IEEE P802.15
Wireless Personal Area Networks

Project	IEEE P802.15 Working Group for Wireless Personal Area Networks (WPANs)	
Title	**02135r0P803-15_TG3-MAC-Distributed-Security-Proposal**	
Date Submitted	6 April 2002	
Source	Gregg Rasor Motorola 1500 Gateway Blvd Boynton Beach, FL 33426 M/S 100	Voice: (561) 739-2952 Fax: (561) 739-3175 E-mail: Gregg.Rasor@motorola.com
Re:	802.15.3 Call for Security Proposals (02/074r1)	
Abstract	This document contains proposed privacy and security elements for use with the 802.15.3 media access control layer and higher layers.	
Purpose	This document contains a compilation of the changes needed in Clauses 6 and 7 of Draft P802.15.3/D09, December 2001, as well as the text for Appendix B.3, as required to implement the proposal in full.	
Notice	This document has been prepared to assist the IEEE P802.15. It is offered as a basis for discussion and is not binding on the contributing individual(s) or organization(s). The material in this document is subject to change in form and content after further study. The contributor(s) reserve(s) the right to add, amend or withdraw material contained herein.	
Release	The contributor acknowledges and accepts that this contribution becomes the property of IEEE and may be made publicly available by P802.15.	

Responsive to the call for Security Suite proposals, the follows elements are included in this compilation of documents:

02/113r0P802-15_TG3-MAC-Distributed-Security-Proposal.doc
02/114r3P802-15_TG3-MAC-Distributed-Security-Proposal.ppt
02/135r0P802-15_TG3-MAC-Distributed-Security-Proposal.pdf

Specifically, the current proposal consists of the following:

Changes and additions to clause 6 (MLME) and clause 7 (frame formats):

We will use the existing commands for Authenticate, Request Key, Distribute Key, and Deauthenticate. The Challenge command is not needed since the components in the Authenticate command will serve its function.

Changes in the Authenticate command are as follows:

Parameters:

Mode: 0 = entity authentication, 1 = key agreement
Pass: 1, 2, 3
DataObjectLength (2 octets)
DataObject: VAR

The authenticate command will be used to convey information that will support mutual authentication of a requesting entity and also serve to establish key agreement (the KEK).

The frame formats described in Section 7 are useable with our current adaptation. However, the Challenge request and Challenge response formats are no longer needed.

Replacement text for clause 10 and appendix B.3

Replacement text for clause 10 is contained in 02/113r0P802-15_TG3-MAC-Distributed-Security-Proposal.doc.

The appendix is included at the end of this document.

Proposed symmetric key method for payload protection

AES is proposed as the symmetric key method for payload protection. Details are contained in the FIPS 197 Standard that is part of appendix B.3 herein. Further implementation details are contained in 02/113r3P802-15_TG3-MAC-Distributed-Security-Proposal.ppt.

Proposed public key method for payload protection

ECC is proposed as the public key method for payload protection. Details are contained in the ANSI X9.63–2001 Standard that is part of appendix B.3 herein The complete version of ANSI X9.63-2001 has been abbreviated, including only the title page, participants, table of contents, and bibliography, pending receipt of a copyright release from the American Bankers Association. For purposes of this document and pursuant to IEEE policy, the before mentioned publicly available standard document is hereby incorporated by reference as part of this submission. Further architectural implementation details are contained in 02/113r3P802-15_TG3-MAC-Distributed-Security-Proposal.ppt, and specific algorithm implementation details are contained in 02/200r0P802-15_TG3-Mandatory-ECC-Security-Algorithm-Suite.doc.

Annex B.3

Security considerations

THE FUTURE OF ELLIPTIC CURVE CRYPTOGRAPHY

Scott Vanstone

Certicom Corp &
University of Waterloo

May 22, 2001

Outline

1. Discrete Logarithm Systems
2. Why Elliptic Curves?
3. The Elliptic Curve Discrete Logarithm Problem
4. ECC Protocols
5. ECC Implementation and Deployment
6. Conclusions

Discrete Logarithm Systems

Public-Key Cryptography began in 1976 with the discovery by Diffie and Hellman of their Key Agreement Protocol:

- Public parameters: A cyclic group of order n generated by g.
- A selects random $a \in [0, n-1]$ and sends g^a to B over a public but authentic channel.
- A selects random $b \in [0, n-1]$ and sends g^b to A over a public but authentic channel.
- A and B can both compute $K = g^{ab}$.

Security Requirements

Basic requirement: K should be A's and B's *shared secret*.

- It should be computationally intractable for an adversary to compute g^{ab} from g, g^a and g^b. This is known as the *Diffie-Hellman Problem* (DHP).
- A necessary (but not known to be sufficient) condition for the DHP to be intractable is that the *Discete Logarithm Problem* (DLP) be intractable: given g and g^a, find a. Best *generic* algorithm is Pollard's rho algorithm: expected running time is $\sqrt{\pi n/2}$ steps, where $n = \mathrm{ord}(g)$.

Efficiency Requirements

Basic requirement: Computing g^a from g and a should be relatively easy.

- Group elements should have a compact representation. Ideally, each group element should be represented using $\approx \log_2 n$ bits.
- Note: Representation of the group element can have important security implications. After all, every cyclic group of order n is $isomorphic$ to $(\mathbb{Z}_n, +)$ where the DLP is: given 1 and a, find a.
- The group operation with respect to this representation should be fast. (Then exponentiation can be efficiently performed using repeated square-and-multiply.)

Cryptographically Suitable Groups

- Diffie and Hellman first proposed using the multiplicative group $\mathbb{Z}_p^*$, where p is a prime.
- The existence of subexponential-time index-calculus methods for solving the ECDLP in $\mathbb{Z}_p^*$ means that larger parameters have to be used to avoid the best attacks known (e.g., a 1024-bit prime p).
- Fundamental question: Are there finite groups for which:

  (i) the group elements can be compactly represented;

  (ii) the group operation can be efficiently computed; and

  (iii) the discrete logarithm problem cannot be solved in subexponential time?

Why Elliptic Curves?

[The following approach is due to Gerhard Frey]

Abelian Varieties

- Abelian varieties A (over finite fields $\mathbb{F}_q$) provide a rich source of finite groups.
  - Set of solutions to a set of equations, together with a group law given by rational functions.
  - Example (multiplicative group of $\mathbb{F}_q$):
    Equation: $xy = 1$
    Group law: $(x_1, y_1) + (x_2, y_2) = (x_1 y_1, x_2 y_2)$.
  - Example (elliptic curve group over $\mathbb{F}_p$):
    Equation: $y^2 = x^3 + ax + b$
    Group law: usual tangent-and-chord formulas.
  - Problem: The number of variables and the degrees of the addition formulas grow exponentially with the dimension of A.

Jacobian Varieties

- Solution: Take special abelian varieties– Jacobian variety J_C of an algebraic curve C over a finite field $\mathbb{F}_q$.
- J_C can be viewed as the quotient group of zero divisors modulo the principal divisors.
- By Riemann-Roch, each divisor class has a *reduced* divisor: $P_1 + P_2 + \cdots + P_t - tP_\infty$, where $t \leq g$.
- Problem: The points in the support of $D \in J_C(\mathbb{F}_q)$ are not necessarily defined over $\mathbb{F}_q$. Also need an efficient method to add 2 reduced divisors and convert the result to reduced form.

Hyperelliptic Curves

- Solution: Consider the jacobian $J_C(\mathbb{F}_q)$ of a hyperelliptic curve C of genus g over $\mathbb{F}_q$. (J_C is a g-dimensional abelian variety.) Note that $\#J_C(\mathbb{F}_q) \approx q^g$.
- $C : v^2 + h(u)v = f(u)$, $h, f \in \mathbb{F}_q[u]$, $\deg h \leq g$, $\deg f = 2g + 1$.
- Elements of $J_C(\mathbb{F}_q)$ can be uniquely represented by a pair of polynomials $a, b \in \mathbb{F}_q[u]$, where $\deg b < \deg a \leq g$, a is monic, and $a|(b^2 + bh - f)$.
- Cantor's algorithm efficiently adds two elements in this form, and provides the result in this same form.

Hyperelliptic Curves (2)

- Problem: For "large" genus hyperelliptic curves, there are subexponential-time algorithms for the DLP in $J_C(\mathbb{F}_q)$. (Adleman-DeMarrais-Huang, Müller-Stein-Thiel, Enge-Gaudry.)
- Problem: For small genus $g = 5, 6, 7, 8, \ldots$, Gaudry's algorithm (2000) has a running time that is faster than Pollard's rho algorithm.
- Problem: $g = 4$ is too close for comfort.
- Solution: Use $g = 1, 2, 3$ hyperelliptic curves.

Genus 1, 2 and 3

- Problem: The group law for $g = 3$ is significantly more complicated than the group law for $g = 2$.
- Solution: Use $g = 2$.
  - Group law can be described by explicit & simple formulas.
  - However the group law for $g = 2$ is still more complicated that the elliptic curve ($g = 1$) group law.
  - $g = 2$ has a potential advantage over $g = 1$ in that one can use a smaller finite field for the same level of security–this may be advantageous for hardware.
  - However, in general $g = 2$ will be slower than $g = 1$. There is no compelling reason (as yet) to use $g = 2$.
  - So, we are stuck with elliptic curves!

The Elliptic Curve Discrete Logarithm Problem

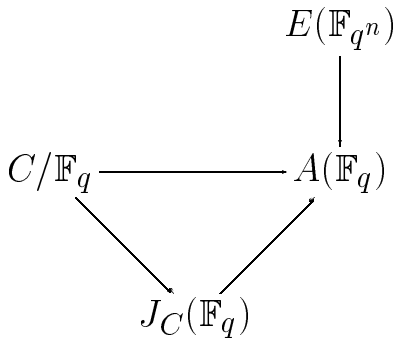
- Let E be an elliptic curve over $\mathbb{F}_q$ with $\#E(\mathbb{F}_q) = nh$, where n is prime and h is small. Let $P \in E(\mathbb{F}_q)$, ord$(P) = n$. ECDLP: Given E, P, $Q \in \langle P \rangle$, find $l \in [0, n-1]$ such that $Q = lP$.
- In general, there is no subexponential-time algorithm known for the DLP in $E(\mathbb{F}_q)$. The best algorithm known is Pollard's rho which takes $(\sqrt{\pi n})/2$ steps.

Special Cases

- (Weil & Tate pairing): Efficiently embed $\langle P \rangle$ in $\mathbb{F}_{q^k}^*$ for small k.
- (Prime field anomalous curves): If $\#E(\mathbb{F}_p) = p$, then ECDLP in $E(\mathbb{F}_p)$ can be efficiently calculated.
- (Frobenius endomorphism): Use equivalence classes of points under Frobenius map to speed up Pollard's rho algorithm for Koblitz curves over $\mathbb{F}_{2^m}$ by a factor of $\sqrt{m}$.

Weil Descent Attack

- $q = 2^l$, $E/\mathbb{F}_{q^n} : y^2 + xy = x^3 + ax^2 + b$.
- Frey (1998); Galbraith & Smart (1999)
- Gaudry, Hess & Smart (2000)



- A: n-dimensional abelian variety over $\mathbb{F}_q$.
  C: genus-g hyperelliptic curve which lies on A.
  $J_C(\mathbb{F}_q) \cong$ subgroup of $E(\mathbb{F}_{q^n})$.

Analysis

- Theorem (Gaudry, Hess & Smart, 2000) $g = 2^{m-1}$ or $2^{m-1} - 1$, where

$$m = m(b) = \dim_{\mathbb{F}_2}(\mathrm{Span}_{\mathbb{F}_2}\{(1, b_0^{1/2}), \ldots, (1, b_{n-1}^{1/2})\}),$$

  where $b_i = b^{q^i}$, $0 \leq i \leq n-1$.

- Notes:

  (a) $1 \leq m \leq n$, so $1 \leq g \leq 2^{n-1}$.

  (b) If $m = 1$, so $g = 1$, the reduction is useless.

  (c) If $m = n$, so $g = 2^{n-1}$, the reduction takes fully exponential time ($\#J_C(\mathbb{F}_q) \approx q^{2^{n-1}}$).

Analysis (2)

Theorem (Menezes & Qu, 2000) Let n be an odd prime, $t = \mathrm{ord}_n(2)$, $s = (n-1)/t$, $b \in \mathbb{F}_{q^n}$. Then

$$m(b) \in \{1, t+1, 2t+1, \ldots, st+1\}.$$

Moreover, the number of $b \in \mathbb{F}_{q^n}$ with $m(b) = it + 1$ is $q\binom{s}{i}(q^t - 1)^i$.

Consequences

1. **Koblitz curves** ($b \in \mathbb{F}_2$).
   - $m = 1$, so $g = 1$.
   - So, GHS attack is useless for Koblitz curves.
2. **Non-Koblitz curves over $\mathbb{F}_{2^n}$, n prime.**
   - If $n \in [160, 600]$, then $m \geq 18$, so $g \geq 2^{17} \approx 128,000$.
   - So GHS attack fails for *all* elliptic curves over $\mathbb{F}_{2^n}$, where n is prime.

3. **Elliptic curves over $\mathbb{F}_{2^{155}}$.**
   (a) $q = 2^{31}$, $n = 5$.
   - $g = 1, 15$ or 16.
   - HCDLP in a genus 15 curve over $\mathbb{F}_{2^{31}}$ appears infeasible (Smart, Eurocrypt 2001).

   (b) $q = 2^5$, $n = 31$.
   - $m = 6$, so $g = 31$ for a small fraction of elliptic curves over $\mathbb{F}_{2^5}$.
   - HCDLP in a genus 31 curve over $\mathbb{F}_{2^5}$ is feasible (Jacobson, Menezes & Stein, 2001).

Certicom's ECC Challenge

- Launched on November 6, 1997.
- See `http://www.certicom.com/`

The Challenges

- The challenge curves are divided into three categories:
  - Randomly generated curves over $\mathbb{F}_{2^m}$ (m prime)
  - Koblitz curves over $\mathbb{F}_{2^m}$ (m prime)
  - Randomly generated curves over $\mathbb{F}_p$ (p prime)
- Within each category, there are 3 subcategories:
  - Exercises
  - Level I challenges
  - Level II challenges

Challenges – Random Curves Over $\mathbb{F}_{2^m}$

	Field size (bits)	Prize (US$)
Exercises		
ECC2-79	79	Book/Maple
ECC2-89	89	Book/Maple
ECC2-97	97	$ 5,000
Level I Challenges		
ECC2-109	109	$10,000
ECC2-131	131	$20,000
Level II Challenges		
ECC2-163	163	$30,000
ECC2-191	191	$40,000
ECC2-238	239	$50,000
ECC2-353	359	$100,000

Challenges – Koblitz Curves

	Field size (bits)	Prize (US$)
Exercises		
ECC2K-95	97	$ 5,000
Level I Challenges		
ECC2K-108	109	$10,000
ECC2K-130	131	$20,000
Level II Challenges		
ECC2K-163	163	$30,000
ECC2K-238	239	$50,000
ECC2K-358	359	$100,000

Challenges – Random Curves Over $\mathbb{F}_p$

	Field size (bits)	Prize (US$)
Exercises		
ECCp-79	79	Book/Maple
ECCp-89	89	Book/Maple
ECCp-97	97	$ 5,000
Level I Challenges		
ECCp-109	109	$10,000
ECCp-131	131	$20,000
Level II Challenges		
ECCp-163	163	$30,000
ECCp-191	191	$40,000
ECCp-239	239	$50,000
ECCp-359	359	$100,000

Results To Date

(Robert Harley, INRIA; Adrian Escott, BT Labs)
Using parallelized Pollard-rho in software:

Challenge	Date solved	Time (group ops)
ECCp-79	Dec 6 97	1.4×10^{12}
ECC2-79	Dec 16 97	1.7×10^{12}
ECCp-89	Jan 12 98	3.0×10^{13}
ECC2-89	Feb 9 98	1.8×10^{13}
ECCp-97	Mar 18 98	2.0×10^{14}
ECC2K-95	May 21 98	2.2×10^{13}
ECC2-97	Sep 22 99	1.2×10^{14}
ECC2K-108	Apr 4 00	1.1×10^{14}

Computational Effort for ECC2-97

- 740 machines (mostly Pentiums and Alphas).
- 20 countries.
- 40 days.
- 1.2×10^{14} elliptic curve points computed.
- $127,497$ distinguished points collected.
- Computing power utilized: About 16,000 MIPS years.
- More than twice as much computing power as used for the recent factorization of RSA-155.

Future Results?

Estimated time to compute logarithms using Pollard-rho.

Challenge	Time (expected)
ECC2-109	2.2×10^{16}
ECCp-109	2.2×10^{16}
ECC2K-130	2.9×10^{18}
ECC2-131	4.6×10^{19}
ECCp-131	4.6×10^{19}

ECC Protocols

- Elliptic curves can be used to design procotols for the basic public-key functions of key agreement, key transport (encryption), and digital signatures.
- Ideally, a protocol should have the following attributes:
  - (i) It should be conceptually simple.
  - (ii) It should be completely specified. (Details are important!)
  - (iii) Is should be well-scrutinized.
  - (iv) It should have a security definition. (What does it mean for the protocol to be secure?)
  - (v) It should have a proof of security (under reasonable assumptions).
  - (vi) It should be standardized (by accredited organizations).

