X[i++] + k1)<<<k2)
```

There are 16 steps based on each of 4 logical functions; 4 based on F are shown here. The constants k1 and k2 are not necessarily identical in basic steps, and are not relevant to this analysis. The logical functions (^ denotes *xor*) are:

```
F(x, y, z) = (((x) & (y)) | ((~x) & (z)))
G(x, y, z) = (((x) & (z)) | ((y) & (~z)))
H(x, y, z) = ((x) ^ (y) ^ (z))
I(x, y, z) = ((y) ^ ((x) | (~z)))
```

The steps have optimization limitations, due to the mathematical properties of the operations used:

- additions can be reordered by commutative laws,
- rotate does not distribute over addition, and
- addition does not distribute over rotation or logicals

2: Software Implementation Measurements

The MD5 RFC-1321 includes a reference implementation written in *C* [28]. The performance of this software gives a baseline against which to compare optimizations. Measurement of the performance of the reference implementation precedes the abstract analysis to determine the focus of the analysis, and the extent of the optimization required.

The MD5 reference implementation software was measured in four configurations on a variety of machines. The Raw configuration used the reference code as provided in the RFC, with modifications to per-process time measurement rather than "wall-clock" time and cache management. The Optimized configuration included the modifications described in Section 2.1, notably byte-order optimizations. Both configurations were executed with 50 passes over alternating blocks of 2 M bytes, to avoid observing cache effects (the i486 used 100 runs over blocks, due to limitations of the 486 platform). This also emphasized the performance of the block digest component, rather than the "housekeeping" overhead. The results are presented in Table 1 and Figure 2.

Two additional runs measured the performance using external (off-chip) and internal (on-chip) caches. The *External-Cache* run executed 5,000 passes over a single 20,000 byte block. The *Internal-Cache* run executed 20,000 passes over a single 5,000 byte block (the HP712 used 100,000 passes over a single 1,000 byte block, due to hardware limitations). The values chosen for block sizes were based on information on external and internal cache sizes as published with the SPEC Benchmark results [8].

All four configurations used randomized block initializations to remove potential data-dependent performance differences. MD5 is not a data-dependent algorithm, but some architectures could have exhibit data-dependent performance variations (though they did not). The code was compiled under both native and *GNU*-based *C* compilers (where available) with all optimizations.

The optimized code spent 95% of its time in the main decode routine on all machines. On little-endian machines this represents only the MD5 data digest routine. On big-endian machines, 2/3 of the time was spent on MD5 data digest, and 1/3 was spent on data byte reordering, required because MD5 uses little-endian grouping of the byte stream. The byte swap cost was higher than expected due to the data copying involved.

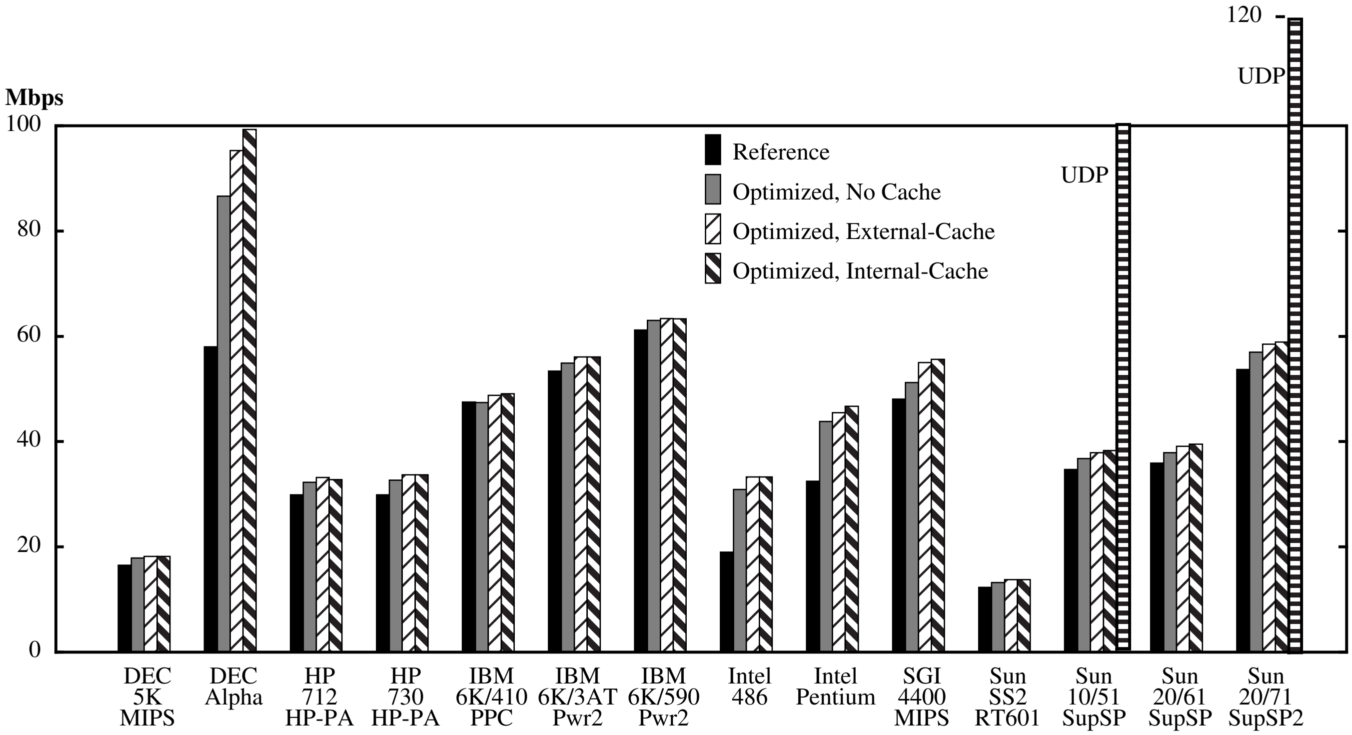

FIGURE 2. Measured performance (plot of values from Table 1)

Host	CPU	CPU (Mhz)	Caches External / Internal	MD5 Rate [no cache] (Mb/s)	Optimized MD5 Rate [no cache] (Mb/s)	In-Cache MD5 Rate: External Cache (Mb/s)	In-Cache MD5 Rate: Internal Cache (Mbps)
Dec 5x33	MIPS 3000	33	128 KB / -	16.5	17.9	18.2	-
Dec 4000 / 710	Alpha	190	4 MB / 8 KB	58.0	86.6	95.3	99.3
HP 712	PA 7100lC	60	64 KB / 1 KB	29.9	32.3	33.2	32.8
HP 9000 / 730	PA 1.1	66	256 KB / -	29.9	32.7	33.7	-
IBM RS6000 / 410	PPC 601	80	512 KB / 32 KB	47.5	47.4	48.8	49.1
IBM RS6000 / 3AT	POWER2	59	64 KB / 32 KB	53.4	54.9	56.1	56.1
IBM RS6000 / 590	POWER2	66.6	256 KB / 32 KB	61.2	63.0	63.4	63.3
Intel	486	66	- / 8 KB	19.0	30.9	-	33.3
Intel	Pentium	90	512 KB / 8 KB	32.5	43.8	45.5	46.7
SGI	MIPS 4400	150	1 MB / 16 KB	48.1	51.2	55.0	55.6
Sun 2	SPARC	40	64 KB / -	12.3	13.2	13.8	-
Sun 10/51	Super-SPARC	50	1 MB / 16 KB	34.7	36.8	37.9	38.3
Sun 20/61	Super-SPARC	60	1 MB / 16 KB	35.9	37.9	39.1	39.5
Sun 20/71	Super-SPARC2	75	1 MB / 16 KB	53.7	57.0	58.5	58.9

TABLE 1. Performance parameters of various platforms[a]