Elliptic Curve Digital Signature Algorithm (ECDSA)

EC Key Pair Generation:

- Domain parameters: E, $\mathbb{F}_q$, $G \in E(\mathbb{F}_q)$, $n = \text{ord}(G)$, $h = \#E(\mathbb{F}_q)/n$.
- Each entity A does the following:
  1. Select a random integer d in the interval $[1, n-1]$.
  2. Compute $Q = dG$.
  3. A's public key is Q; A's private key is d.

ECDSA Signature Generation

To sign a message m, A does the following:

1. Select a random integer k, $1 \leq k \leq n-1$.
2. Compute $kG = (x_1, y_1)$ and $r = x_1 \bmod n$.
   If $r = 0$ then go to step 1.
3. Compute $k^{-1} \bmod n$.
4. Compute $e = \text{SHA-1}(m)$.
5. Compute $s = k^{-1}\{e + dr\} \bmod n$.
   If $s = 0$ then go to step 1.
6. A's signature for the message m is (r, s).

ECDSA Signature Verification

To verify A's signature (r, s) on m, B should do the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute $e = \text{SHA-1}(m)$.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $u_1G + u_2Q = (x_1, y_1)$ and $v = x_1 \bmod n$.
6. Accept the signature if and only if $v = r$.

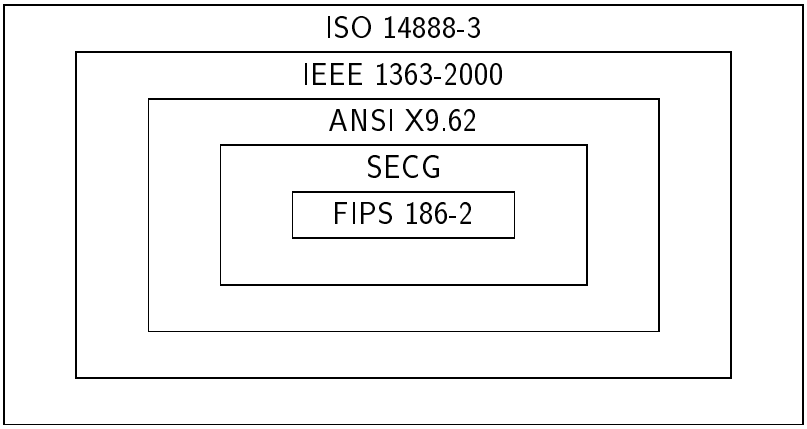
Security Definition

- **Definition**: A signature scheme is said to be *secure* if it is existentially unforgeable against chosen-message attack by a computationally bounded adversary.
- NOTE: The adversary has access to a signing *oracle*. Its goal is to compute a *single* valid message/signature pair for *any* message that was not previously given to the signing oracle.

Security Proof for ECDSA

Theorem (Brown, 2000) ECDSA is existentially unforgeable by chosen-message adversaries assuming that H is a collision-resistant hash function, and that the underlying group is a generic group.

ECDSA Standardization



ECDSA Summary

- ECDSA has the desired design attributes:
  - (i) It should be conceptually simple.
  - (ii) It should be completely specified. (Details are important!)
  - (iii) Is should be well-scrutinized.
  - (iv) It should have a security definition. (What does it mean for the protocol to be secure?)
  - (v) It should have a proof of security (under reasonable assumptions).
  - (vi) It should be standardized (by accredited organizations).

ECC Implementation and Deployment

- The absence of a subexponential-time algorithm for the ECDLP means that significantly smaller parameters can be used in ECC than with competing technologies such as DSA and RSA, but with equivalent levels of security.
- Advantages to be gained from smaller parameters include: speed, and smaller keys and certificates.
- These advantages are especially important in environments where at least one of the following resources are limited: processing power, storage space, bandwidth, power.
- Thus, ECC is especially well-suited for constrained environments smart cards, cellular phones, pagers, PDAs, digital postal marks.

NIST Recommended Elliptic Curves

- Collection of elliptic curves recommended in FIPS 186-2 for use with ECDSA by the US Federal Government.
- Recommended fields:

Block cipher key length	Block cipher	$\mathbb{F}_p$ $\|p\|$	$\mathbb{F}_{2^m}$ m	RSA modulus length
80	SKIPJACK	192	163	1,024
112	Triple-DES	224	233	2,048
128	AES Small	256	283	3,072
192	AES Medium	384	409	7,680
256	AES Large	521	571	15,360

Reduction Polynomials for $\mathbb{F}_{2^m}$

- $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$
- $f(x) = x^{233} + x^{74} + 1$
- $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
- $f(x) = x^{409} + x^{87} + 1$
- $f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

Recommended Curves Over $\mathbb{F}_{2^m}$

- K-163: $y^2 + xy = x^3 + x^2 + 1$ over $\mathbb{F}_{2^{163}}$, cofactor=2.
- K-233: $y^2 + xy = x^3 + 1$ over $\mathbb{F}_{2^{233}}$, cofactor=4.
- K-283: $y^2 + xy = x^3 + 1$ over $\mathbb{F}_{2^{283}}$, cofactor=4.
- K-409: $y^2 + xy = x^3 + 1$ over $\mathbb{F}_{2^{409}}$, cofactor=4.
- K-571: $y^2 + xy = x^3 + 1$ over $\mathbb{F}_{2^{571}}$, cofactor=4.
- Also, 1 randomly generated curve over each of these fields, each having cofactor 2: $y^2 + xy = x^3 + x^2 + b$.
  B-163, B-233, B-283, B-409, B-571.

Recommended Curves Over $\mathbb{F}_p$

- Field sizes:
  - $p = 2^{192} - 2^{64} - 1$.
  - $p = 2^{224} - 2^{96} + 1$.
  - $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.
  - $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$.
  - $p = 2^{521} - 1$.
- All curves were randomly generated and have prime order.
- $y^2 = x^3 - 3x + b$.
- P-192, P-224, P-256, P-384, P-521.

RIM Pager

- Custom-built Intel 386 processor, 10 MHz.
- 2 Mbytes of flash memory, 304 Kbytes of SRAM.
- Single AA battery which lasts roughly three weeks.
- Runs on the Mobitex network.
- Multitasking is cooperative – failure to yield control within 10 seconds can trigger a pager reset.

Palm Pilot

- Palm VII is wireless.
- 16 MHz Motorola 68000-type Dragonball processor.
- 2-8 MB of memory.
- Code segment and stack restrictions.

Timings (in ms)

	K-163			B-163			RSA-1024		
	RIM	Pilot	PII	RIM	Pilot	PII	RIM	Pilot	PII
Key gen	751	1,334	1.47	1,085	1,891	2.12	580,405	1,705,442	2,740
Encrypt	1,759	2,928	4.37	3,132	5,458	6.67	533	1,023	2.70
Decrypt	1,065	1,610	2.85	2,114	3,564	4.69	15,901	36,284	67
Signing	1,011	1,793	2.11	1,335	2,230	2.64	15,889	36,130	67
Verifying	1,826	3,263	4.09	3,243	5,370	6.46	301	729	1.23

- ECC and RSA code optimized for a Pentium II.
- Ported without further optimizations to the Palm Pilot and the RIM pager.
- ECC: Fixed curve (no point counting).
- RSA code from OpenSSL (Eric Young); $e = 3$.

Timings (in ms)

	K-233			B-233			RSA-2048		
	RIM	Pilot	PII	RIM	Pilot	PII	RIM	Pilot	PII
Key gen	1,552	2,573	3.11	2,478	3,948	4.58	—	—	26,442
Encrypt	3,475	5,563	7.83	6,914	11,373	13.99	1,586	3,431	7.26
Decrypt	2,000	2,969	4.85	4,593	7,551	9.55	112,091	292,041	440
Signing	1,910	3,080	4.03	3,066	4,407	5.52	111,956	288,236	440
Verifying	3,701	5,878	7.87	7,321	11,964	14.08	1,087	2,392	4.20

Timings (in ms)

	K-283			B-283		
	RIM	Pilot	Pent II	RIM	Pilot	Pent II
Key gen	2,369	4,062	4.50	3,857	6,245	6.88
Encrypt	5,227	8,579	11.02	11,264	18,273	20.86
Decrypt	2,932	4,495	6.78	7,498	12,046	13.88
Signing	2,760	4,716	5.64	4,264	6,816	8.08
Verifying	5,485	9,059	11.46	11,587	18,753	21.15

VPN Handheld Client (Certicom)

Secure Wireless and Mobile Access to Corporate Intranets

- First IPSec handheld VPN client to secure wireless and mobile device access to corporate intranets.
- Interoperable with VPN gateways from Alcatel, Check Point, Cisco, Intel, Nortel.
- Client runs on Palm, WinCE 3.0, EPOC, and other platforms.
- ECDH and ECDSA; also DH and DSA.
- Small memory footprint:
  - 60-120K for IPSec/IKE protocol.
  - 60K for crypto engines.

Conclusions

- Elliptic curves appear to be the best choice of group for use in discrete logarithm cryptography.
- ECC is now well-accepted and widely standardized.
- ECC is being offered in commercial products by numerous companies, especially for use in constrained environments.

Conclusions

- Elliptic curves appear to be the best choice of group for use in discrete logarithm cryptography.
- ECC is now well-accepted and widely standardized.
- ECC is being offered in commercial products by numerous companies, especially for use in constrained environments.

Federal Information

Processing Standards Publication 197

November 26, 2001

Announcing the

ADVANCED ENCRYPTION STANDARD (AES)

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235).

1. Name of Standard. Advanced Encryption Standard (AES) (FIPS PUB 197).

2. Category of Standard. Computer Security Standard, Cryptography.

3. Explanation. The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

4. Approving Authority. Secretary of Commerce.

5. Maintenance Agency. Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. Applicability. This standard may be used by Federal departments and agencies when an agency determines that sensitive (unclassified) information (as defined in P. L. 100-235) requires cryptographic protection.

Other FIPS-approved cryptographic algorithms may be used in addition to, or in lieu of, this standard. Federal agencies or departments that use cryptographic devices for protecting classified information can use those devices for protecting sensitive (unclassified) information in lieu of this standard.

In addition, this standard may be adopted and used by non-Federal Government organizations. Such use is encouraged when it provides the desired security for commercial and private organizations.

7. Specifications. Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES) (affixed).

8. Implementations. The algorithm specified in this standard may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors such as the application, the environment, the technology used, etc. The algorithm shall be used in conjunction with a FIPS approved or NIST recommended mode of operation. Object Identifiers (OIDs) and any associated parameters for AES used in these modes are available at the Computer Security Objects Register (CSOR), located at http://csrc.nist.gov/csor/ [2].

Implementations of the algorithm that are tested by an accredited laboratory and validated will be considered as complying with this standard. Since cryptographic security depends on many factors besides the correct implementation of an encryption algorithm, Federal Government employees, and others, should also refer to NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, for additional information and guidance (NIST SP 800-21 is available at http://csrc.nist.gov/publications/).

9. Implementation Schedule. This standard becomes effective on May 26, 2002.

10. Patents. Implementations of the algorithm specified in this standard may be covered by U.S. and foreign patents.

11. Export Control. Certain cryptographic devices and technical data regarding them are subject to Federal export controls. Exports of cryptographic modules implementing this standard and technical data regarding them must comply with these Federal regulations and be licensed by the Bureau of Export Administration of the U.S. Department of Commerce. Applicable Federal government export controls are specified in Title 15, Code of Federal Regulations (CFR) Part 740.17; Title 15, CFR Part 742; and Title 15, CFR Part 774, Category 5, Part 2.

12. Qualifications. NIST will continue to follow developments in the analysis of the AES algorithm. As with its other cryptographic algorithm standards, NIST will formally reevaluate this standard every five years.

Both this standard and possible threats reducing the security provided through the use of this standard will undergo review by NIST as appropriate, taking into account newly available analysis and technology. In addition, the awareness of any breakthrough in technology or any mathematical weakness of the algorithm will cause NIST to reevaluate this standard and provide necessary revisions.

13. Waiver Procedure. Under certain exceptional circumstances, the heads of Federal agencies, or their delegates, may approve waivers to Federal Information Processing Standards (FIPS). The heads of such agencies may redelegate such authority only to a senior official designated pursuant to Section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when compliance with this standard would

a. adversely affect the accomplishment of the mission of an operator of Federal computer system or

b. cause a major adverse financial impact on the operator that is not offset by government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision that explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decision, Information Technology Laboratory, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under Section 552(b) of Title 5, U.S. Code, shall be part of the procurement documentation and retained by the agency.

14. Where to obtain copies. This publication is available electronically by accessing http://csrc.nist.gov/publications/. A list of other available computer security publications, including ordering information, can be obtained from NIST Publications List 91, which is available at the same web site. Alternatively, copies of NIST computer security publications are available from: National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, VA 22161.

Federal Information

Processing Standards Publication 197

November 26, 2001

Specification for the

ADVANCED ENCRYPTION STANDARD (AES)

Table of Contents

Table of Figures

1. Introduction

This standard specifies the **Rijndael** algorithm ([3] and [4]), a symmetric block cipher that can process **data blocks** of **128 bits**, using cipher **keys** with lengths of **128**, **192**, and **256 bits**. Rijndael was designed to handle additional block sizes and key lengths, however they are not adopted in this standard.

Throughout the remainder of this standard, the algorithm specified herein will be referred to as "the AES algorithm." The algorithm may be used with the three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256".

This specification includes the following sections:

2. Definitions of terms, acronyms, and algorithm parameters, symbols, and functions;
3. Notation and conventions used in the algorithm specification, including the ordering and numbering of bits, bytes, and words;
4. Mathematical properties that are useful in understanding the algorithm;
5. Algorithm specification, covering the key expansion, encryption, and decryption routines;
6. Implementation issues, such as key length support, keying restrictions, and additional block/key/round sizes.

The standard concludes with several appendices that include step-by-step examples for Key Expansion and the Cipher, example vectors for the Cipher and Inverse Cipher, and a list of references.

2. Definitions

2.1 Glossary of Terms and Acronyms

The following definitions are used throughout this standard:

AES	Advanced Encryption Standard
Affine Transformation	A transformation consisting of multiplication by a matrix followed by the addition of a vector.
Array	An enumerated collection of identical entities (e.g., an array of bytes).
Bit	A binary digit having a value of 0 or 1.
Block	Sequence of binary bits that comprise the input, output, State, and Round Key. The length of a sequence is the number of bits it contains. Blocks are also interpreted as arrays of bytes.
Byte	A group of eight bits that is treated either as a single entity or as an array of 8 individual bits.

Cipher	Series of transformations that converts plaintext to ciphertext using the Cipher Key.
Cipher Key	Secret, cryptographic key that is used by the Key Expansion routine to generate a set of Round Keys; can be pictured as a rectangular array of bytes, having four rows and ***Nk*** columns.
Ciphertext	Data output from the Cipher or input to the Inverse Cipher.
Inverse Cipher	Series of transformations that converts ciphertext to plaintext using the Cipher Key.
Key Expansion	Routine used to generate a series of Round Keys from the Cipher Key.
Plaintext	Data input to the Cipher or output from the Inverse Cipher.
Rijndael	Cryptographic algorithm specified in this Advanced Encryption Standard (AES).
Round Key	Round keys are values derived from the Cipher Key using the Key Expansion routine; they are applied to the State in the Cipher and Inverse Cipher.
State	Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and ***Nb*** columns.
S-box	Non-linear substitution table used in several byte substitution transformations and in the Key Expansion routine to perform a one-for-one substitution of a byte value.
Word	A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.

2.2 Algorithm Parameters, Symbols, and Functions

The following algorithm parameters, symbols, and functions are used throughout this standard:

`AddRoundKey()`	Transformation in the Cipher and Inverse Cipher in which a Round Key is added to the State using an XOR operation. The length of a Round Key equals the size of the State (i.e., for *Nb* = 4, the Round Key length equals 128 bits/16 bytes).
`InvMixColumns()`	Transformation in the Inverse Cipher that is the inverse of **`MixColumns()`**.
`InvShiftRows()`	Transformation in the Inverse Cipher that is the inverse of **`ShiftRows()`**.
`InvSubBytes()`	Transformation in the Inverse Cipher that is the inverse of **`SubBytes()`**.
K	Cipher Key.

`MixColumns()`	Transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns.
Nb	Number of columns (32-bit words) comprising the State. For this standard, ***Nb*** = 4. (Also see Sec. 6.3.)
Nk	Number of 32-bit words comprising the Cipher Key. For this standard, ***Nk*** = 4, 6, or 8. (Also see Sec. 6.3.)
Nr	Number of rounds, which is a function of ***Nk*** and ***Nb*** (which is fixed). For this standard, ***Nr*** = 10, 12, or 14. (Also see Sec. 6.3.)
Rcon[]	The round constant word array.
`RotWord()`	Function used in the Key Expansion routine that takes a four-byte word and performs a cyclic permutation.
`ShiftRows()`	Transformation in the Cipher that processes the State by cyclically shifting the last three rows of the State by different offsets.
`SubBytes()`	Transformation in the Cipher that processes the State using a non-linear byte substitution table (S-box) that operates on each of the State bytes independently.
`SubWord()`	Function used in the Key Expansion routine that takes a four-byte input word and applies an S-box to each of the four bytes to produce an output word.
XOR	Exclusive-OR operation.
$\oplus$	Exclusive-OR operation.
$\otimes$	Multiplication of two polynomials (each with degree < 4) modulo $x^4 + 1$.
$\bullet$	Finite field multiplication.