a. Measured performance values are +/-3% for a 95% confidence interval averaged over 20 runs.

Table 1 closely verifies a reported figure of 100 Mbps for a DEC Alpha[1]. This figure was found to occur only for data in the on-chip cache; optimized code run to avoid caching resulted in 86.6 Mbps. This table also indicates the measured speed of UDP/IP on Sun SPARC 2, 10/51 and 20/71 machines (measured at USC/ISI using Fore SBA-200 IP over ATM). Note that even the cache-based optimized software cannot keep pace with the IP capabilities of some of these machines.

2.1: Specific Modifications

The Raw configuration included the following modification:

- Change CPU timer from *gettimeofday()* to *getrusage()*
- Add appropriate code and flags for:
  - use block size, repeat number command-line options.
  - use randomized initialization of test block.
  - double-buffer the test block.

The flags were added to permit run-time configuration, and to provide configuration for cache management. The CPU time measurement was changed to use the more accurate *getrusage()* function, although even that function is known to have potentially large errors on heavily-loaded machines [18]. Lightly-loaded machines were used to avoid such errors.

The Optimized configuration included modifications:

- On little-endian machines, load 32-bit words directly.
  - Avoid byte-swapping and copy overhead altogether.
- On big-endian machines, use more effective swap code.
  - Use a more efficient byte-swap source code.
  - Unroll the byte swap routine.

These optimizations can be grouped into categories:

- cache management
- byte-swapping optimizations
- loop-unrolling

Manual optimizations were limited to overhead code only. Prior to optimization, overhead was 33% on all machines. After optimization, overhead was reduced to 4% on little-endian machines, and 26% on big-endian machines. Manual optimization of the block digest algorithm was not performed after analysis indicated the effective yield would be low (see Section 3).

2.1.1 Cache management

Preliminary test runs indicated that the reference implementation of MD5 exhibited cache effects. This code included a test-mode, in which multiple passes over a single data block emulated the body of a long message. This data block was initialized with data prior to the execution of the digest algorithm. The combination of data initialization and block repetition over-emphasizes the effects of caching on some architectures, especially because the test-block size of 1,000 bytes is within the on-chip cache size of all architectures tested.

This source code was modified to reduce over-emphasis of cache effects. Provisions for double-buffering were added. By alternating the data blocks used during block repetition, the effects of caching were avoided. Command-line configuration of the block size permitted configuration for blocks larger than the cache (with double-buffering, this eliminates cache effects altogether), blocks that fit in the external cache but not the internal cache (without double-buffering), and blocks that fit in the internal cache (without double-buffering). Command-line configuration of the number of block repetitions were used such that the resulting data blocks emulated a 2 M byte stream.

It is not clear whether internal-cache, external-cache, or non-cached measures are more appropriate measure of MD5 performance. On hosts using DMA data transfer, non-cached performance is appropriate. On hosts using programmed I/O, the computation of MD5 might be overlapped with the transfer of IP packets to the network interface, so internal-cache performance is more appropriate. There are also rumored proposed architectures that support DMA directly into the processor's external cache.

2.1.2 Byte-swapping optimizations

The MD5 algorithm uses little-endian groupings of the byte stream, so implementations on native architectures can avoid the byte ordering and copying routine. Avoiding the reordering and copying resulted in code that ran in 2/3 the original time, as expected. The specification of little-endian byte order is opposite that of "network standard byte order", which is big-endian [24].