3. Notation and Conventions

3.1 Inputs and Outputs

The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits** (digits with values of 0 or 1). These sequences will sometimes be referred to as **blocks** and the number of bits they contain will be referred to as their length. The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in one of the ranges $0 \le i < 128$, $0 \le i < 192$ or $0 \le i < 256$ depending on the block length and key length (specified above).

3.2 Bytes

The basic unit for processing in the AES algorithm is a **byte,** a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences described in Sec. 3.1 are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes (see Sec. 3.3). For an input, output or Cipher Key denoted by *a*, the bytes in the resulting array will be referenced using one of the two forms, a_n or $a[n]$, where n will be in one of the following ranges:

Key length = 128 bits, $0 \le n < 16$; Block length = 128 bits, $0 \le n < 16$;

Key length = 192 bits, $0 \le n < 24$;

Key length = 256 bits, $0 \le n < 32$.

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 = \sum_{i=0}^{7} b_i x^i \,. \qquad (3.1)$$

For example, {01100011} identifies the specific finite field element $x^6 + x^5 + x + 1$.

It is also convenient to denote byte values using hexadecimal notation with each of two groups of four bits being denoted by a single character as in Fig. 1.

Bit Pattern	Character
0000	0
0001	1
0010	2
0011	3

Bit Pattern	Character
0100	4
0101	5
0110	6
0111	7

Bit Pattern	Character
1000	8
1001	9
1010	a
1011	b

Bit Pattern	Character
1100	c
1101	d
1110	e
1111	f

Figure 1. Hexadecimal representation of bit patterns.

Hence the element {01100011} can be represented as {63}, where the character denoting the four-bit group containing the higher numbered bits is again to the left.

Some finite field operations involve one additional bit (b_8) to the left of an 8-bit byte. Where this extra bit is present, it will appear as '{01}' immediately preceding the 8-bit byte; for example, a 9-bit sequence will be presented as {01}{1b}.

3.3 Arrays of Bytes

Arrays of bytes will be represented in the following form:

$$a_0 a_1 a_2 \ldots a_{15}$$

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence

$$input_0\ input_1\ input_2 \ldots input_{126}\ input_{127}$$

as follows:

$$a_0 = \{input_0, input_1, \ldots, input_7\};$$

$$a_1 = \{input_8, input_9, \ldots, input_{15}\};$$

$$\vdots$$

$$a_{15} = \{input_{120}, input_{121}, \ldots, input_{127}\}.$$

The pattern can be extended to longer sequences (i.e., for 192- and 256-bit keys), so that, in general,

$$a_n = \{input_{8n}, input_{8n+1}, \ldots, input_{8n+7}\}. \qquad (3.2)$$

Taking Sections 3.2 and 3.3 together, Fig. 2 shows how bits within each byte are numbered.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	…
Byte number	0								1								2								…
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	…

Figure 2. Indices for Bytes and Bits.

3.4 The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the **State**. The State consists of four rows of bytes, each containing ***Nb*** bytes, where ***Nb*** is the block length divided by 32. In the State array denoted by the symbol *s*, each individual byte has two indices, with its row number r in the range $0 \le r < 4$ and its column number c in the range $0 \le c < \boldsymbol{Nb}$. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or $s[r,c]$. For this standard, ***Nb***=4, i.e., $0 \le c < 4$ (also see Sec. 6.3).

At the start of the Cipher and Inverse Cipher described in Sec. 5, the input – the array of bytes $in_0, in_1, \ldots in_{15}$ – is copied into the State array as illustrated in Fig. 3. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes $out_0, out_1, \ldots out_{15}$.

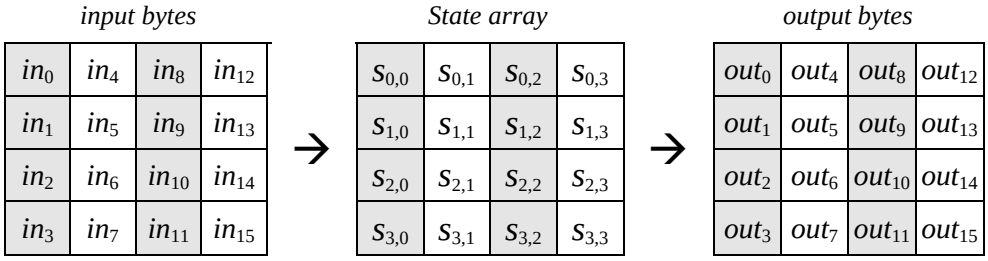


Figure 3. State array input and output.

Hence, at the beginning of the Cipher or Inverse Cipher, the input array, *in*, is copied to the State array according to the scheme:

$$s[r, c] = in[r + 4c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb, \qquad (3.3)$$

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array *out* as follows:

$$out[r + 4c] = s[r, c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb. \tag{3.4}$$

3.5 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit **words**, where the row number r provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), $w_0 ... w_3$, where the column number c provides an index into this array. Hence, for the example in Fig. 3, the State can be considered as an array of four words, as follows:

$$\begin{aligned} w_0 &= s_{0,0}\, s_{1,0}\, s_{2,0}\, s_{3,0} & w_2 &= s_{0,2}\, s_{1,2}\, s_{2,2}\, s_{3,2} \\ w_1 &= s_{0,1}\, s_{1,1}\, s_{2,1}\, s_{3,1} & w_3 &= s_{0,3}\, s_{1,3}\, s_{2,3}\, s_{3,3}\,. \end{aligned} \tag{3.5}$$

4. Mathematical Preliminaries

All bytes in the AES algorithm are interpreted as finite field elements using the notation introduced in Sec. 3.2. Finite field elements can be added and multiplied, but these operations are different from those used for numbers. The following subsections introduce the basic mathematical concepts needed for Sec. 5.

4.1 Addition

The addition of two elements in a finite field is achieved by "adding" the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by $\oplus$) - i.e., modulo 2 - so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, and $0 \oplus 0 = 0$. Consequently, subtraction of polynomials is identical to addition of polynomials.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0\}$ and $\{b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0\}$, the sum is $\{c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0\}$, where each $c_i = a_i \oplus b_i$ (i.e., $c_7 = a_7 \oplus b_7$, $c_6 = a_6 \oplus b_6$, ... $c_0 = a_0 \oplus b_0$).

For example, the following expressions are equivalent to one another:

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation);

{01010111} ⊕ {10000011} = {11010100} (binary notation);

{57} ⊕ {83} = {d4} (hexadecimal notation).

4.2 Multiplication

In the polynomial representation, multiplication in GF(2^8) (denoted by •) corresponds with the multiplication of polynomials modulo an **irreducible polynomial** of degree 8. A polynomial is irreducible if its only divisors are one and itself. **For the AES algorithm, this <u>irreducible</u> polynomial is**

$$m(x) = x^8 + x^4 + x^3 + x + 1, \tag{4.1}$$

or {01}{1b} in hexadecimal notation.

For example, {57} • {83} = {c1}, because

$$(x^6 + x^4 + x^2 + x + 1)\ (x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1 = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

and

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \ \texttt{modulo}\ (x^8 + x^4 + x^3 + x + 1) = x^7 + x^6 + 1.$$

The modular reduction by $m(x)$ ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

The multiplication defined above is associative, and the element {01} is the multiplicative identity. For any non-zero binary polynomial $b(x)$ of degree less than 8, the multiplicative inverse of $b(x)$, denoted $b^{-1}(x)$, can be found as follows: the extended Euclidean algorithm [7] is used to compute polynomials $a(x)$ and $c(x)$ such that

$$b(x)a(x) + m(x)c(x) = 1. \tag{4.2}$$

Hence, $a(x) \bullet b(x) \bmod m(x) = 1$, which means

$$b^{-1}(x) = a(x) \bmod m(x). \tag{4.3}$$

Moreover, for any $a(x)$, $b(x)$ and $c(x)$ in the field, it holds that

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x).$$

It follows that the set of 256 possible byte values, with XOR used as addition and the multiplication defined as above, has the structure of the finite field GF(2^8).

4.2.1 Multiplication by *x*

Multiplying the binary polynomial defined in equation (3.1) with the polynomial x results in

$$b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x. \tag{4.4}$$

The result $x \bullet b(x)$ is obtained by reducing the above result modulo $m(x)$, as defined in equation (4.1). If $b_7 = 0$, the result is already in reduced form. If $b_7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial $m(x)$. It follows that multiplication by x (i.e., {00000010} or {02}) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with {1b}. This operation on bytes is denoted by `xtime()`. Multiplication by higher powers of x can be implemented by repeated application of `xtime()`. By adding intermediate results, multiplication by any constant can be implemented.

For example, {57} • {13} = {fe} because

$$\{57\} \bullet \{02\} = \texttt{xtime}(\{57\}) = \{ae\}$$

$$\{57\} \bullet \{04\} = \texttt{xtime}(\{ae\}) = \{47\}$$

$$\{57\} \bullet \{08\} = \texttt{xtime}(\{47\}) = \{8e\}$$

$$\{57\} \bullet \{10\} = \texttt{xtime}(\{8e\}) = \{07\},$$

thus,

$$\begin{aligned} \{57\} \bullet \{13\} &= \{57\} \bullet (\{01\} \oplus \{02\} \oplus \{10\}) \\ &= \{57\} \oplus \{ae\} \oplus \{07\} \\ &= \{fe\}. \end{aligned}$$

4.3 Polynomials with Coefficients in GF(2^8)

Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \tag{4.5}$$

which will be denoted as a word in the form $[a_0, a_1, a_2, a_3]$. Note that the polynomials in this section behave somewhat differently than the polynomials used in the definition of finite field elements, even though both types of polynomials use the same indeterminate, x. The coefficients in this section are themselves finite field elements, i.e., bytes, instead of bits; also, the multiplication of four-term polynomials uses a different reduction polynomial, defined below. The distinction should always be clear from the context.

To illustrate the addition and multiplication operations, let

$$b(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0 \tag{4.6}$$

define a second four-term polynomial. Addition is performed by adding the finite field coefficients of like powers of x. This addition corresponds to an XOR operation between the corresponding bytes in each of the words – in other words, the XOR of the complete word values.

Thus, using the equations of (4.5) and (4.6),

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0) \tag{4.7}$$

Multiplication is achieved in two steps. In the first step, the polynomial product $c(x) = a(x) \bullet b(x)$ is algebraically expanded, and like powers are collected to give

$$c(x) = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 \tag{4.8}$$

where

$$\begin{aligned} c_0 &= a_0 \bullet b_0 & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\ c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 & c_5 &= a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\ c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 & c_6 &= a_3 \bullet b_3 \end{aligned} \tag{4.9}$$

$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3 .$$

The result, $c(x)$, does not represent a four-byte word. Therefore, the second step of the multiplication is to reduce $c(x)$ modulo a polynomial of degree 4; the result can be reduced to a polynomial of degree less than 4. **For the AES algorithm, this is accomplished with the polynomial $x^4 + 1$**, so that

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4} . \tag{4.10}$$

The modular product of $a(x)$ and $b(x)$, denoted by $a(x) \otimes b(x)$, is given by the four-term polynomial $d(x)$, defined as follows:

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0 \tag{4.11}$$

with

$$\begin{aligned} d_0 &= (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3) \\ d_1 &= (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3) \\ d_2 &= (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3) \\ d_3 &= (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3) \end{aligned} \tag{4.12}$$

When $a(x)$ is a fixed polynomial, the operation defined in equation (4.11) can be written in matrix form as:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{4.13}$$

Because $x^4 + 1$ is not an irreducible polynomial over GF(2^8), multiplication by a fixed four-term polynomial is not necessarily invertible. However, the AES algorithm specifies a fixed four-term polynomial that *does* have an inverse (see Sec. 5.1.3 and Sec. 5.3.3):

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{4.14}$$

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \tag{4.15}$$

Another polynomial used in the AES algorithm (see the **`RotWord()`** function in Sec. 5.2) has $a_0 = a_1 = a_2 = \{00\}$ and $a_3 = \{01\}$, which is the polynomial x^3. Inspection of equation (4.13) above will show that its effect is to form the output word by rotating bytes in the input word. This means that $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.

5. Algorithm Specification

For the AES algorithm**, the length of the input block, the output block and the State is 128 bits.** This is represented by ***Nb*** = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES algorithm**, the length of the Cipher Key, *K*, is 128, 192, or 256 bits.** The key length is represented by ***Nk*** = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by ***Nr***, where ***Nr*** = 10 when ***Nk*** = 4, ***Nr*** = 12 when ***Nk*** = 6, and ***Nr*** = 14 when ***Nk*** = 8.

The only Key-Block-Round combinations that conform to this standard are given in Fig. 4. For implementation issues relating to the key length, block size and number of rounds, see Sec. 6.3.

	Key Length *(Nk words)*	**Block Size** *(Nb words)*	**Number of Rounds** *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 4. Key-Block-Round Combinations.

For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations: 1) byte substitution using a substitution table (S-box), 2) shifting rows of the State array by different offsets, 3) mixing the data within each column of the State array, and 4) adding a Round Key to the State. These transformations (and their inverses) are described in Sec. 5.1.1-5.1.4 and 5.3.1-5.3.4.

The Cipher and Inverse Cipher are described in Sec. 5.1 and Sec. 5.3, respectively, while the Key Schedule is described in Sec. 5.2.

5.1 Cipher

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec. 3.4. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first $\boldsymbol{Nr} - 1$ rounds. The final State is then copied to the output as described in Sec. 3.4.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion routine described in Sec. 5.2.

The Cipher is described in the pseudo code in Fig. 5. The individual transformations - **`SubBytes()`**, **`ShiftRows()`**, **`MixColumns()`**, and **`AddRoundKey()`** – process the State and are described in the following subsections. In Fig. 5, the array **`w[]`** contains the key schedule, which is described in Sec. 5.2.

As shown in Fig. 5, all ***Nr*** rounds are identical with the exception of the final round, which does not include the **`MixColumns()`** transformation.

Appendix B presents an example of the Cipher, showing values for the State array at the beginning of each round and after the application of each of the four transformations described in the following sections.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr–1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 5. Pseudo Code for the Cipher.[1]

5.1.1 SubBytes()Transformation

The **SubBytes()** transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box (Fig. 7), which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field GF(2^8), described in Sec. 4.2; the element {00} is mapped to itself.
2. Apply the following affine transformation (over GF(2)):

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \qquad (5.1)$$

for $0 \le i < 8$, where b_i is the i^{th} bit of the byte, and c_i is the i^{th} bit of a byte c with the value {63} or {01100011}. Here and elsewhere, a prime on a variable (e.g., b') indicates that the variable is to be updated with the value on the right.

In matrix form, the affine transformation element of the S-box can be expressed as:

[1] The various transformations (e.g., **SubBytes()**, **ShiftRows()**, etc.) act upon the State array that is addressed by the 'state' pointer. **AddRoundKey()** uses an additional pointer to address the Round Key.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \qquad (5.2)$$

Figure 6 illustrates the effect of the **SubBytes()** transformation on the State.

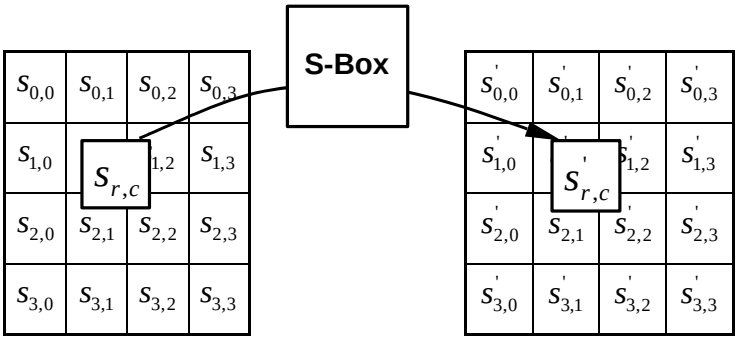


Figure 6. SubBytes() applies the S-box to each byte of the State.

The S-box used in the **SubBytes()** transformation is presented in hexadecimal form in Fig. 7.

For example, if $s_{1,1} = \{53\}$, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Fig. 7. This would result in $s'_{1,1}$ having a value of {ed}.

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

Figure 7. S-box: substitution values for the byte xy (in hexadecimal format).

5.1.2 ShiftRows() Transformation

In the **ShiftRows()** transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted.

Specifically, the **ShiftRows()** transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb)) \bmod Nb} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < \boldsymbol{Nb}, \tag{5.3}$$

where the shift value *shift*(*r*,*Nb*) depends on the row number, *r*, as follows (recall that ***Nb*** = 4):

$$shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3. \tag{5.4}$$

This has the effect of moving bytes to "lower" positions in the row (i.e., lower values of *c* in a given row), while the "lowest" bytes wrap around into the "top" of the row (i.e., higher values of *c* in a given row).

Figure 8 illustrates the **ShiftRows()** transformation.

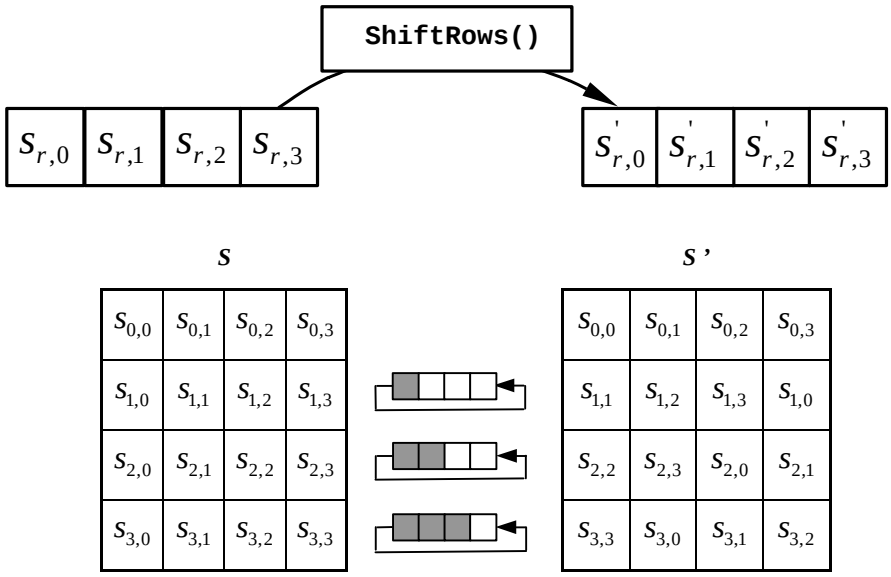


Figure 8. ShiftRows() cyclically shifts the last three rows in the State.

5.1.3 MixColumns() Transformation

The **MixColumns()** transformation operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}. \tag{5.5}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$s'(x) = a(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \mathbf{Nb}. \tag{5.6}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

Figure 9 illustrates the **`MixColumns()`** transformation.

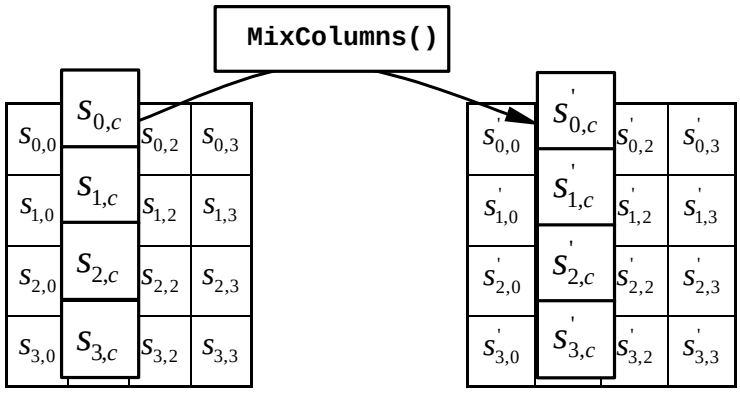


Figure 9. `MixColumns()` operates on the State column-by-column.

5.1.4 AddRoundKey() Transformation

In the **`AddRoundKey()`** transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of ***Nb*** words from the key schedule (described in Sec. 5.2). Those ***Nb*** words are each added into the columns of the State, such that

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad \text{for } 0 \le c < \mathbf{Nb}, \tag{5.7}$$

where $[w_i]$ are the key schedule words described in Sec. 5.2, and *round* is a value in the range $0 \le round \le \mathbf{Nr}$. In the Cipher, the initial Round Key addition occurs when *round* = 0, prior to the first application of the round function (see Fig. 5). The application of the **`AddRoundKey()`** transformation to the ***Nr*** rounds of the Cipher occurs when $1 \le round \le \mathbf{Nr}$.

The action of this transformation is illustrated in Fig. 10, where $l = round * \mathbf{Nb}$. The byte address within words of the key schedule was described in Sec. 3.1.

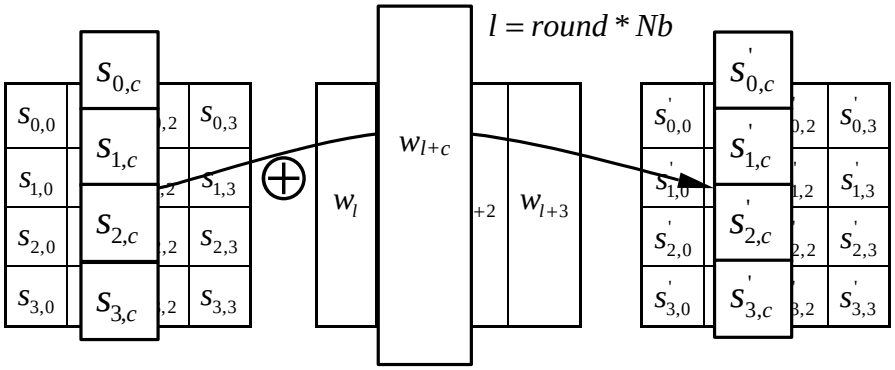


Figure 10. `AddRoundKey()` XORs each column of the State with a word from the key schedule.

5.2 Key Expansion

The AES algorithm takes the Cipher Key, ***K***, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of ***Nb*** (***Nr*** + 1) words: the algorithm requires an initial set of ***Nb*** words, and each of the ***Nr*** rounds requires ***Nb*** words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted $[w_i]$, with i in the range $0 \le i < \mathbf{Nb}(\mathbf{Nr} + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Fig. 11.

`SubWord()` is a function that takes a four-byte input word and applies the S-box (Sec. 5.1.1, Fig. 7) to each of the four bytes to produce an output word. The function **`RotWord()`** takes a word $[a_0,a_1,a_2,a_3]$ as input, performs a cyclic permutation, and returns the word $[a_1,a_2,a_3,a_0]$. The round constant word array, **`Rcon[i]`**, contains the values given by $[x^{i-1},\{00\},\{00\},\{00\}]$, with x^{i-1} being powers of x (x is denoted as {02}) in the field GF(2^8), as discussed in Sec. 4.2 (note that i starts at 1, not 0).

From Fig. 11, it can be seen that the first ***Nk*** words of the expanded key are filled with the Cipher Key. Every following word, **`w[i]`**, is equal to the XOR of the previous word, **`w[i-1]`**, and the word ***Nk*** positions earlier, **`w[i-`*`Nk`*`]`**. For words in positions that are a multiple of ***Nk***, a transformation is applied to **`w[i-1]`** prior to the XOR, followed by an XOR with a round constant, **`Rcon[i]`**. This transformation consists of a cyclic shift of the bytes in a word (**`RotWord()`**), followed by the application of a table lookup to all four bytes of the word (**`SubWord()`**).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys (***Nk*** = 8) is slightly different than for 128- and 192-bit Cipher Keys. If ***Nk*** = 8 and `i-4` is a multiple of ***Nk***, then **`SubWord()`** is applied to **`w[i-1]`** prior to the XOR.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Note that *Nk*=4, 6, and 8 do not all have to be implemented; they are all included in the conditional statement above for conciseness. Specific implementation requirements for the Cipher Key are presented in Sec. 6.1.

Figure 11. Pseudo Code for Key Expansion.[2]

Appendix A presents examples of the Key Expansion.

5.3 Inverse Cipher

The Cipher transformations in Sec. 5.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - **`InvShiftRows()`**, **`InvSubBytes()`**,**`InvMixColumns()`**, and **`AddRoundKey()`** – process the State and are described in the following subsections.

The Inverse Cipher is described in the pseudo code in Fig. 12. In Fig. 12, the array **`w[]`** contains the key schedule, which was described previously in Sec. 5.2.

[2] The functions **`SubWord()`** and **`RotWord()`** return a result that is a transformation of the function input, whereas the transformations in the Cipher and Inverse Cipher (e.g., **`ShiftRows()`**, **`SubBytes()`**, etc.) transform the State array that is addressed by the '`state`' pointer.

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)                    // See Sec. 5.3.1
      InvSubBytes(state)                     // See Sec. 5.3.2
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)                   // See Sec. 5.3.3
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

Figure 12. Pseudo Code for the Inverse Cipher.[3]

5.3.1 InvShiftRows() Transformation

InvShiftRows() is the inverse of the **ShiftRows()** transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. The bottom three rows are cyclically shifted by $\boldsymbol{Nb} - shift(r, Nb)$ bytes, where the shift value *shift(r,Nb)* depends on the row number, and is given in equation (5.4) (see Sec. 5.1.2).

Specifically, the **InvShiftRows()** transformation proceeds as follows:

$$s'_{r,(c+shift(r,Nb)) \bmod Nb} = s_{r,c} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < \boldsymbol{Nb} \tag{5.8}$$

Figure 13 illustrates the **InvShiftRows()** transformation.

[3] The various transformations (e.g., **InvSubBytes()**, **InvShiftRows()**, etc.) act upon the State array that is addressed by the '`state`' pointer. **AddRoundKey()** uses an additional pointer to address the Round Key.

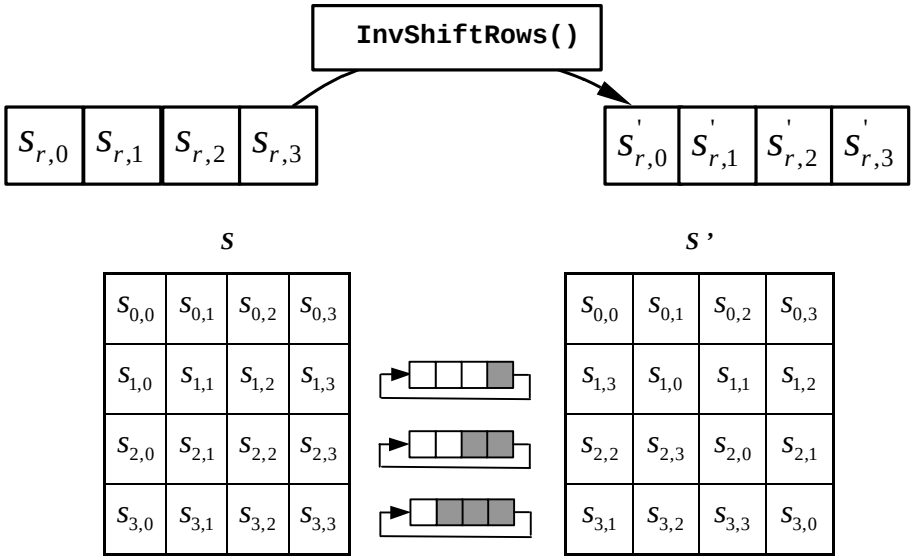


Figure 13. `InvShiftRows()` cyclically shifts the last three rows in the State.

5.3.2 `InvSubBytes()` Transformation

`InvSubBytes()` is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (5.1) followed by taking the multiplicative inverse in GF(2^8).

The inverse S-box used in the **`InvSubBytes()`** transformation is presented in Fig. 14:

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

Figure 14. Inverse S-box: substitution values for the byte xy (in hexadecimal format).

5.3.3 InvMixColumns() Transformation

InvMixColumns() is the inverse of the **MixColumns()** transformation. **InvMixColumns()** operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \qquad (5.9)$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$s'(x) = a^{-1}(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}. \qquad (5.10)$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

5.3.4 Inverse of the AddRoundKey() Transformation

AddRoundKey(), which was described in Sec. 5.1.4, is its own inverse, since it only involves an application of the XOR operation.

5.3.5 Equivalent Inverse Cipher

In the straightforward Inverse Cipher presented in Sec. 5.3 and Fig. 12, the sequence of the transformations differs from that of the Cipher, while the form of the key schedules for encryption and decryption remains the same. However, several properties of the AES algorithm allow for an Equivalent Inverse Cipher that has the same sequence of transformations as the Cipher (with the transformations replaced by their inverses). This is accomplished with a change in the key schedule.

The two properties that allow for this Equivalent Inverse Cipher are as follows:

1. The **SubBytes()** and **ShiftRows()** transformations commute; that is, a **SubBytes()** transformation immediately followed by a **ShiftRows()** transformation is equivalent to a **ShiftRows()** transformation immediately followed buy a **SubBytes()** transformation. The same is true for their inverses, **InvSubBytes()** and **InvShiftRows.**

2. The column mixing operations - **MixColumns()** and **InvMixColumns()** - are linear with respect to the column input, which means

```
InvMixColumns(state XOR Round Key) =
                InvMixColumns(state) XOR InvMixColumns(Round Key).
```

These properties allow the order of **InvSubBytes()** and **InvShiftRows()** transformations to be reversed. The order of the **AddRoundKey()** and **InvMixColumns()** transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the **InvMixColumns()** transformation.

The equivalent inverse cipher is defined by reversing the order of the **InvSubBytes()** and **InvShiftRows()** transformations shown in Fig. 12, and by reversing the order of the **AddRoundKey()** and **InvMixColumns()** transformations used in the "round loop" after first modifying the decryption key schedule for *round* = 1 to *Nr*-1 using the **InvMixColumns()** transformation. The first and last *Nb* words of the decryption key schedule shall *not* be modified in this manner.

Given these changes, the resulting Equivalent Inverse Cipher offers a more efficient structure than the Inverse Cipher described in Sec. 5.3 and Fig. 12. Pseudo code for the Equivalent Inverse Cipher appears in Fig. 15. (The word array **dw[]** contains the modified decryption key schedule. The modification to the Key Expansion routine is also provided in Fig. 15.)

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])

   for round = Nr-1 step -1 downto 1
      InvSubBytes(state)
      InvShiftRows(state)
      InvMixColumns(state)
      AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
   end for

   InvSubBytes(state)
   InvShiftRows(state)
   AddRoundKey(state, dw[0, Nb-1])

   out = state
end
```

For the Equivalent Inverse Cipher, the following pseudo code is added at the end of the Key Expansion routine (Sec. 5.2):

```
   for i = 0 step 1 to (Nr+1)*Nb-1
      dw[i] = w[i]
   end for


   for round = 1 step 1 to Nr-1
      InvMixColumns(dw[round*Nb, (round+1)*Nb-1])     //   note   change   of
type
   end for
```

Note that, since InvMixColumns operates on a two-dimensional array of bytes while the Round Keys are held in an array of words, the call to InvMixColumns in this code sequence involves a change of type (i.e. the input to InvMixColumns() is normally the State array, which is considered to be a two-dimensional array of bytes, whereas the input here is a Round Key computed as a one-dimensional array of words).

Figure 15. Pseudo Code for the Equivalent Inverse Cipher.

6. Implementation Issues

6.1 Key Length Requirements

An implementation of the AES algorithm shall support *at least one* of the three key lengths specified in Sec. 5: 128, 192, or 256 bits (i.e., ***Nk*** = 4, 6, or 8, respectively). Implementations

may optionally support two or three key lengths, which may promote the interoperability of algorithm implementations.

6.2 Keying Restrictions

No weak or semi-weak keys have been identified for the AES algorithm, and there is no restriction on key selection.

6.3 Parameterization of Key Length, Block Size, and Round Number

This standard explicitly defines the allowed values for the key length (***Nk***), block size (***Nb***), and number of rounds (***Nr***) – see Fig. 4. However, future reaffirmations of this standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind.

6.4 Implementation Suggestions Regarding Various Platforms

Implementation variations are possible that may, in many cases, offer performance or other advantages. Given the same input key and data (plaintext or ciphertext), any implementation that produces the same output (ciphertext or plaintext) as the algorithm specified in this standard is an acceptable implementation of the AES.

Reference [3] and other papers located at Ref. [1] include suggestions on how to efficiently implement the AES algorithm on a variety of platforms.

Appendix A - Key Expansion Examples

This appendix shows the development of the key schedule for various key sizes. Note that multi-byte values are presented using the notation described in Sec. 3. The intermediate values produced during the development of the key schedule (see Sec. 5.2) are given in the following table (all values are in hexadecimal format, with the exception of the index column (i)).

A.1 Expansion of a 128-bit Cipher Key

This section contains the key expansion of the following cipher key:

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

for ***Nk*** = 4, which results in

w_0 = **2b7e1516** w_1 = **28aed2a6** w_2 = **abf71588** w_3 = **09cf4f3c**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc

24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

A.2 Expansion of a 192-bit Cipher Key

This section contains the key expansion of the following cipher key:

Cipher Key = 8e 73 b0 f7 da 0e 64 52 c8 10 f3 2b
80 90 79 e5 62 f8 ea d2 52 2c 6b 7b

for ***Nk*** = 6, which results in

w_0 = **8e73b0f7** w_1 = **da0e6452** w_2 = **c810f32b** w_3 = **809079e5**

w_4 = **62f8ead2** w_5 = **522c6b7b**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e

9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd
13	4db7b4bd					2402f5a5	69b54118
14	69b54118					ec12068e	85a74796
15	85a74796					6c827f6b	e92538fd
16	e92538fd					0e7a95b9	e75fad44
17	e75fad44					5c56fec2	bb095386
18	bb095386	095386bb	01ed44ea	04000000	05ed44ea	4db7b4bd	485af057
19	485af057					69b54118	21efb14f
20	21efb14f					85a74796	a448f6d9
21	a448f6d9					e92538fd	4d6dce24
22	4d6dce24					e75fad44	aa326360
23	aa326360					bb095386	113b30e6
24	113b30e6	3b30e611	e2048e82	08000000	ea048e82	485af057	a25e7ed5
25	a25e7ed5					21efb14f	83b1cf9a
26	83b1cf9a					a448f6d9	27f93943
27	27f93943					4d6dce24	6a94f767
28	6a94f767					aa326360	c0a69407
29	c0a69407					113b30e6	d19da4e1
30	d19da4e1	9da4e1d1	5e49f83e	10000000	4e49f83e	a25e7ed5	ec1786eb
31	ec1786eb					83b1cf9a	6fa64971
32	6fa64971					27f93943	485f7032
33	485f7032					6a94f767	22cb8755
34	22cb8755					c0a69407	e26d1352
35	e26d1352					d19da4e1	33f0b7b3
36	33f0b7b3	f0b7b333	8ca96dc3	20000000	aca96dc3	ec1786eb	40beeb28
37	40beeb28					6fa64971	2f18a259
38	2f18a259					485f7032	6747d26b
39	6747d26b					22cb8755	458c553e
40	458c553e					e26d1352	a7e1466c
41	a7e1466c					33f0b7b3	9411f1df
42	9411f1df	11f1df94	82a19e22	40000000	c2a19e22	40beeb28	821f750a
43	821f750a					2f18a259	ad07d753

44	ad07d753					6747d26b	ca400538
45	ca400538					458c553e	8fcc5006
46	8fcc5006					a7e1466c	282d166a
47	282d166a					9411f1df	bc3ce7b5
48	bc3ce7b5	3ce7b5bc	eb94d565	80000000	6b94d565	821f750a	e98ba06f
49	e98ba06f					ad07d753	448c773c
50	448c773c					ca400538	8ecc7204
51	8ecc7204					8fcc5006	01002202

A.3 Expansion of a 256-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =      60 3d eb 10 15 ca 71 be 2b 73 ae f0 85 7d 77 81
                  1f 35 2c 07 3b 61 08 d7 2d 98 10 a3 09 14 df f4
```