On big-endian architectures, the reordering was optimized by replacing it with a more efficiently compiled source code. The reference code used the following byte-swapping code, which is machine independent (i.e., correct on both big- and little-endian machines), but inefficient:

```
out[i]  =  ((u_int)in[j])
        |  (((u_int)in[j+1]) << 8)
        |  (((u_int)in[j+2]) << 16)
        |  (((u_int)in[j+3]) << 24)
```

This code compiles to the following pseudo-assembly, yielding 4 loads, 6 internal operations, and one store:

```
loadb  in@j      r1
loadb  in@j+1    r2
loadb  in@j+2    r3
loadb  in@j+3    r4
shl    r2,#8,r2
or     r2,r1,r1
shl    r3,#16,r3
or     r3,r1,r1
shl    r4,#24,r4
or     r4,r1,r1
stw    r1,out@i
```

The following code was found to run faster:

```
/* left rotate */
#def ROL(x,n)  ((x)<<(n))|((x)>>(32-(n))))
temp1  =    ROL(((u_int)input[i]),16);
temp2  =    temp1 >> 8;
temp1  &=   0x00ff00ff;
temp2  &=   0x00ff00ff;
temp1  <<=  8;
out[i] =    temp1 | temp2;
```

The second code compiles to 8 internal operations on machines with no rotate or swap opcodes, and uses a single 32-bit load (see Table 2). This sequence has more internal operations (8, vs. 6 before), but is much more efficient in its interaction with memory (1 word-load, vs. 4 byte-loads before). This code runs 25% faster than the original time. On big-endian machines, this was responsible for an overall speedup of approximately 9% (25% * 33% of the code is a reduction of 8.3% in the overall speedup, or a speedup of 1/0.917 = 9% faster).

On the HP PA-RISC architectures, this source compiled to only 6 internal instructions because it has a 32-bit rotate instruction[1]. The following code took 1 load, 1 store, and 5 internal operations, because this machine has a 32-bit rotate (see Table 2). As a result, the number of internal operations further decreased from 6 to 5, an

1. Personal communication, R. Atkinson, November 1994.

additional 1% improvement in overall optimization. This code was not used on other machines because there it would generate a (longer) sequence of 9 internal instructions.

```
out[i] = (ROL(in[i],8) & 0x00ff00ff)
       | ROL(in[i] & 0x00ff00ff,24);
```

The following table summarizes the length in opcodes of the most efficient byte reordering algorithms, based on given opcode capabilities. It assumes a single 32-bit load before, and a single 32-bit store afterwards:

Opcode Capability	Optimal Length	Optimal sequence
Full swap	1	`swap r1,r1`

16- and 32-bit rotates	3	`rot32 r1,#16,r1`
		`rot16 r1H,#8,r1H`
		`rot16 r1L,#8,r1L`

32-bit rotate only	5	`rot r1,#8,r2`
		`and r2,#mask,r2`
		`and r1,#mask,r1`
		`rot r1,#24,r1`
		`or r1,#r2,r1`

32-bit shift only	8	`shr r1,#16,r2`
		`shl r1,#16,r3`
		`or r2,r3,r4`
		`shr r4,#8,r5`
		`and r4,#mask,r4`
		`and r5,#mask,r5`
		`shl r4,#8,r4`
		`or r5,r4,r6`

TABLE 2. Optimal swap depends on opcodes available

This optimization replaces 4 loads and 6 compute operations with 1 load and 8 compute operations. The result is a trade of 3 loads for 2 computes. Even on current RISC architectures, loads take multiple clocks, due to memory access delays (Figure 3). Loads are typically scheduled and queued, such that subsequent operations on registers pending loads will stall until the load completes. This trade-off, between computation and memory access, can be used to develop a faster hash algorithm. (Section 6.2).

The potential parallelism differs as well. The reference code swap has higher compute parallelism, but no load parallelism. The optimization allows 2-way compute parallelism. It also allows pipelined loads, because the load time is small compared to the compute time (the height of the icons in Figure 3 is approximately representative of execution time).

On the architectures examined, a 25% speedup was measured. Even given the complex interaction of register scheduling and load queues, the increase in compute time is more than accounted by the reduction in load time.

2.1.3 Loop-unrolling

Loop-unrolling is provided by many current compilers. In the case of the MD5 reference code, more efficient loop-unrolling was possible by converting array indices into directly incremented pointers. Manual loop unrolling was required for the minimal byte-swapping optimizations described in Section 2.1.2.

1. HP PA-based machines do not strictly have a rotate instruction. They have a shift-right instruction that can use two copies of a 32-bit register as a virtual 64-bit source, i.e., "shr (r1,r1) #8 r2". The result is effectively a 32-bit rotate instruction in one opcode.

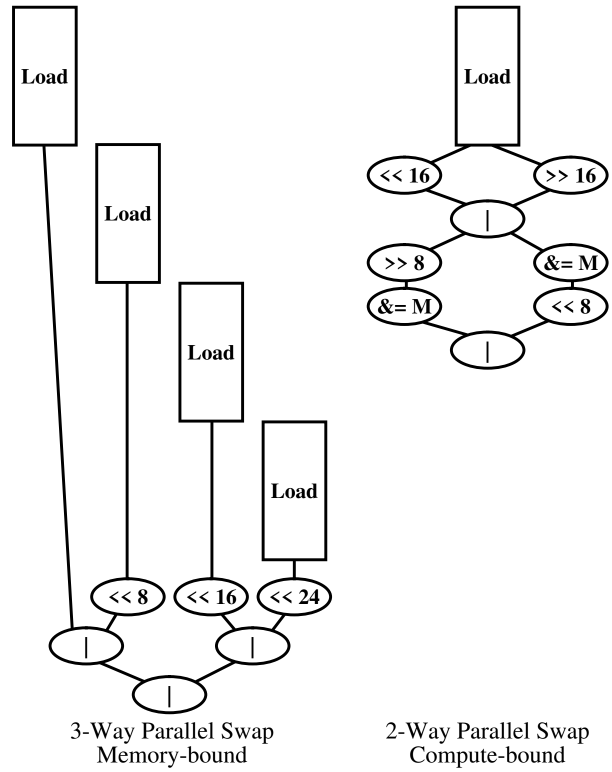

FIGURE 3. Data flow analysis of swap execution

3: Software Analysis

Analysis of the manual optimizations in Section 2.1 indicated that analytical bounds were sufficient predictors of performance limitations (e.g., swap analysis). Manual optimization of the main digest algorithm was postponed to determine the analytical performance speedup potential. The main block digest algorithm differs from the byte-swap algorithm, in that byte-swapping should be a very efficient operation on many machines. Analysis of the main digest algorithm is more complex.