for ***Nk*** = 8, which results in

w_0 = **603deb10** w_1 = **15ca71be** w_2 = **2b73aef0** w_3 = **857d7781**

w_4 = **1f352c07** w_5 = **3b6108d7** w_6 = **2d9810a3** w_7 = **0914dff4**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8
17	d59aecb8					8e6925af	5bf3c917
18	5bf3c917					a51a8b5f	fee94248
19	fee94248					2067fcde	de8ebe96
20	de8ebe96		1d19ae90			a8b09c1a	b5a9328a
21	b5a9328a					93d194cd	2678a647
22	2678a647					be49846e	98312229

23	98312229					b75d5b9a	2f6c79b3
24	2f6c79b3	6c79b32f	50b66d15	04000000	54b66d15	d59aecb8	812c81ad
25	812c81ad					5bf3c917	dadf48ba
26	dadf48ba					fee94248	24360af2
27	24360af2					de8ebe96	fab8b464
28	fab8b464		2d6c8d43			b5a9328a	98c5bfc9
29	98c5bfc9					2678a647	bebd198e
30	bebd198e					98312229	268c3ba7
31	268c3ba7					2f6c79b3	09e04214
32	09e04214	e0421409	e12cfa01	08000000	e92cfa01	812c81ad	68007bac
33	68007bac					dadf48ba	b2df3316
34	b2df3316					24360af2	96e939e4
35	96e939e4					fab8b464	6c518d80
36	6c518d80		50d15dcd			98c5bfc9	c814e204
37	c814e204					bebd198e	76a9fb8a
38	76a9fb8a					268c3ba7	5025c02d
39	5025c02d					09e04214	59c58239
40	59c58239	c5823959	a61312cb	10000000	b61312cb	68007bac	de136967
41	de136967					b2df3316	6ccc5a71
42	6ccc5a71					96e939e4	fa256395
43	fa256395					6c518d80	9674ee15
44	9674ee15		90922859			c814e204	5886ca5d
45	5886ca5d					76a9fb8a	2e2f31d7
46	2e2f31d7					5025c02d	7e0af1fa
47	7e0af1fa					59c58239	27cf73c3
48	27cf73c3	cf73c327	8a8f2ecc	20000000	aa8f2ecc	de136967	749c47ab
49	749c47ab					6ccc5a71	18501dda
50	18501dda					fa256395	e2757e4f
51	e2757e4f					9674ee15	7401905a
52	7401905a		927c60be			5886ca5d	cafaaae3
53	cafaaae3					2e2f31d7	e4d59b34
54	e4d59b34					7e0af1fa	9adf6ace
55	9adf6ace					27cf73c3	bd10190d
56	bd10190d	10190dbd	cad4d77a	40000000	8ad4d77a	749c47ab	fe4890d1
57	fe4890d1					18501dda	e6188d0b

58	e6188d0b					e2757e4f	046df344
59	046df344					7401905a	706c631e

Appendix B – Cipher Example

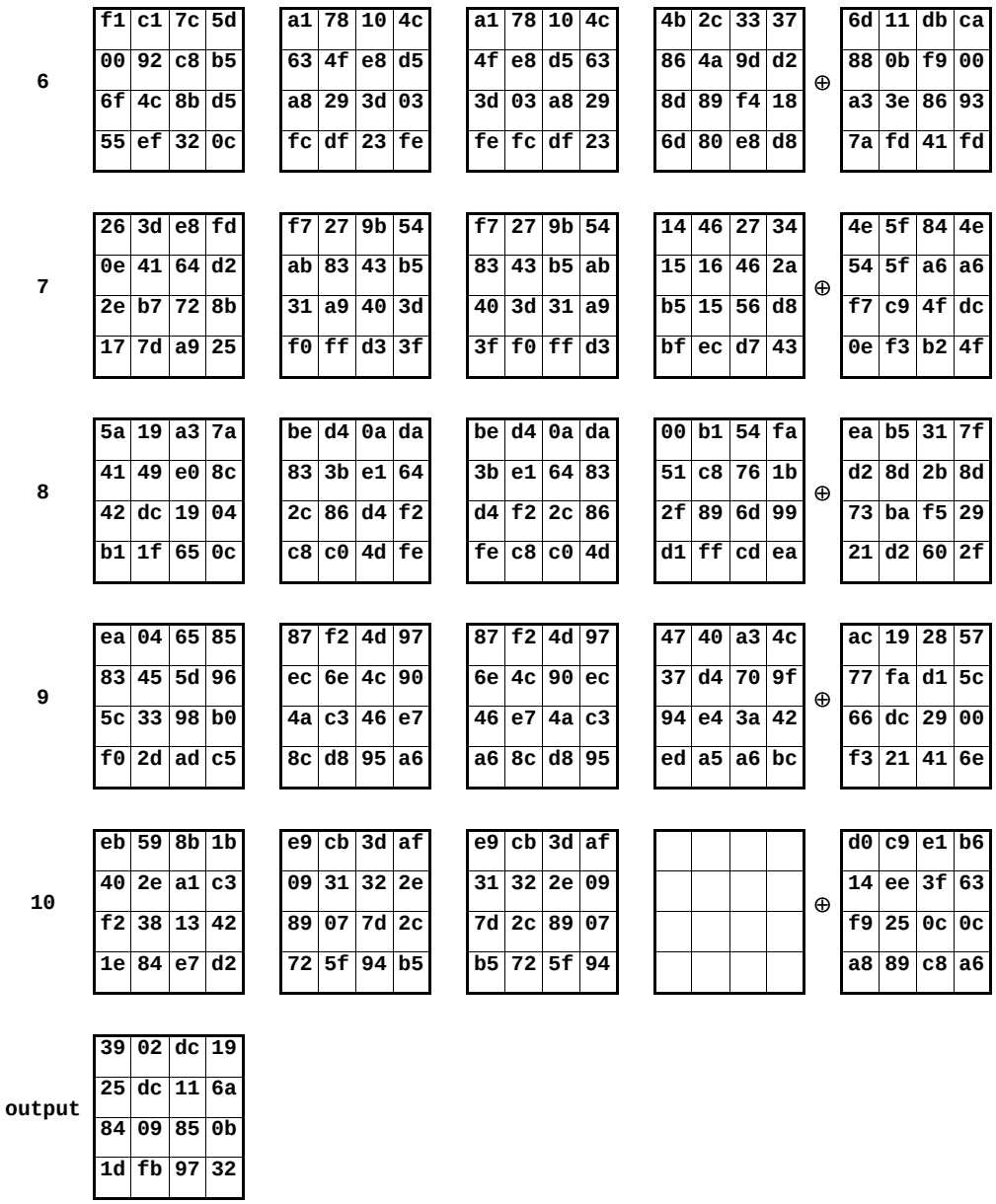
The following diagram shows the values in the State array as the Cipher progresses for a block length and a Cipher Key length of 16 bytes each (i.e., *Nb* = 4 and *Nk* = 4).

Input = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

The Round Key values are taken from the Key Expansion example in Appendix A.

Round Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns		Round Key Value	
input	32 88 31 e0 43 5a 31 37 f6 30 98 07 a8 8d a2 34				⊕	2b 28 ab 09 7e ae f7 cf 15 d2 15 4f 16 a6 88 3c	=
1	19 a0 9a e9 3d f4 c6 f8 e3 e2 8d 48 be 2b 2a 08	d4 e0 b8 1e 27 bf b4 41 11 98 5d 52 ae f1 e5 30	d4 e0 b8 1e bf b4 41 27 5d 52 11 98 30 ae f1 e5	04 e0 48 28 66 cb f8 06 81 19 d3 26 e5 9a 7a 4c	⊕	a0 88 23 2a fa 54 a3 6c fe 2c 39 76 17 b1 39 05	=
2	a4 68 6b 02 9c 9f 5b 6a 7f 35 ea 50 f2 2b 43 49	49 45 7f 77 de db 39 02 d2 96 87 53 89 f1 1a 3b	49 45 7f 77 db 39 02 de 87 53 d2 96 3b 89 f1 1a	58 1b db 1b 4d 4b e7 6b ca 5a ca b0 f1 ac a8 e5	⊕	f2 7a 59 73 c2 96 35 59 95 b9 80 f6 f2 43 7a 7f	=
3	aa 61 82 68 8f dd d2 32 5f e3 4a 46 03 ef d2 9a	ac ef 13 45 73 c1 b5 23 cf 11 d6 5a 7b df b5 b8	ac ef 13 45 c1 b5 23 73 d6 5a cf 11 b8 7b df b5	75 20 53 bb ec 0b c0 25 09 63 cf d0 93 33 7c dc	⊕	3d 47 1e 6d 80 16 23 7a 47 fe 7e 88 7d 3e 44 3b	=
4	48 67 4d d6 6c 1d e3 5f 4e 9d b1 58 ee 0d 38 e7	52 85 e3 f6 50 a4 11 cf 2f 5e c8 6a 28 d7 07 94	52 85 e3 f6 a4 11 cf 50 c8 6a 2f 5e 94 28 d7 07	0f 60 6f 5e d6 31 c0 b3 da 38 10 13 a9 bf 6b 01	⊕	ef a8 b6 db 44 52 71 0b a5 5b 25 ad 41 7f 3b 00	=
5	e0 c8 d9 85 92 63 b1 b8 7f 63 35 be e8 c0 50 01	e1 e8 35 97 4f fb c8 6c d2 fb 96 ae 9b ba 53 7c	e1 e8 35 97 fb c8 6c 4f 96 ae d2 fb 7c 9b ba 53	25 bd b6 4c d1 11 3a 4c a9 d1 33 c0 ad 68 8e b0	⊕	d4 7c ca 11 d1 83 f2 f9 c6 9d b8 15 f8 87 bc bc	=


6
f1 c1 7c 5d
00 92 c8 b5
6f 4c 8b d5
55 ef 32 0c
a1 78 10 4c
63 4f e8 d5
a8 29 3d 03
fc df 23 fe
a1 78 10 4c
4f e8 d5 63
3d 03 a8 29
fe fc df 23
4b 2c 33 37
86 4a 9d d2
8d 89 f4 18
6d 80 e8 d8
⊕
6d 11 db ca
88 0b f9 00
a3 3e 86 93
7a fd 41 fd
=
7
26 3d e8 fd
0e 41 64 d2
2e b7 72 8b
17 7d a9 25
f7 27 9b 54
ab 83 43 b5
31 a9 40 3d
f0 ff d3 3f
f7 27 9b 54
83 43 b5 ab
40 3d 31 a9
3f f0 ff d3
14 46 27 34
15 16 46 2a
b5 15 56 d8
bf ec d7 43
⊕
4e 5f 84 4e
54 5f a6 a6
f7 c9 4f dc
0e f3 b2 4f
=
8
5a 19 a3 7a
41 49 e0 8c
42 dc 19 04
b1 1f 65 0c
be d4 0a da
83 3b e1 64
2c 86 d4 f2
c8 c0 4d fe
be d4 0a da
3b e1 64 83
d4 f2 2c 86
fe c8 c0 4d
00 b1 54 fa
51 c8 76 1b
2f 89 6d 99
d1 ff cd ea
⊕
ea b5 31 7f
d2 8d 2b 8d
73 ba f5 29
21 d2 60 2f
=
9
ea 04 65 85
83 45 5d 96
5c 33 98 b0
f0 2d ad c5
87 f2 4d 97
ec 6e 4c 90
4a c3 46 e7
8c d8 95 a6
87 f2 4d 97
6e 4c 90 ec
46 e7 4a c3
a6 8c d8 95
47 40 a3 4c
37 d4 70 9f
94 e4 3a 42
ed a5 a6 bc
⊕
ac 19 28 57
77 fa d1 5c
66 dc 29 00
f3 21 41 6e
=
10
eb 59 8b 1b
40 2e a1 c3
f2 38 13 42
1e 84 e7 d2
e9 cb 3d af
09 31 32 2e
89 07 7d 2c
72 5f 94 b5
e9 cb 3d af
31 32 2e 09
7d 2c 89 07
b5 72 5f 94
⊕
d0 c9 e1 b6
14 ee 3f 63
f9 25 0c 0c
a8 89 c8 a6
=
output
39 02 dc 19
25 dc 11 6a
84 09 85 0b
1d fb 97 32

Appendix C – Example Vectors

This appendix contains example vectors, including intermediate values – for all three AES key lengths (***Nk*** = 4, 6, and 8), for the Cipher, Inverse Cipher, and Equivalent Inverse Cipher that are described in Sec. 5.1, 5.3, and 5.3.5, respectively. Additional examples may be found at [1] and [5].