3.1: Analysis of MD5 costs

The MD5 algorithm costs are proportional to the cost of a basic step. For each word of input, 4 basic steps are executed. These basic steps have little opportunity for pipelining or parallelization. Thus by analyzing the costs of a basic step, the overall performance limit can be determined. The basic step can be mapped as a data flow diagram (Figure 4). The critical path in this diagram is indicated by the black lines; the other paths are in grey. In this diagram, dependencies on previous steps (as indicated with a star '*') have been delayed (pushed as low as possible).

The time to process a basic step depends on the time to process each step, as well as the amount of parallelization possible. As described before (Section 2.1.2), the cost of a *rotate* operation can range from 1-3 opcodes on a RISC machine; CISC machines exhibit similar variability in the number of clock cycles required to execute this instruction. The cost of the logical operations also vary, depending on whether the processor has a separate or combined *AND-NOT* and OR-*NOT* instructions. Table 4 indicates the cost of the *AND-NOT*/*OR-NOT* and *rotate* instructions on various processors. The time to processes the logical functions F,G,H,I can similarly be diagrammed both without and with the *AND-NOT/OR-NOT* operators (Figure 5 and Figure 6). In each case, X is pushed as low (late) in the diagram as possible.

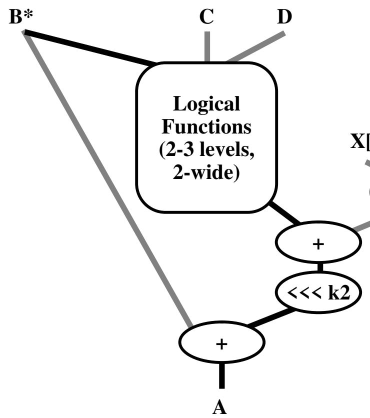

FIGURE 4. Dataflow timing diagram for a basic step

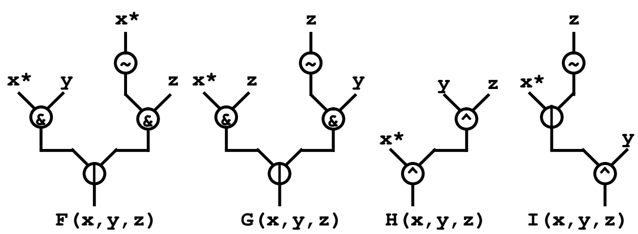

FIGURE 5. Dataflow diagrams for functions F,G,H, and I

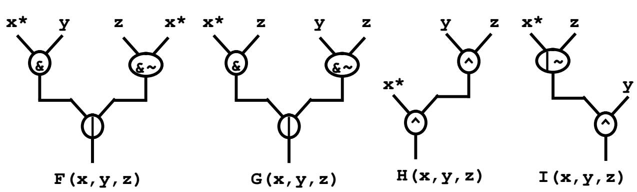

FIGURE 6. Function dataflows with *AND-NOT / OR-NOT*

3.1.1 Serial cost

The serial cost to execute the basic step in Section 1.2.1 is the sum of the cost of executing its opcodes. There are 4 additions, 1 *rotate*, and the cost of the logical operation. On a *NOT only* machine, the logical functions F,G,H,I cost 4,5,2, and 3, averaging to 3.5. On an *AND-NOT* machine, F,G,H, and I cost 3,3,2, and 2, averaging 2.5. The total is 7.5 + *rotate* on a *NOT only* machine, and 6.5 + *rotate* on an *AND-NOT* machine. As noted before (Section 2.1.2), *rotate*s cost either 1 or 3 opcodes. The resulting cost for serial operations is indicated in Table 3. The cost is presented as an average of the costs for each of the different logical functions (i.e., 4 times the average is the cost of executing one of each step).

3.1.2 Parallel cost

Critical height denotes the number of operations between the highest appearing data dependency and the output (Figure 4). In the case where explicit NOTs are required (Figure 5), the respective critical heights for F,G,H,I are 3,2,1, and 2, for an average of 2. In the case where *AND-NOT* is available, the critical heights are 2,2,1, and 2, and the average is 1.75. The overall height of F,G,H,I include 2 additions and 1 shift, or a total of 4 + shift on a *NOT* machine, and 3.75 + shift on an *AND-NOT* machine. As noted before (Section 2.1.2), rotates take between 1 and 3 opcodes. The matrix of possibilities for parallel architectures is shown in Table 3.

	NOT only		*AND-NOT / OR-NOT*	
	1-ROT	**3-ROT**	**1-ROT**	**3-ROT**
Serial	8.5	10.5	7.5	9.5
Parallel	5	7	4.75	6.75

TABLE 3. Costs of basic steps in opcodes[a]

a. Fractional costs indicate averages. Also, some CPUs have only one of AND-NOT / OR-NOT.

In most cases, there is a parallelism potential of 2-3 integer operations. This is probably the reason the Sun SPARC 20/71, with a Super-SPARC2 2-way integer parallel CPU, achieves more of a speedup compared to a 20/61 than CPU speed differences alone account (Table 1). If CPU speed alone were the difference, we would expect the Sun 20/71 to operate at 88 Mhz, rather than its real value of 75 Mhz.

On most machines there were 1-4 opcodes of potentially unnecessary overhead. Subsequent examination indicated that the overhead was required, either for explicitly loading constants (1-2 opcodes on the Sun, Intel, HP, and Dec Alpha), or for zeroing out high-ends of 64-bit registers (2 opcodes on the Dec Alpha). This leaves little opportunity for manual optimization.

There are 4 basic steps executed for each word of input and 10-12 opcodes per basic step, so there are a total of approximately 40-50 opcodes required per word of input (Table 4). This is inordinately high, and will limit the bandwidth of processors to a maximum of 1/50th their MIPS rate. Table 4 also indicates that the potential for speedup (right-most column) is limited.

Machine	**CPU**	**AND/NOT Opr.**	**Rot. Opr.**	**Serial Limit**	**Obs.**	**% Poss.**
Dec 5x33	MIPS 3000	2	3	10.5	10.5	1.00
Dec 4000/710	Alpha	1	3	9.5	13.75	1.45
HP 712	PA 7100lC	1.5	1	7.75	9.75	1.26
HP 9000/730	PA1.1	1.5	1	7.75	9.75	1.26
IBM RS/410	PPC 601	1	1	7.5	8	1.07
IBM RS/3AT	POWER2	1	1	7.5	8	1.07
IBM RS/590	POWER2	1	1	7.5	8	1.07
Intel	486	2	1	8.5	10.5	1.24
Intel	Pentium	2	1	8.5	10.75	1.26
SGI	MIPS 4400	2	3	10.5	10.5	1.00
Sun 2	SPARC	1	3	9.5	11.5	1.21
Sun 10/51	Super-SP	1	3	9.5	11.5	1.21
Sun 20/61	Super-SP	1	3	9.5	11.5	1.21
Sun 20/71	SuperSP2	1	3	9.5	11.5	1.21

TABLE 4. Measured opcode cost of a basic step

The software analysis indicates that the serial execution speed dominates the current cost of the MD5 algorithm. Superscalar RISC CPUs, which can issue multiple integer operations in a single clock, are of limited use. MD5 supports a parallelism of 2-3 integer operations in software. CPUs optimized for floating-point operations are of no help, because MD5 is an integer only algorithm.

MD5 also inhibits software pipelining, by the frequency of the reuse of the intermediate hash data. This, combined with analysis of the opcodes used for MD5, indicate that MD5 may not be optimally tuned to the expected opcode scheduling of current processors. Other hash algorithms may be able to exploit the trade-offs in load vs. compute latency or unused memory bandwidth (if any) (Section 6.2).

4: Hardware Analysis

The software analysis indicated a performance bound that is insufficient to keep pace with IPv4 implementations. A hardware analysis was performed to determine the speed and size of a potential hardware implementation.

The parallel implementation was used for the hardware analysis, based on the basic step dataflow diagram (Figure 4). The hardware design is a clocked dataflow implementation of the abstract dataflow diagram, including the critical path indicated in black (Figure 7). The 3-input logical functions F,G,H,I were replaced with a single logical block. The four adders remain.

For VLSI CMOS implementation, the one fast adder has a single-cycle time of 3.2 ns, and requires 3.2 ns precharging [14]. This adder would run at approximately 300 Mhz, and require two clocks per add (i.e., supporting a 150 Mhz processor). The *rotate* uses with a zero-cost wiring permutation in the best case, and logical circuit in the worst. CMOS latch setup time can be as low as 2 ns. As a result, we can design a clocked CMOS circuit with a 5.2 ns - 3.2 ns clocks (dictated by the adder) and 2 ns latch setups.

The critical path through the basic step is 6 clocks long, i.e., 31.2 ns for the critical path of each basic step (Figure 8). Given 4 basic steps per word of input, there are 124.8 ns per word of input, or 256 Mbps.

A multi-chip CMOS implementation would require 15 ns per 32-bit addition and 8 ns per logical function or rotate (including off-chip driver delays). The resulting speed of the critical path (two adds, one logical, one *rotate*) would be 46 ns per step, or 184 ns per word of input. The resulting system would require approximately 9 chips (2 adders, 4 PLD logical units, 1 shifter, 1 RAM, 1 register file) speed would be 175 Mbps.

Both implementations assume a parallelization of 2 logical operations and 2 32-bit additions, as well as near-zero times for register update (due to write-through to the next stage). This design requires a parallelism of 2 adders, 4 logical units, and 1 shifter (Figure 7). The chip requires the following:

- Functional units
  - 2 high-speed 32-bit adders
  - 4 high-speed logical units (one for each F,G,H,I)
  - 1 high-speed barrel-shifter
- Storage units
  - ROM of 64 words of 37-bits (32-bit k1, 5-bit k2)
  - 32 RAM/register entries of 32-bit block data
  - 12 32-bit registers for the accumulated digest

The functional units are obvious from the dataflow diagrams. The ROM represents the addition and rotation constants for each step. The RAM comprises two buffers of data blocks, such that one can be used for computation while the other is being loaded. The registers store three sets of the hash - one being currently updated, one from the previous block (for block chaining), and one accessible externally during the current hash. This amount of storage and function is feasible in existing custom CMOS and possibly in GaAs, but probably would not be feasible in ECL.

As with the software, this is not sufficient to keep pace with existing hardware for IP, capable of speeds in excess of 1 Gbps.