All vectors are in hexadecimal notation, with each pair of characters giving a byte value in which the left character of each pair provides the bit pattern for the 4 bit group containing the higher numbered bits using the notation explained in Sec. 3.2, while the right character provides the bit pattern for the lower-numbered bits. The array index for all bytes (groups of two hexadecimal digits) within these test vectors starts at zero and increases from left to right.

```
Legend for CIPHER (ENCRYPT) (round number r = 0 to 10, 12 or 14):

   input:   cipher input
   start:   state at start of round[r]
   s_box:   state after SubBytes()
   s_row:   state after ShiftRows()
   m_col:   state after MixColumns()
   k_sch:   key schedule value for round[r]
   output:  cipher output


Legend for INVERSE CIPHER (DECRYPT) (round number r = 0 to 10, 12 or 14):
   iinput:  inverse cipher input
   istart:  state at start of round[r]
   is_box:  state after InvSubBytes()
   is_row:  state after InvShiftRows()
   ik_sch:  key schedule value for round[r]
   ik_add:  state after AddRoundKey()
   ioutput: inverse cipher output


Legend for EQUIVALENT INVERSE CIPHER (DECRYPT) (round number r = 0 to 10, 12
   or 14):

   iinput:  inverse cipher input
   istart:  state at start of round[r]
   is_box:  state after InvSubBytes()
   is_row:  state after InvShiftRows()
   im_col:  state after InvMixColumns()
   ik_sch:  key schedule value for round[r]
   ioutput: inverse cipher output
```

C.1 AES-128 (*Nk*=4, *Nr*=10)

```
PLAINTEXT:          00112233445566778899aabbccddeeff
KEY:                000102030405060708090a0b0c0d0e0f

CIPHER (ENCRYPT):
```

```
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 2].start    89d810e8855ace682d1843d8cb128fe4
round[ 2].s_box    a761ca9b97be8b45d8ad1a611fc97369
round[ 2].s_row    a7be1a6997ad739bd8c9ca451f618b61
round[ 2].m_col    ff87968431d86a51645151fa773ad009
round[ 2].k_sch    b692cf0b643dbdf1be9bc5006830b3fe
round[ 3].start    4915598f55e5d7a0daca94fa1f0a63f7
round[ 3].s_box    3b59cb73fcd90ee05774222dc067fb68
round[ 3].s_row    3bd92268fc74fb735767cbe0c0590e2d
round[ 3].m_col    4c9c1e66f771f0762c3f868e534df256
round[ 3].k_sch    b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 4].start    fa636a2825b339c940668a3157244d17
round[ 4].s_box    2dfb02343f6d12dd09337ec75b36e3f0
round[ 4].s_row    2d6d7ef03f33e334093602dd5bfb12c7
round[ 4].m_col    6385b79ffc538df997be478e7547d691
round[ 4].k_sch    47f7f7bc95353e03f96c32bcfd058dfd
round[ 5].start    247240236966b3fa6ed2753288425b6c
round[ 5].s_box    36400926f9336d2d9fb59d23c42c3950
round[ 5].s_row    36339d50f9b539269f2c092dc4406d23
round[ 5].m_col    f4bcd45432e554d075f1d6c51dd03b3c
round[ 5].k_sch    3caaa3e8a99f9deb50f3af57adf622aa
round[ 6].start    c81677bc9b7ac93b25027992b0261996
round[ 6].s_box    e847f56514dadde23f77b64fe7f7d490
round[ 6].s_row    e8dab6901477d4653ff7f5e2e747dd4f
round[ 6].m_col    9816ee7400f87f556b2c049c8e5ad036
round[ 6].k_sch    5e390f7df7a69296a7553dc10aa31f6b
round[ 7].start    c62fe109f75eedc3cc79395d84f9cf5d
round[ 7].s_box    b415f8016858552e4bb6124c5f998a4c
round[ 7].s_row    b458124c68b68a014b99f82e5f15554c
round[ 7].m_col    c57e1c159a9bd286f05f4be098c63439
round[ 7].k_sch    14f9701ae35fe28c440adf4d4ea9c026
round[ 8].start    d1876c0f79c4300ab45594add66ff41f
round[ 8].s_box    3e175076b61c04678dfc2295f6a8bfc0
round[ 8].s_row    3e1c22c0b6fcbf768da85067f6170495
round[ 8].m_col    baa03de7a1f9b56ed5512cba5f414d23
round[ 8].k_sch    47438735a41c65b9e016baf4aebf7ad2
round[ 9].start    fde3bad205e5d0d73547964ef1fe37f1
round[ 9].s_box    5411f4b56bd9700e96a0902fa1bb9aa1
round[ 9].s_row    54d990a16ba09ab596bbf40ea111702f
round[ 9].m_col    e9f74eec023020f61bf2ccf2353c21c7
round[ 9].k_sch    549932d1f08557681093ed9cbe2c974e
round[10].start    bd6e7c3df2b5779e0b61216e8b10b689
round[10].s_box    7a9f102789d5f50b2beffd9f3dca4ea7
round[10].s_row    7ad5fda789ef4e272bca100b3d9ff59f
round[10].k_sch    13111d7fe3944a17f307a78b4d2b30c5
round[10].output   69c4e0d86a7b0430d8cdb78070b4c55a

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch   13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart   7ad5fda789ef4e272bca100b3d9ff59f
```

```
round[ 1].is_row   7a9f102789d5f50b2beffd9f3dca4ea7
round[ 1].is_box   bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].ik_sch   549932d1f08557681093ed9cbe2c974e
round[ 1].ik_add   e9f74eec023020f61bf2ccf2353c21c7
round[ 2].istart   54d990a16ba09ab596bbf40ea111702f
round[ 2].is_row   5411f4b56bd9700e96a0902fa1bb9aa1
round[ 2].is_box   fde3bad205e5d0d73547964ef1fe37f1
round[ 2].ik_sch   47438735a41c65b9e016baf4aebf7ad2
round[ 2].ik_add   baa03de7a1f9b56ed5512cba5f414d23
round[ 3].istart   3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_row   3e175076b61c04678dfc2295f6a8bfc0
round[ 3].is_box   d1876c0f79c4300ab45594add66ff41f
round[ 3].ik_sch   14f9701ae35fe28c440adf4d4ea9c026
round[ 3].ik_add   c57e1c159a9bd286f05f4be098c63439
round[ 4].istart   b458124c68b68a014b99f82e5f15554c
round[ 4].is_row   b415f8016858552e4bb6124c5f998a4c
round[ 4].is_box   c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].ik_sch   5e390f7df7a69296a7553dc10aa31f6b
round[ 4].ik_add   9816ee7400f87f556b2c049c8e5ad036
round[ 5].istart   e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_row   e847f56514dadde23f77b64fe7f7d490
round[ 5].is_box   c81677bc9b7ac93b25027992b0261996
round[ 5].ik_sch   3caaa3e8a99f9deb50f3af57adf622aa
round[ 5].ik_add   f4bcd45432e554d075f1d6c51dd03b3c
round[ 6].istart   36339d50f9b539269f2c092dc4406d23
round[ 6].is_row   36400926f9336d2d9fb59d23c42c3950
round[ 6].is_box   247240236966b3fa6ed2753288425b6c
round[ 6].ik_sch   47f7f7bc95353e03f96c32bcfd058dfd
round[ 6].ik_add   6385b79ffc538df997be478e7547d691
round[ 7].istart   2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_row   2dfb02343f6d12dd09337ec75b36e3f0
round[ 7].is_box   fa636a2825b339c940668a3157244d17
round[ 7].ik_sch   b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 7].ik_add   4c9c1e66f771f0762c3f868e534df256
round[ 8].istart   3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_row   3b59cb73fcd90ee05774222dc067fb68
round[ 8].is_box   4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].ik_sch   b692cf0b643dbdf1be9bc5006830b3fe
round[ 8].ik_add   ff87968431d86a51645151fa773ad009
round[ 9].istart   a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_row   a761ca9b97be8b45d8ad1a611fc97369
round[ 9].is_box   89d810e8855ace682d1843d8cb128fe4
round[ 9].ik_sch   d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 9].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[10].istart   6353e08c0960e104cd70b751bacad0e7
round[10].is_row   63cab7040953d051cd60e0e7ba70e18c
round[10].is_box   00102030405060708090a0b0c0d0e0f0
round[10].ik_sch   000102030405060708090a0b0c0d0e0f
round[10].ioutput  00112233445566778899aabbccddeeff

EQUIVALENT INVERSE CIPHER (DECRYPT):
round[ 0].iinput   69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch   13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart   7ad5fda789ef4e272bca100b3d9ff59f
round[ 1].is_box   bdb52189f261b63d0b107c9e8b6e776e
round[ 1].is_row   bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].im_col   4773b91ff72f354361cb018ea1e6cf2c
```

```
round[ 1].ik_sch   13aa29be9c8faff6f770f58000f7bf03
round[ 2].istart   54d990a16ba09ab596bbf40ea111702f
round[ 2].is_box   fde596f1054737d235febad7f1e3d04e
round[ 2].is_row   fde3bad205e5d0d73547964ef1fe37f1
round[ 2].im_col   2d7e86a339d9393ee6570a1101904e16
round[ 2].ik_sch   1362a4638f2586486bff5a76f7874a83
round[ 3].istart   3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_box   d1c4941f7955f40fb46f6c0ad68730ad
round[ 3].is_row   d1876c0f79c4300ab45594add66ff41f
round[ 3].im_col   39daee38f4f1a82aaf432410c36d45b9
round[ 3].ik_sch   8d82fc749c47222be4dadc3e9c7810f5
round[ 4].istart   b458124c68b68a014b99f82e5f15554c
round[ 4].is_box   c65e395df779cf09ccf9e1c3842fed5d
round[ 4].is_row   c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].im_col   9a39bf1d05b20a3a476a0bf79fe51184
round[ 4].ik_sch   72e3098d11c5de5f789dfe1578a2cccb
round[ 5].istart   e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_box   c87a79969b0219bc2526773bb016c992
round[ 5].is_row   c81677bc9b7ac93b25027992b0261996
round[ 5].im_col   18f78d779a93eef4f6742967c47f5ffd
round[ 5].ik_sch   2ec410276326d7d26958204a003f32de
round[ 6].istart   36339d50f9b539269f2c092dc4406d23
round[ 6].is_box   2466756c69d25b236e4240fa8872b332
round[ 6].is_row   247240236966b3fa6ed2753288425b6c
round[ 6].im_col   85cf8bf472d124c10348f545329c0053
round[ 6].ik_sch   a8a2f5044de2c7f50a7ef79869671294
round[ 7].istart   2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_box   fab38a1725664d2840246ac957633931
round[ 7].is_row   fa636a2825b339c940668a3157244d17
round[ 7].im_col   fc1fc1f91934c98210fbfb8da340eb21
round[ 7].ik_sch   c7c6e391e54032f1479c306d6319e50c
round[ 8].istart   3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_box   49e594f755ca638fda0a59a01f15d7fa
round[ 8].is_row   4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].im_col   076518f0b52ba2fb7a15c8d93be45e00
round[ 8].ik_sch   a0db02992286d160a2dc029c2485d561
round[ 9].istart   a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_box   895a43e485188fe82d121068cbd8ced8
round[ 9].is_row   89d810e8855ace682d1843d8cb128fe4
round[ 9].im_col   ef053f7c8b3d32fd4d2a64ad3c93071a
round[ 9].ik_sch   8c56dff0825dd3f9805ad3fc8659d7fd
round[10].istart   6353e08c0960e104cd70b751bacad0e7
round[10].is_box   0050a0f04090e03080d02070c01060b0
round[10].is_row   00102030405060708090a0b0c0d0e0f0
round[10].ik_sch   000102030405060708090a0b0c0d0e0f
round[10].ioutput  00112233445566778899aabbccddeeff
```

C.2 AES-192 (*Nk*=6, *Nr*=12)

```
PLAINTEXT:  00112233445566778899aabbccddeeff
KEY:        000102030405060708090a0b0c0d0e0f1011121314151617

CIPHER (ENCRYPT):
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
```

```
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    101112131415161758462f95c43f4fe
round[ 2].start    4f63760643e0aa85aff8c9d041fa0de4
round[ 2].s_box    84fb386f1ae1ac977941dd70832dd769
round[ 2].s_row    84e1dd691a41d76f792d389783fbac70
round[ 2].m_col    9f487f794f955f662afc86abd7f1ab29
round[ 2].k_sch    544afef55847f0fa4856e2e95c43f4fe
round[ 3].start    cb02818c17d2af9c62aa64428bb25fd7
round[ 3].s_box    1f770c64f0b579deaaac432c3d37cf0e
round[ 3].s_row    1fb5430ef0accf64aa370cde3d77792c
round[ 3].m_col    b7a53ecbbf9d75a0c40efc79b674cc11
round[ 3].k_sch    40f949b31cbabd4d48f043b810b7b342
round[ 4].start    f75c7778a327c8ed8cfebfc1a6c37f53
round[ 4].s_box    684af5bc0acce85564bb0878242ed2ed
round[ 4].s_row    68cc08ed0abbd2bc642ef555244ae878
round[ 4].m_col    7a1e98bdacb6d1141a6944dd06eb2d3e
round[ 4].k_sch    58e151ab04a2a5557effb5416245080c
round[ 5].start    22ffc916a81474416496f19c64ae2532
round[ 5].s_box    9316dd47c2fa92834390a1de43e43f23
round[ 5].s_row    93faa123c2903f4743e4dd83431692de
round[ 5].m_col    aaa755b34cffe57cef6f98e1f01c13e6
round[ 5].k_sch    2ab54bb43a02f8f662e3a95d66410c08
round[ 6].start    80121e0776fd1d8a8d8c31bc965d1fee
round[ 6].s_box    cdc972c53854a47e5d64c765904cc028
round[ 6].s_row    cd54c7283864c0c55d4c727e90c9a465
round[ 6].m_col    921f748fd96e937d622d7725ba8ba50c
round[ 6].k_sch    f501857297448d7ebdf1c6ca87f33e3c
round[ 7].start    671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 7].s_box    8572a1542fe5727b9e86c8df27bc1404
round[ 7].s_row    85e5c8042f8614549ebca17b277272df
round[ 7].m_col    e913e7b18f507d4b227ef652758acbcc
round[ 7].k_sch    e510976183519b6934157c9ea351f1e0
round[ 8].start    0c0370d00c01e622166b8accd6db3a2c
round[ 8].s_box    fe7b5170fe7c8e93477f7e4bf6b98071
round[ 8].s_row    fe7c7e71fe7f807047b95193f67b8e4b
round[ 8].m_col    6cf5edf996eb0a069c4ef21cbfc25762
round[ 8].k_sch    1ea0372a995309167c439e77ff12051e
round[ 9].start    7255dad30fb80310e00d6c6b40d0527c
round[ 9].s_box    40fc5766766c7bcae1d7507f09700010
round[ 9].s_row    406c501076d70066e17057ca09fc7b7f
round[ 9].m_col    7478bcdce8a50b81d4327a9009188262
round[ 9].k_sch    dd7e0e887e2fff68608fc842f9dcc154
round[10].start    a906b254968af4e9b4bdb2d2f0c44336
round[10].s_box    d36f3720907ebf1e8d7a37b58c1c1a05
round[10].s_row    d37e3705907a1a208d1c371e8c6fbfb5
round[10].m_col    0d73cc2d8f6abe8b0cf2dd9bb83d422e
round[10].k_sch    859f5f237a8d5a3dc0c02952beefd63a
round[11].start    88ec930ef5e7e4b6cc32f4c906d29414
round[11].s_box    c4cedcabe694694e4b23bfdd6fb522fa
round[11].s_row    c494bffae62322ab4bb5dc4e6fce69dd
round[11].m_col    71d720933b6d677dc00b8f28238e0fb7
round[11].k_sch    de601e7827bcdf2ca223800fd8aeda32
round[12].start    afb73eeb1cd1b85162280f27fb20d585
round[12].s_box    79a9b2e99c3e6cd1aa3476cc0fb70397
round[12].s_row    793e76979c3403e9aab7b2d10fa96ccc
```

```
round[12].k_sch    a4970a331a78dc09c418c271e3a41d5d
round[12].output   dda97ca4864cdfe06eaf70a0ec0d7191

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   dda97ca4864cdfe06eaf70a0ec0d7191
round[ 0].ik_sch   a4970a331a78dc09c418c271e3a41d5d
round[ 1].istart   793e76979c3403e9aab7b2d10fa96ccc
round[ 1].is_row   79a9b2e99c3e6cd1aa3476cc0fb70397
round[ 1].is_box   afb73eeb1cd1b85162280f27fb20d585
round[ 1].ik_sch   de601e7827bcdf2ca223800fd8aeda32
round[ 1].ik_add   71d720933b6d677dc00b8f28238e0fb7
round[ 2].istart   c494bffae62322ab4bb5dc4e6fce69dd
round[ 2].is_row   c4cedcabe694694e4b23bfdd6fb522fa
round[ 2].is_box   88ec930ef5e7e4b6cc32f4c906d29414
round[ 2].ik_sch   859f5f237a8d5a3dc0c02952beefd63a
round[ 2].ik_add   0d73cc2d8f6abe8b0cf2dd9bb83d422e
round[ 3].istart   d37e3705907a1a208d1c371e8c6fbfb5
round[ 3].is_row   d36f3720907ebf1e8d7a37b58c1c1a05
round[ 3].is_box   a906b254968af4e9b4bdb2d2f0c44336
round[ 3].ik_sch   dd7e0e887e2fff68608fc842f9dcc154
round[ 3].ik_add   7478bcdce8a50b81d4327a9009188262
round[ 4].istart   406c501076d70066e17057ca09fc7b7f
round[ 4].is_row   40fc5766766c7bcae1d7507f09700010
round[ 4].is_box   7255dad30fb80310e00d6c6b40d0527c
round[ 4].ik_sch   1ea0372a995309167c439e77ff12051e
round[ 4].ik_add   6cf5edf996eb0a069c4ef21cbfc25762
round[ 5].istart   fe7c7e71fe7f807047b95193f67b8e4b
round[ 5].is_row   fe7b5170fe7c8e93477f7e4bf6b98071
round[ 5].is_box   0c0370d00c01e622166b8accd6db3a2c
round[ 5].ik_sch   e510976183519b6934157c9ea351f1e0
round[ 5].ik_add   e913e7b18f507d4b227ef652758acbcc
round[ 6].istart   85e5c8042f8614549ebca17b277272df
round[ 6].is_row   8572a1542fe5727b9e86c8df27bc1404
round[ 6].is_box   671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 6].ik_sch   f501857297448d7ebdf1c6ca87f33e3c
round[ 6].ik_add   921f748fd96e937d622d7725ba8ba50c
round[ 7].istart   cd54c7283864c0c55d4c727e90c9a465
round[ 7].is_row   cdc972c53854a47e5d64c765904cc028
round[ 7].is_box   80121e0776fd1d8a8d8c31bc965d1fee
round[ 7].ik_sch   2ab54bb43a02f8f662e3a95d66410c08
round[ 7].ik_add   aaa755b34cffe57cef6f98e1f01c13e6
round[ 8].istart   93faa123c2903f4743e4dd83431692de
round[ 8].is_row   9316dd47c2fa92834390a1de43e43f23
round[ 8].is_box   22ffc916a81474416496f19c64ae2532
round[ 8].ik_sch   58e151ab04a2a5557effb5416245080c
round[ 8].ik_add   7a1e98bdacb6d1141a6944dd06eb2d3e
round[ 9].istart   68cc08ed0abbd2bc642ef555244ae878
round[ 9].is_row   684af5bc0acce85564bb0878242ed2ed
round[ 9].is_box   f75c7778a327c8ed8cfebfc1a6c37f53
round[ 9].ik_sch   40f949b31cbabd4d48f043b810b7b342
round[ 9].ik_add   b7a53ecbbf9d75a0c40efc79b674cc11
round[10].istart   1fb5430ef0accf64aa370cde3d77792c
round[10].is_row   1f770c64f0b579deaaac432c3d37cf0e
round[10].is_box   cb02818c17d2af9c62aa64428bb25fd7
round[10].ik_sch   544afef55847f0fa4856e2e95c43f4fe
round[10].ik_add   9f487f794f955f662afc86abd7f1ab29
round[11].istart   84e1dd691a41d76f792d389783fbac70
```

```
round[11].is_row   84fb386f1ae1ac977941dd70832dd769
round[11].is_box   4f63760643e0aa85aff8c9d041fa0de4
round[11].ik_sch   101112131415161758462f95c43f4fe
round[11].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[12].istart   6353e08c0960e104cd70b751bacad0e7
round[12].is_row   63cab7040953d051cd60e0e7ba70e18c
round[12].is_box   00102030405060708090a0b0c0d0e0f0
round[12].ik_sch   000102030405060708090a0b0c0d0e0f
round[12].ioutput  00112233445566778899aabbccddeeff

EQUIVALENT INVERSE CIPHER (DECRYPT):
round[ 0].iinput   dda97ca4864cdfe06eaf70a0ec0d7191
round[ 0].ik_sch   a4970a331a78dc09c418c271e3a41d5d
round[ 1].istart   793e76979c3403e9aab7b2d10fa96ccc
round[ 1].is_box   afd10f851c28d5eb62203e51fbb7b827
round[ 1].is_row   afb73eeb1cd1b85162280f27fb20d585
round[ 1].im_col   122a02f7242ac8e20605afce51cc7264
round[ 1].ik_sch   d6bebd0dc209ea494db073803e021bb9
round[ 2].istart   c494bffae62322ab4bb5dc4e6fce69dd
round[ 2].is_box   88e7f414f532940eccd293b606ece4c9
round[ 2].is_row   88ec930ef5e7e4b6cc32f4c906d29414
round[ 2].im_col   5cc7aecce3c872194ae5ef8309a933c7
round[ 2].ik_sch   8fb999c973b26839c7f9d89d85c68c72
round[ 3].istart   d37e3705907a1a208d1c371e8c6fbfb5
round[ 3].is_box   a98ab23696bd4354b4c4b2e9f006f4d2
round[ 3].is_row   a906b254968af4e9b4bdb2d2f0c44336
round[ 3].im_col   b7113ed134e85489b20866b51d4b2c3b
round[ 3].ik_sch   f77d6ec1423f54ef5378317f14b75744
round[ 4].istart   406c501076d70066e17057ca09fc7b7f
round[ 4].is_box   72b86c7c0f0d52d3e0d0da104055036b
round[ 4].is_row   7255dad30fb80310e00d6c6b40d0527c
round[ 4].im_col   ef3b1be1b9b0e64bdcb79f1e0a707fbb
round[ 4].ik_sch   1147659047cf663b9b0ece8dfc0bf1f0
round[ 5].istart   fe7c7e71fe7f807047b95193f67b8e4b
round[ 5].is_box   0c018a2c0c6b3ad016db7022d603e6cc
round[ 5].is_row   0c0370d00c01e622166b8accd6db3a2c
round[ 5].im_col   592460b248832b2952e0b831923048f1
round[ 5].ik_sch   dcc1a8b667053f7dcc5c194ab5423a2e
round[ 6].istart   85e5c8042f8614549ebca17b277272df
round[ 6].is_box   672ab1304edc9bfddf78f1033d1e1eef
round[ 6].is_row   671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 6].im_col   0b8a7783417ae3a1f9492dc0c641a7ce
round[ 6].ik_sch   c6deb0ab791e2364a4055fbe568803ab
round[ 7].istart   cd54c7283864c0c55d4c727e90c9a465
round[ 7].is_box   80fd31ee768c1f078d5d1e8a96121dbc
round[ 7].is_row   80121e0776fd1d8a8d8c31bc965d1fee
round[ 7].im_col   4ee1ddf9301d6352c9ad769ef8d20515
round[ 7].ik_sch   dd1b7cdaf28d5c158a49ab1dbbc497cb
round[ 8].istart   93faa123c2903f4743e4dd83431692de
round[ 8].is_box   2214f132a896251664aec94164ff749c
round[ 8].is_row   22ffc916a81474416496f19c64ae2532
round[ 8].im_col   1008ffe53b36ee6af27b42549b8a7bb7
round[ 8].ik_sch   78c4f708318d3cd69655b701bfc093cf
round[ 9].istart   68cc08ed0abbd2bc642ef555244ae878
round[ 9].is_box   f727bf53a3fe7f788cc377eda65cc8c1
round[ 9].is_row   f75c7778a327c8ed8cfebfc1a6c37f53
round[ 9].im_col   7f69ac1ed939ebaac8ece3cb12e159e3
```

```
round[ 9].ik_sch   60dcef10299524ce62dbef152f9620cf
round[10].istart   1fb5430ef0accf64aa370cde3d77792c
round[10].is_box   cbd264d717aa5f8c62b2819c8b02af42
round[10].is_row   cb02818c17d2af9c62aa64428bb25fd7
round[10].im_col   cfaf16b2570c18b52e7fef50cab267ae
round[10].ik_sch   4b4ecbdb4d4dcfda5752d7c74949cbde
round[11].istart   84e1dd691a41d76f792d389783fbac70
round[11].is_box   4fe0c9e443f80d06affa76854163aad0
round[11].is_row   4f63760643e0aa85aff8c9d041fa0de4
round[11].im_col   794cf891177bfd1d8a327086f3831b39
round[11].ik_sch   1a1f181d1e1b1c194742c7d74949cbde
round[12].istart   6353e08c0960e104cd70b751bacad0e7
round[12].is_box   0050a0f04090e03080d02070c01060b0
round[12].is_row   00102030405060708090a0b0c0d0e0f0
round[12].ik_sch   0001020304050607080 90a0b0c0d0e0f
round[12].ioutput  00112233445566778899aabbccddeeff
```