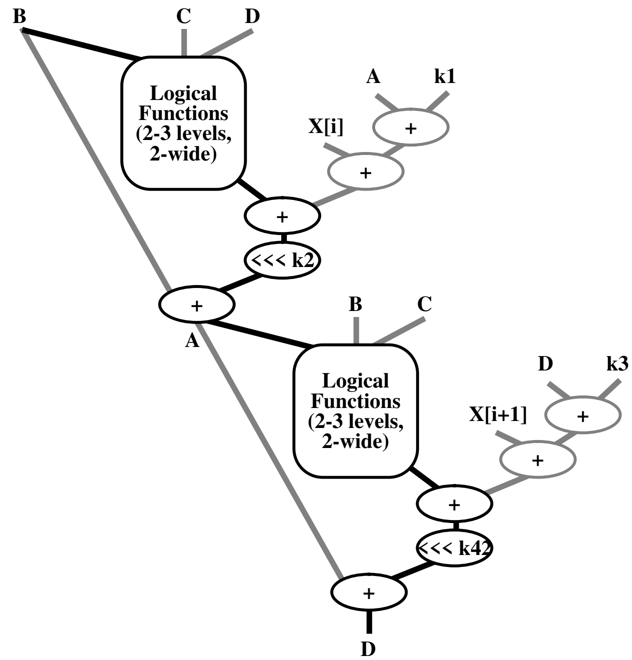

FIGURE 7. Hardware timing diagram (2 steps shown)

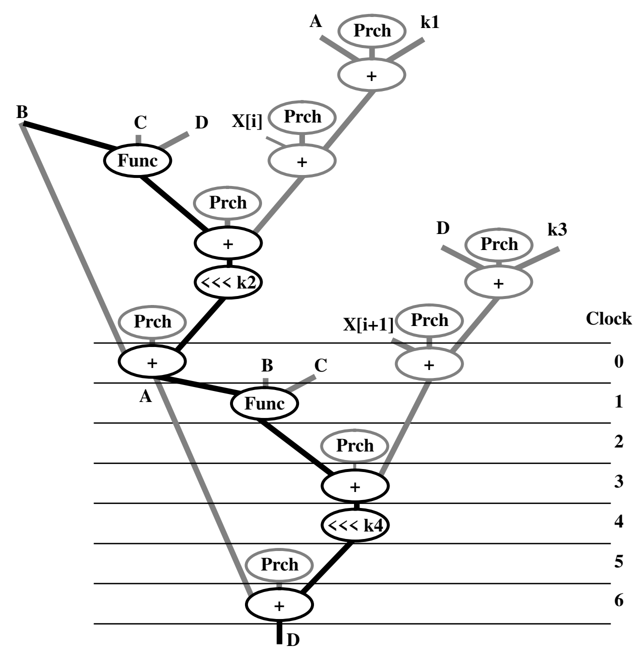

FIGURE 8. Timing for a clocked, precharged circuit

5: A Word About MD4 and other alternatives

The analysis of MD4 is similar to that of MD5. MD4 uses only 3 phases of 16 basic steps, so the cost is 3 basic steps per word of input, rather than 4 [27]. The critical path of a basic step is further reduced by the final addition, so the resulting algorithm can run with one add, one logical function, and one *rotate*. MD4 is expected to run in 2/3 the time of MD5. MD4 is not necessarily the best substitute for MD5, due to security considerations.

Implementations of other algorithms have been measured as follows. MD5 is given for comparison; all measurements are in software on a Sun SPARC 20/71:

- MD2 [15] - 1.8 Mbps
- SHA [22] - 30 Mbps
- DES [21] - 20.6 Mbps
- MD5 - 57 Mbps

The result is that current alternatives are slower than MD5 in software. MD2 and SHA are message digest algorithms; DES is an encryption algorithm.

5.1: Expected hardware acceleration

DES has a hardware implementation that runs at 1 Gbps in GaAs [19]. This factor of 50x improvement in hardware is not a representative speedup for arbitrary algorithms. MD5 in particular is difficult to accelerate in hardware.

DES can be accelerated by a factor of 50x in hardware, because its basic operations (bit select and bit-logical) are particularly slow in software (3-4 opcodes each), and trivially fast in hardware (implemented with a wire).

The basic operations of MD5 are 32-bit additions. The fundamental clock rate of a CPU in a technology is largely limited by the speed at which these additions can occur. There is no advantage to building custom hardware; little further speedup is possible. As a result, MD5 can be accelerated by a factor of 4x in hardware.

Arbitrary algorithms can be accelerated by a factor of 10x when moved from software to hardware (this is an estimate).

6: Suggested Courses of Action

The MD5 algorithm, whether in hardware or software, cannot keep pace with existing IPv4 implementations. As a result, it violates the requirements of IPv6, of "first doing no harm" in reducing processing costs compared to IPv4. Although most mechanisms using MD5 propose a per-message digest, parallelization is prohibitive because a software implementation is desired. In addition, streamed MD5 throughput will always be fundamentally limited.

There are several alternatives to be considered, the best of which are:

- Modify MD5 to increase its performance
- Propose an alternative hash algorithm

6.1: MD5++

MD5 has performance limitations and design choices that affect its use in IPv6. First and foremost, the choice of little-endian byte-order is a detriment to its use in the Internet, where network-standard byte order is big-endian. Redefining MD5++ to use big-endian order will increase its implementation speed by 33% on native architectures. This argument is based on the assumption that IP processing should uniformly occur in big-endian order, because much of it already is. This modification will have no effect on the security analysis of the algorithm.

Another modification to MD5 would permit parallelization, both in custom hardware and software on integer-parallel architectures. There are several proposals to modify MD5 to permit finite parallelism over a single stream. One solution replaces chaining with per-block seeds [8]. Each seed is computed as a hash of the offset of the block in the stream, thus retaining the block-order dependency of block chaining. It does not retain the property that the digest of a block is dependent on the contents of all prior blocks. Another proposes to replace a linear block-chain (Figure 9) with a finite number of block chains (Figure 10). A predetermined finite number of chains are processed from independent seeds, such that the I-th block is part of the "I mod K"-th chain. The resulting sequence of K digests forms another message, which can be MD5-encoded using a single-block algorithm[1]. This supports finite parallelism to provide adequate bandwidth at current processing rates, without providing arbitrary power for spoofing. Further analysis is required to determine its authentication properties.

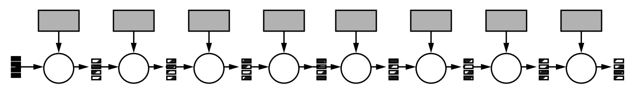

FIGURE 9. MD5 linear block-chained digest algorithm

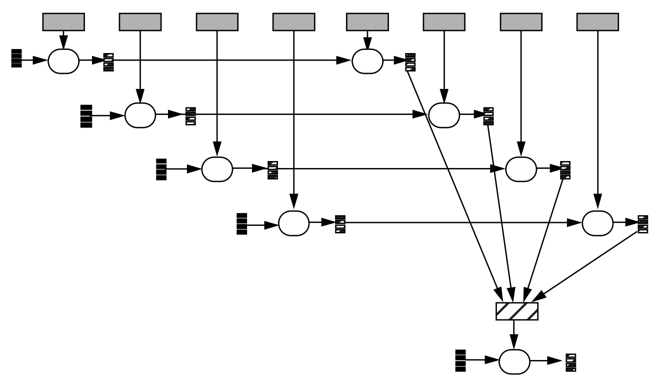

FIGURE 10. MD5 interleaved block-chained digest algorithm (with final digest-of-digest shown)

This proposal increases the performance of MD5 over continuous streams of blocks. IPv6 uses per-header authentication digests, so that a simple per-packet parallelization would suffice. This assumes that packet digests are not chained for other reasons. Neither parallel solution increases the performance of MD5 on a single processor. The IPv6 performance criterion requires implementations to operate as fast as IPv4, presumably on a single processor. MD5 does not, partly due to the complexity of the algorithm, and partly due to its processing the entire body of a packet (rather than only the header).

These two proposals together, of big-endian byte order and 4-way parallelism, comprise a modified MD5 that we call "MD5++".

1. proposed independently by the author [30] and by Burt Kaliski of RSA, Inc.

6.2: An alternate hash algorithm

In the process of analyzing MD5, there are several design principles that were observed. Some affect the performance of hardware, some affect the performance of software:

- Avoid bit-based operations (slows software)
- Avoid carry-based scrambling (slows hardware)
- Realize that rotates can be costly (as much as 3 opcodes)
- Use finite parallelism (will help software and hardware)

These techniques have been used elsewhere to develop hash algorithms with as little as ~10 opcodes per 32-bit word of input. down from the 40-50 that MD5 uses [29]. Performance measurements for that algorithm were not available at the time of publication, but that estimate uses self-modifying code, which may affect its performance on processors with write-through instruction caches.

In RC5, a parameterized number of phases of data-dependent rotates are used to compute a symmetric block-cipher (comparable to DES) [26]. For a 32-bit quantity to have the resulting hash depend on a 32-bit data-dependent rotate would require 7 rounds of 5-bit hashes. Data-dependent rotates are costly in software, requiring 2 additional opcodes per rotate (mask constant, shift data to right of register), not including the fact that rotates are not single opcodes on many RISC processors. As a result, this algorithm may not be competitive with MD5 in performance, although analysis was not possible prior to this publication. RC5 has the feature of being parameterized, such that the levels of security and performance can be adjusted by changing the number of rounds, word size, and key size. This is a useful feature for any hash or encryption algorithm.

An alternate hash algorithm can be developed by using some of these recognitions, together with a few trade-offs. Note that hashing does not have the advantage of off-line precomputation of one-time pads that suffices for encryption. By design, a hash must access every word of input with a high level of “strength”, to provide authentication over the entire data stream in a uniform fashion.

The alternate hash algorithm (called “AHA” for brevity only) should provide cross-bit scrambling like 23-bit addition or data-dependent rotates, but more efficiently. A lookup-table provides arbitrary bit scrambling, of which rotates and additions are instances. Such a lookup table also trades space and memory bandwidth for computation speed. Provided that the table can reside in a local cache, a performance advantage might be achieved.

The AHA is based on the notion of finite-state-machine hash algorithms. It uses 16-bit table lookups, because 64 K-word tables can be stored effectively current caches. The basis of the algorithm is as follows:

```
state  ^= table[state ^ data[i]]
```

AHA grabs the data in 16-bit units, XOR’s that with the current state, looks the result up in a 64 K table of 32-bit words, and XORs the result back into the state. By keeping 8 words of state and weaving the state variable accesses, an integer parallelism of 8 is achieved.

The AHA algorithm is thus based on FSM hashes. The table lookup depends on the current state and the input, thus making the algorithm input-order sensitive, although allowing finite parallelism of the input data (because there are 8 half-words of state). The table is small enough to fit in a local cache. The use of lookups replaces that of rotates or additions, to perform thorough scrambling of the data. The table can be either constant or session-dependent, as required to ensure sufficient authentication without unnecessary overhead.

This algorithm was measured on a Sun SPARC 20/71, and uses 13 opcodes per 32-bit word of input on a SPARC, and 10 opcodes per 32-bit word of input on a 68040. The algorithm achieved 115 Mbps without caching, and 121 Mbps with caching on the Sun 20/71, compared with 57-59 Mbps for MD5. Analysis of the implementation may yield up to another 50% increase in performance. The current performance is sufficient to support existing IP rates on that host. Further performance analysis and cryptographic strength analysis is underway.

AHA is believed to be useful as a minimal hash, because it is derived from the principles of performance first. Even if its cryptographic strength is low, it may prove useful for the authentication of high bandwidth traffic, where the sheer volume of data obviates the need for the same strength as off-line data would require.

7: Conclusions

MD5 cannot be implemented in existing technology at rates in excess of 100 Mbps, and cannot be implemented in special-purpose CMOS hardware feasibly at rates in excess of 175 Mbps. MD5 cannot be used to support IP authentication in existing networks at existing rates. Although MD5 will support higher bandwidth in the future due to technological advances, these will be offset by similar advances in protocol processing. The MD5 mandate in IPv6 should be reconsidered.