C.3 AES-256 (*Nk*=8, *Nr*=14)

```
PLAINTEXT:  00112233445566778899aabbccddeeff
KEY:        000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

CIPHER (ENCRYPT):
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    101112131415161718191a1b1c1d1e1f
round[ 2].start    4f63760643e0aa85efa7213201a4e705
round[ 2].s_box    84fb386f1ae1ac97df5cfd237c49946b
round[ 2].s_row    84e1fd6b1a5c946fdf4938977cfbac23
round[ 2].m_col    bd2a395d2b6ac438d192443e615da195
round[ 2].k_sch    a573c29fa176c498a97fce93a572c09c
round[ 3].start    1859fbc28a1c00a078ed8aadc42f6109
round[ 3].s_box    adcb0f257e9c63e0bc557e951c15ef01
round[ 3].s_row    ad9c7e017e55ef25bc150fe01ccb6395
round[ 3].m_col    810dce0cc9db8172b3678c1e88a1b5bd
round[ 3].k_sch    1651a8cd0244beda1a5da4c10640bade
round[ 4].start    975c66c1cb9f3fa8a93a28df8ee10f63
round[ 4].s_box    884a33781fdb75c2d380349e19f876fb
round[ 4].s_row    88db34fb1f807678d3f833c2194a759e
round[ 4].m_col    b2822d81abe6fb275faf103a078c0033
round[ 4].k_sch    ae87dff00ff11b68a68ed5fb03fc1567
round[ 5].start    1c05f271a417e04ff921c5c104701554
round[ 5].s_box    9c6b89a349f0e18499fda678f2515920
round[ 5].s_row    9cf0a62049fd59a399518984f26be178
round[ 5].m_col    aeb65ba974e0f822d73f567bdb64c877
round[ 5].k_sch    6de1f1486fa54f9275f8eb5373b8518d
round[ 6].start    c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 6].s_box    2e5bacf8af6ea9e73ac67a34c286ee2d
round[ 6].s_row    2e6e7a2dafc6eef83a86ace7c25ba934
round[ 6].m_col    b951c33c02e9bd29ae25cdb1efa08cc7
round[ 6].k_sch    c656827fc9a799176f294cec6cd5598b
round[ 7].start    7f074143cb4e243ec10c815d8375d54c
round[ 7].s_box    d2c5831a1f2f36b278fe0c4cec9d0329
```

```
round[ 7].s_row    d22f0c291ffe031a789d83b2ecc5364c
round[ 7].m_col    ebb19e1c3ee7c9e87d7535e9ed6b9144
round[ 7].k_sch    3de23a75524775e727bf9eb45407cf39
round[ 8].start    d653a4696ca0bc0f5acaab5db96c5e7d
round[ 8].s_box    f6ed49f950e06576be74624c565058ff
round[ 8].s_row    f6e062ff507458f9be50497656ed654c
round[ 8].m_col    5174c8669da98435a8b3e62ca974a5ea
round[ 8].k_sch    0bdc905fc27b0948ad5245a4c1871c2f
round[ 9].start    5aa858395fd28d7d05e1a38868f3b9c5
round[ 9].s_box    bec26a12cfb55dff6bf80ac4450d56a6
round[ 9].s_row    beb50aa6cff856126b0d6aff45c25dc4
round[ 9].m_col    0f77ee31d2ccadc05430a83f4ef96ac3
round[ 9].k_sch    45f5a66017b2d387300d4d33640a820a
round[10].start    4a824851c57e7e47643de50c2af3e8c9
round[10].s_box    d61352d1a6f3f3a04327d9fee50d9bdd
round[10].s_row    d6f3d9dda6279bd1430d52a0e513f3fe
round[10].m_col    bd86f0ea748fc4f4630f11c1e9331233
round[10].k_sch    7ccff71cbeb4fe5413e6bbf0d261a7df
round[11].start    c14907f6ca3b3aa070e9aa313b52b5ec
round[11].s_box    783bc54274e280e0511eacc7e200d5ce
round[11].s_row    78e2acce741ed5425100c5e0e23b80c7
round[11].m_col    af8690415d6e1dd387e5fbedd5c89013
round[11].k_sch    f01afafee7a82979d7a5644ab3afe640
round[12].start    5f9c6abfbac634aa50409fa766677653
round[12].s_box    cfde0208f4b418ac5309db5c338538ed
round[12].s_row    cfb4dbedf4093808538502ac33de185c
round[12].m_col    7427fae4d8a695269ce83d315be0392b
round[12].k_sch    2541fe719bf500258813bbd55a721c0a
round[13].start    516604954353950314fb86e401922521
round[13].s_box    d133f22a1aed2a7bfa0f44697c4f3ffd
round[13].s_row    d1ed44fd1a0f3f2afa4ff27b7c332a69
round[13].m_col    2c21a820306f154ab712c75eee0da04f
round[13].k_sch    4e5a6699a9f24fe07e572baacdf8cdea
round[14].start    627bceb9999d5aaac945ecf423f56da5
round[14].s_box    aa218b56ee5ebeacdd6ecebf26e63c06
round[14].s_row    aa5ece06ee6e3c56dde68bac2621bebf
round[14].k_sch    24fc79ccbf0979e9371ac23c6d68de36
round[14].output   8ea2b7ca516745bfeafc49904b496089

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   8ea2b7ca516745bfeafc49904b496089
round[ 0].ik_sch   24fc79ccbf0979e9371ac23c6d68de36
round[ 1].istart   aa5ece06ee6e3c56dde68bac2621bebf
round[ 1].is_row   aa218b56ee5ebeacdd6ecebf26e63c06
round[ 1].is_box   627bceb9999d5aaac945ecf423f56da5
round[ 1].ik_sch   4e5a6699a9f24fe07e572baacdf8cdea
round[ 1].ik_add   2c21a820306f154ab712c75eee0da04f
round[ 2].istart   d1ed44fd1a0f3f2afa4ff27b7c332a69
round[ 2].is_row   d133f22a1aed2a7bfa0f44697c4f3ffd
round[ 2].is_box   516604954353950314fb86e401922521
round[ 2].ik_sch   2541fe719bf500258813bbd55a721c0a
round[ 2].ik_add   7427fae4d8a695269ce83d315be0392b
round[ 3].istart   cfb4dbedf4093808538502ac33de185c
round[ 3].is_row   cfde0208f4b418ac5309db5c338538ed
round[ 3].is_box   5f9c6abfbac634aa50409fa766677653
round[ 3].ik_sch   f01afafee7a82979d7a5644ab3afe640
round[ 3].ik_add   af8690415d6e1dd387e5fbedd5c89013
```

```
round[ 4].istart   78e2acce741ed5425100c5e0e23b80c7
round[ 4].is_row   783bc54274e280e0511eacc7e200d5ce
round[ 4].is_box   c14907f6ca3b3aa070e9aa313b52b5ec
round[ 4].ik_sch   7ccff71cbeb4fe5413e6bbf0d261a7df
round[ 4].ik_add   bd86f0ea748fc4f4630f11c1e9331233
round[ 5].istart   d6f3d9dda6279bd1430d52a0e513f3fe
round[ 5].is_row   d61352d1a6f3f3a04327d9fee50d9bdd
round[ 5].is_box   4a824851c57e7e47643de50c2af3e8c9
round[ 5].ik_sch   45f5a66017b2d387300d4d33640a820a
round[ 5].ik_add   0f77ee31d2ccadc05430a83f4ef96ac3
round[ 6].istart   beb50aa6cff856126b0d6aff45c25dc4
round[ 6].is_row   bec26a12cfb55dff6bf80ac4450d56a6
round[ 6].is_box   5aa858395fd28d7d05e1a38868f3b9c5
round[ 6].ik_sch   0bdc905fc27b0948ad5245a4c1871c2f
round[ 6].ik_add   5174c8669da98435a8b3e62ca974a5ea
round[ 7].istart   f6e062ff507458f9be50497656ed654c
round[ 7].is_row   f6ed49f950e06576be74624c565058ff
round[ 7].is_box   d653a4696ca0bc0f5acaab5db96c5e7d
round[ 7].ik_sch   3de23a75524775e727bf9eb45407cf39
round[ 7].ik_add   ebb19e1c3ee7c9e87d7535e9ed6b9144
round[ 8].istart   d22f0c291ffe031a789d83b2ecc5364c
round[ 8].is_row   d2c5831a1f2f36b278fe0c4cec9d0329
round[ 8].is_box   7f074143cb4e243ec10c815d8375d54c
round[ 8].ik_sch   c656827fc9a799176f294cec6cd5598b
round[ 8].ik_add   b951c33c02e9bd29ae25cdb1efa08cc7
round[ 9].istart   2e6e7a2dafc6eef83a86ace7c25ba934
round[ 9].is_row   2e5bacf8af6ea9e73ac67a34c286ee2d
round[ 9].is_box   c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 9].ik_sch   6de1f1486fa54f9275f8eb5373b8518d
round[ 9].ik_add   aeb65ba974e0f822d73f567bdb64c877
round[10].istart   9cf0a62049fd59a399518984f26be178
round[10].is_row   9c6b89a349f0e18499fda678f2515920
round[10].is_box   1c05f271a417e04ff921c5c104701554
round[10].ik_sch   ae87dff00ff11b68a68ed5fb03fc1567
round[10].ik_add   b2822d81abe6fb275faf103a078c0033
round[11].istart   88db34fb1f807678d3f833c2194a759e
round[11].is_row   884a33781fdb75c2d380349e19f876fb
round[11].is_box   975c66c1cb9f3fa8a93a28df8ee10f63
round[11].ik_sch   1651a8cd0244beda1a5da4c10640bade
round[11].ik_add   810dce0cc9db8172b3678c1e88a1b5bd
round[12].istart   ad9c7e017e55ef25bc150fe01ccb6395
round[12].is_row   adcb0f257e9c63e0bc557e951c15ef01
round[12].is_box   1859fbc28a1c00a078ed8aadc42f6109
round[12].ik_sch   a573c29fa176c498a97fce93a572c09c
round[12].ik_add   bd2a395d2b6ac438d192443e615da195
round[13].istart   84e1fd6b1a5c946fdf4938977cfbac23
round[13].is_row   84fb386f1ae1ac97df5cfd237c49946b
round[13].is_box   4f63760643e0aa85efa7213201a4e705
round[13].ik_sch   101112131415161718191a1b1c1d1e1f
round[13].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[14].istart   6353e08c0960e104cd70b751bacad0e7
round[14].is_row   63cab7040953d051cd60e0e7ba70e18c
round[14].is_box   00102030405060708090a0b0c0d0e0f0
round[14].ik_sch   000102030405060708090a0b0c0d0e0f
round[14].ioutput  00112233445566778899aabbccddeeff
```