At a minimum, the IPv6 specification should recognize the performance limitations of MD5. The use of MD5 in high-performance environments should be recommended against. The modified MD5++ or alternate hash algorithms, as well as other hash algorithms, should be considered before a default standard is specified.

The source code for the optimized version of MD5 presented here is available at <ftp://ftp.isi.edu/pub/hocc-papers/touch/md5-opt.tar.Z>. Current information on the status of this work is available at <http://www.isi.edu/div7/atomic2/md5.html>.

This document was prepared with the assistance and feedback of Steve Kent at BBN, Burt Kaliski, Victor Chang, and Steve Burnett at RSA, and Ran Atkinson at the NRL. Mike Carlton of USC/ISI assisted with the byte-swapping code, cache interaction, and performance measurement analysis. The alternate hash algorithm (AHA) was developed in conjunction with Amir Herzberg, Hugo Krawczyk, and Moti Yung of IBM.

8: References

[1] Atkinson, R., “IPv6 Authentication Header,” (working draft - draft-ietf-ipngwg-auth-00.txt), February 1995.

[2] Atkinson, R., “IPv6 Security Architecture,” (working draft - draft-ietf-ipngwg-sec-00.txt), February 1995.

[3] Atkinson, R., “IPv6 Encapsulating Security Payload (ESP),” (working draft - draft-ietf-ipngwg-esp-00.txt), February 1995.

[4] Baker, F., and Atkinson, R., “OSPF MD5 Authentication,” (working draft - draft-ietf-osp5-md5-03.txt), March 1995.

[5] Baker, F., and Atkinson, R., “RIP-II Cryptographic Authentication,” (working draft - draft-ietf-ripv2-md5-04.txt), March 1995.

[6] Bradner, S., and Mankin, A., "The Recommendation for the IP Next Generation Protocol," RFC 1752, Harvard University, USC/Information Sciences Institute, January 1995.

[7] Deering, S., "Simple Internet Protocol Plus (SIPP)," (working draft - draft-ietf-sipp-spec-01.txt), July 1994.

[8] DiMarco, J., "Spec Benchmark table, V4.12" <ftp://ftp.cdf.toronto.edu/pub/spectable>.

[9] Feldmeier, D., and McAuley, A., "Reducing Protocol Ordering Constraints to Improve Performance," in *Protocols for High-Speed Networks, III*, Eds. Pehrson, B., Gunningberg, P., and Pink, S., North-Holland, Amsterdam, 1992, pp. 3-17.

[10] Galvin, J., and McCloghrie, H., "Security Protocols for version 2 of the Simple Network Management Protocol(SNMPv2)," RFC 1446, Trusted Information Systems, Hughes LAN Systems, April 1993.

[11] Heffernan, A., "TCP MD5 Signature Option," (working draft - draft-hefferman-tcp-md5-01.txt), March 1995.

[12] Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification," (working draft- draft-ietf-ipngwg-ipv6-spec-01.txt), March 1995.

[13] Hostetler, J., and Sink, E., "A Proposed Extension to HTTP: SimpleMD5 Access Authorization," (work in progress).

[14] Irissou, B., Design *Techniques for High-Speed Datapaths*, Master's Thesis, University of California at Berkeley, CSD, November 1992.

[15] Kaliski, B., "The MD2 Message-Digest Algorithm," RFC-1319, RSA Data Security, Inc., April 1992.

[16] Leech, M., "Key-seeded MD5 authentication for SOCKS," (working draft - draft-ietf-aft-socks-md5-auth-00.txt), October 1994.

[17] Malkin, G., "RIP for IPv6," (working draft - draft-ietf-ripv2-ripng-00.txt), November 1994.

[18] McCanne, S., and Torek, C., "A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling," Proc. Winter USENIX, San Diego, January 1993.

[19] Metzger, P., Karn, P., and Simpson, W., "The ESP DES-CBC Transform," (working draft - draft-ietf-ipsec-esp-des-cbc-04.txt), April 1995.

[20] Metzger, P., and Simpson, W., "IP Authentication using Keyed MD5," (working draft - draft-ietf-ipsec-ah-md5-03.txt), April 1995.

[21] National Bureau of Standards, *Data Encryption Standard*, Federal Information Processing Standards Publication 46, Government Printing Office, Washington, D.C., 1977.

[22] National Institute for Standards and Technology, *Secure Hash Standard*, Federal Information Processing Standards Publication 180, Government Printing Office, Washington, D.C., 1993.

[23] Partridge, C., and Kastenholz, F., "Technical Criteria for Choosing IP The Next Generation (IPng)," RFC 1726, BBN Systems and Technologies, FTP Software, December 1994.

[24] Postel, J., "Internet Protocol - DARPA Internet Program Protocol Specification," STD-5, RFC-791, ISI, September 1981.

[25] Rescorla, E., and Schiffman, A., "The Secure HyperText Transfer Protocol," (working draft - draft-rescorla-shttp-0.txt), December 1994.

[26] Rivest, R., "The RC5 Encryption Algorithm," RSA Data Security Technical Report, April 1995.

[27] Rivest, R., "The MD4 Message-Digest Algorithm," RFC-1320, MIT LCS and RSA Data Security, Inc., April 1992.

[28] Rivest, R., "The MD5 Message-Digest Algorithm," RFC-1321, MIT LCS and RSA Data Security, Inc., April 1992.

[29] Rogaway, P., "Bucket Hashing and its Application to Fast Message Authentication," to appear in Advanced in Cryptology, Crypto '95.

[30] Touch, J., "Report on MD5 Performance," (working draft - draft-touch-md5-performance-00.txt), December 1994.

[31] Touch, J., "Implementing the Internet Checksum in Hardware," (work in progress).