EQUIVALENT INVERSE CIPHER (DECRYPT):

```
round[ 0].iinput   8ea2b7ca516745bfeafc49904b496089
round[ 0].ik_sch   24fc79ccbf0979e9371ac23c6d68de36
round[ 1].istart   aa5ece06ee6e3c56dde68bac2621bebf
round[ 1].is_box   629deca599456db9c9f5ceaa237b5af4
round[ 1].is_row   627bceb9999d5aaac945ecf423f56da5
round[ 1].im_col   e51c9502a5c1950506a61024596b2b07
round[ 1].ik_sch   34f1d1ffbfceaa2ffce9e25f2558016e
round[ 2].istart   d1ed44fd1a0f3f2afa4ff27b7c332a69
round[ 2].is_box   5153862143fb259514920403016695e4
round[ 2].is_row   516604954353950314fb86e401922521
round[ 2].im_col   91a29306cc450d0226f4b5eaef5efed8
round[ 2].ik_sch   5e1648eb384c350a7571b746dc80e684
round[ 3].istart   cfb4dbedf4093808538502ac33de185c
round[ 3].is_box   5fc69f53ba4076bf50676aaa669c34a7
round[ 3].is_row   5f9c6abfbac634aa50409fa766677653
round[ 3].im_col   b041a94eff21ae9212278d903b8a63f6
round[ 3].ik_sch   c8a305808b3f7bd043274870d9b1e331
round[ 4].istart   78e2acce741ed5425100c5e0e23b80c7
round[ 4].is_box   c13baaeccae9b5f6705207a03b493a31
round[ 4].is_row   c14907f6ca3b3aa070e9aa313b52b5ec
round[ 4].im_col   638357cec07de6300e30d0ec4ce2a23c
round[ 4].ik_sch   b5708e13665a7de14d3d824ca9f151c2
round[ 5].istart   d6f3d9dda6279bd1430d52a0e513f3fe
round[ 5].is_box   4a7ee5c9c53de85164f348472a827e0c
round[ 5].is_row   4a824851c57e7e47643de50c2af3e8c9
round[ 5].im_col   ca6f71058c642842a315595fdf54f685
round[ 5].ik_sch   74da7ba3439c7e50c81833a09a96ab41
round[ 6].istart   beb50aa6cff856126b0d6aff45c25dc4
round[ 6].is_box   5ad2a3c55fe1b93905f3587d68a88d88
round[ 6].is_row   5aa858395fd28d7d05e1a38868f3b9c5
round[ 6].im_col   ca46f5ea835eab0b9537b6dbb221b6c2
round[ 6].ik_sch   3ca69715d32af3f22b67ffade4ccd38e
round[ 7].istart   f6e062ff507458f9be50497656ed654c
round[ 7].is_box   d6a0ab7d6cca5e695a6ca40fb953bc5d
round[ 7].is_row   d653a4696ca0bc0f5acaab5db96c5e7d
round[ 7].im_col   2a70c8da28b806e9f319ce42be4baead
round[ 7].ik_sch   f85fc4f3374605f38b844df0528e98e1
round[ 8].istart   d22f0c291ffe031a789d83b2ecc5364c
round[ 8].is_box   7f4e814ccb0cd543c175413e8307245d
round[ 8].is_row   7f074143cb4e243ec10c815d8375d54c
round[ 8].im_col   f0073ab7404a8a1fc2cba0b80df08517
round[ 8].ik_sch   de69409aef8c64e7f84d0c5fcfab2c23
round[ 9].istart   2e6e7a2dafc6eef83a86ace7c25ba934
round[ 9].is_box   c345bdfa1bc799e1a2dcaab0a857b728
round[ 9].is_row   c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 9].im_col   3225fe3686e498a32593c1872b613469
round[ 9].ik_sch   aed55816cf19c100bcc24803d90ad511
round[10].istart   9cf0a62049fd59a399518984f26be178
round[10].is_box   1c17c554a4211571f970f24f0405e0c1
round[10].is_row   1c05f271a417e04ff921c5c104701554
round[10].im_col   9d1d5c462e655205c4395b7a2eac55e2
round[10].ik_sch   15c668bd31e5247d17c168b837e6207c
round[11].istart   88db34fb1f807678d3f833c2194a759e
round[11].is_box   979f2863cb3a0fc1a9e166a88e5c3fdf
round[11].is_row   975c66c1cb9f3fa8a93a28df8ee10f63
round[11].im_col   d24bfb0e1f997633cfce86e37903fe87
round[11].ik_sch   7fd7850f61cc991673db890365c89d12
```

```
round[12].istart   ad9c7e017e55ef25bc150fe01ccb6395
round[12].is_box   181c8a098aed61c2782ffba0c45900ad
round[12].is_row   1859fbc28a1c00a078ed8aadc42f6109
round[12].im_col   aec9bda23e7fd8aff96d74525cdce4e7
round[12].ik_sch   2a2840c924234cc026244cc5202748c4
round[13].istart   84e1fd6b1a5c946fdf4938977cfbac23
round[13].is_box   4fe0210543a7e706efa476850163aa32
round[13].is_row   4f63760643e0aa85efa7213201a4e705
round[13].im_col   794cf891177bfd1ddf67a744acd9c4f6
round[13].ik_sch   1a1f181d1e1b1c191217101516131411
round[14].istart   6353e08c0960e104cd70b751bacad0e7
round[14].is_box   0050a0f04090e03080d02070c01060b0
round[14].is_row   00102030405060708090a0b0c0d0e0f0
round[14].ik_sch   000102030405060708090a0b0c0d0e0f
round[14].ioutput  00112233445566778899aabbccddeeff
```

Appendix D - References

[1] AES page available via http://www.nist.gov/CryptoToolkit.[4]

[2] Computer Security Objects Register (CSOR): http://csrc.nist.gov/csor/.

[3] J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999, available at [1].

[4] J. Daemen and V. Rijmen, *The block cipher Rijndael*, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp. 288-296.

[5] B. Gladman's AES related home page http://fp.gladman.plus.com/cryptography_technology/.

[6] A. Lee, NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, National Institute of Standards and Technology, November 1999.

[7] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997, p. 81-83.

[8] J. Nechvatal, et. al., *Report on the Development of the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, October 2, 2000, available at [1].

[4] A complete set of documentation from the AES development effort – including announcements, public comments, analysis papers, conference proceedings, etc. – is available from this site.

FIPS PUB *#HMAC*

FEDERAL INFORMATION PROCESSING STANDARD PUBLICATION

The Keyed-Hash Message Authentication Code (HMAC)

CATEGORY: COMPUTER SECURITY **SUBCATEGORY: CRYPTOGRAPHY**

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

Issued MONTH DAY, 2001

U.S. Department of Commerce
Norman Y. Mineta, Secretary

Technology Administration
Cheryl L. Shavers, Under Secretary for Technology

National Institute of Standards and Technology
Raymond G. Kammer, Director

Foreword

The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST) is the official series of publications relating to standards and guidelines adopted and promulgated under the provisions of Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235). These mandates have given the Secretary of Commerce and NIST important responsibilities for improving the utilization and management of computer and related telecommunications systems in the Federal government. The NIST, through its Information Technology Laboratory, provides leadership, technical guidance, and coordination of government efforts in the development of standards and guidelines in these areas.

Comments concerning Federal Information Processing Standards Publications are welcomed and should be addressed to the Director, Information Technology Laboratory, National Institute of Standards and Technology, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

William Mehuron, Director
Information Technology Laboratory

Abstract

This standard describes a keyed-hash message authentication code (HMAC), a mechanism for message authentication using cryptographic hash functions. HMAC can be used with any iterative FIPS-approved cryptographic hash function, in combination with a shared secret key. The cryptographic strength of HMAC depends on the properties of the underlying hash function. The HMAC specification in this standard is a generalization of Internet RFC 2104, *HMAC, Keyed-Hashing for Message Authentication*, and ANSI X9.71, *Keyed Hash Message Authentication Code*.

Keywords: computer security, cryptography, HMAC, MAC, message authentication, Federal Information Processing Standard (FIPS).

National Institute of Standards and Technology
FIPS PUB *#HMAC*
pages (MONTH DAY, 2001)
CODEN: XXXXX

U.S. Government Printing Office
Washington, 2000

For Sale by the National Technical Information Service
U.S. Department of Commerce
Springfield, VA 22161

Federal Information Processing Standards Publication *#HMAC*

2001 MONTH DAY

Announcing the Standard for

The Keyed-Hash Message Authentication Code (HMAC)

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235).

1. **Name of Standard.** Keyed-Hash Message Authentication Code (HMAC) (FIPS PUB *#HMAC*).

2. **Category of Standard.** Computer Security Standard. **Subcategory.** Cryptography.

3. **Explanation.** This standard specifies an algorithm for applications requiring message authentication. Message authentication is achieved via the construction of a message authentication code (MAC). MACs based on cryptographic hash functions are known as HMACs.

The purpose of a MAC is to authenticate both the source of a message and its integrity without the use of any additional mechanisms. HMACs have two functionally distinct parameters, a message input and a secret key known only to the message originator and intended receiver(s). Additional applications of keyed hash functions include their use in challenge-response identification protocols for computing responses, which are a function of both a secret key and a challenge message.

An HMAC function is used by the message sender to produce a value (the MAC) that is formed by condensing the secret key and the message input. The MAC is typically sent to the message receiver along with the message. The receiver computes the MAC on the received message using the same key and HMAC function as was used by the sender, and compares the result computed with the received MAC. If the two values match, the message has been correctly received, and the receiver is assured that the sender is a member of the community of users that share the key.

The HMAC specification in this standard is a generalization of HMAC as specified in Internet RFC 2104, *HMAC, Keyed-Hashing for Message Authentication*, and ANSI X9.71, *Keyed Hash Message Authentication Code.*

4. **Approving Authority.** Secretary of Commerce.

5. Maintenance Agency. Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. Applicability. This standard is applicable to all Federal departments and agencies for the protection of sensitive unclassified information that is not subject to section 2315 of Title 10, United States Code, or section 3502(2) of Title 44, United States Code. This standard shall be used in designing, acquiring and implementing message authentication in systems that Federal departments and agencies operate or which are operated for them under contract. The adoption and use of this standard is available to private and commercial organizations.

7. Specifications. Federal Information Processing Standard (FIPS) *#HMAC*, Keyed-Hash Message Authentication Code (HMAC) (affixed).

8. Implementations. Cryptographic modules that implement this standard shall conform to FIPS 140-1. The authentication mechanism described in this standard may be implemented in software, firmware, hardware, or any combination thereof. NIST has developed a Cryptographic Module Validation Program that will test implementations for conformance with this HMAC standard. Information on this program is available at http://csrc.nist.gov/cryptval/.

Agencies are advised that keys used for HMAC applications should not be used for other purposes.

9. Other Approved Security Functions. HMAC implementations that comply with this standard shall employ cryptographic algorithms, cryptographic key generation algorithms and key management techniques that have been approved for protecting Federal government sensitive information. Approved cryptographic algorithms and techniques include those that are either:

a. specified in a Federal Information Processing Standard (FIPS), or
b. adopted in a FIPS and specified either in an appendix to the FIPS or in a document referenced by the FIPS.

10. Export Control. Certain cryptographic devices and technical data regarding them are subject to Federal export controls and exports of cryptographic modules implementing this standard and technical data regarding them must comply with these Federal regulations and be licensed by the Bureau of Export Administration of the U.S. Department of Commerce. Applicable Federal government export controls are specified in Title 15, Code of Federal Regulations (CFR) Part 740.17; Title 15, CFR Part 742; and Title 15, CFR Part 774, Category 5, Part 2.

11. Implementation Schedule. This standard becomes effective on [insert date: six months after approval by the Secretary of Commerce].

12. Qualifications. The security afforded by the HMAC function is dependent on maintaining the secrecy of the key. Users must therefore guard against disclosure of these

keys. While it is the intent of this standard to specify a mechanism to provide message authentication, conformance to this standard does not assure that a particular implementation is secure. It is the responsibility of the implementer to ensure that any module containing an HMAC implementation is designed and built in a secure manner.

Similarly, the use of a product containing an implementation that conforms to this standard does not guarantee the security of the overall system in which the product is used. The responsible authority in each agency shall assure that an overall system provides an acceptable level of security.

Since a standard of this nature must be flexible enough to adapt to advancements and innovations in science and technology, this standard will be reviewed every five years in order to assess its adequacy.

13. Waiver Procedure. Under certain exceptional circumstances, the heads of Federal agencies, or their delegates, may approve waivers to Federal Information Processing Standards (FIPS). The heads of such agencies may redelegate such authority only to a senior official designated pursuant to Section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when compliance with this standard would

a. adversely affect the accomplishment of the mission of an operator of Federal computer system or

b. cause a major adverse financial impact on the operator that is not offset by government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision that explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decision, Information Technology Laboratory, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is

authorized and decides to make under Section 552(b) of Title 5, U.S. Code, shall be part of the procurement documentation and retained by the agency.

14. Where to obtain copies. This publication is available by accessing http://csrc.nist.gov/publications/. A list of other available computer security publications, including ordering information, can be obtained from NIST Publications List 91, which is available at the same web site. Alternatively, copies of NIST computer security publications are available from: National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, VA 22161.

Federal Information Processing Standards Publication xxx

2000 *Month Day*

Specifications for

The Keyed-Hash Message Authentication Code

TABLE OF CONTENTS

1. INTRODUCTION

Providing a way to check the integrity of information transmitted over or stored in an unreliable medium is a prime necessity in the world of open computing and communications. Mechanisms that provide such integrity checks based on a secret key are usually called message authentication codes (MACs). Typically, message authentication codes are used between two parties that share a secret key in order to authenticate information transmitted between these parties. This standard defines a MAC that uses a cryptographic hash function in conjunction with a secret key. This mechanism is called HMAC and is a generalization of HMAC as specificatied in [RFC2104] and [ANSIX9.17].

HMAC shall be used in combination with a cryptographic hash function specified in a Federal Information Processing Standard (FIPS). HMAC uses a secret key for the calculation and verification of the MACs. The main goals behind the HMAC construction [RFC2104] are:

- To use available hash functions without modifications; in particular, hash functions that perform well in software, and for which code is freely and widely available,

- To preserve the original performance of the hash function without incurring a significant degradation,

- To use and handle keys in a simple way,

- To have a well-understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the underlying hash function, and

- To allow for easy replaceability of the underlying hash function in the event that faster or more secure hash functions are later available.

2. GLOSSARY OF TERMS AND ACRONYMS

2.1 Glossary of Terms

The following definitions are used throughout this standard:

FIPS-Approved: An algorithm or technique that is either 1) specified in a FIPS, or 2) adopted in a FIPS and specified either in an appendix to the FIPS or in a document referenced by the FIPS..

Message Authentication Code (MAC): a cryptographic checksum that results from passing data through a message authentication algorithm. In this standard, the message authentication algorithm is called HMAC.

Cryptographic key (key): a parameter used in conjunction with a cryptographic algorithm that determines the specific operation of that algorithm. In this standard, the cryptographic key is used by the HMAC algorithm to produce a MAC on the data.

Keyed hash-based message authentication code (HMAC): a message authentication code that uses a cryptographic key in conjunction with a hash function.

Secret key: a cryptographic key that is uniquely associated with one or more entities. The use of the term "secret" in this context does not imply a classification level; rather the term implies the need to protect the key from disclosure or substitution.

2.2 Acronyms

The following acronyms and abbreviations are used throughout this standard:

FIPS	Federal Information Processing Standard
FIPS PUB	FIPS Publication
HMAC	Keyed-Hash Message Authentication Code
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
SHA-1	The Secure Hash Algorithm specified in FIPS 180-1.

2.3 HMAC Parameters and Symbols

HMAC uses the following parameters:

B	Block size (in bytes) of the input to the FIPS-approved hash function; e.g., for SHA-1, *B* = 64.
H	FIPS-approved hash function, e.g., FIPS 180-1, *Secure Hash Algorithm-1 (SHA-1).*
ipad	Inner pad; the byte x’36’ repeated *B* times.
K	Secret key shared between the originator and the intended receiver(s).
K_0	The key *K* with zeros appended to form a *B* byte key.

L Block size (in bytes) of the output of the FIPS-approved hash function; for SHA-1, *L* = 20.

opad Outer pad; the byte x'5c' repeated *B* times.

t The number of bytes of MAC.

text The data on which the HMAC is calculated; the length of the data is *n* bits, where the maximum value for *n* depends on the hash algorithm used.

x'*N*' Hexadecimal notation, where each '*N*' represents 4 binary bits.

|| Concatenation

⊕ Exclusive-Or operation.

3. CRYPTOGRAPHIC KEYS

The size of the key, *K*, shall be equal to or greater than *L*/2, where *L* is the size of the hash function output. Note that keys greater than *L* bytes do not significantly increase the function strength. Applications that use keys longer than *B*-bytes shall first hash the key using *H* and then use the resultant *L*-byte string as the HMAC key, *K*. Keys shall be chosen at random using a FIPS-approved key generation method and shall be changed periodically. The keys shall be protected in a manner that is consistent with the value of the data that is to be protected (i.e., the data that is authenticated using the HMAC function).

4. TRUNCATED OUTPUT

A well-known practice with MACs is to truncate their output (i.e., the length of the MAC used is less than the length of the output of the MAC function *L*). Applications of this standard may truncate the output of HMAC. When a truncated HMAC is used, the *t* leftmost bytes of the HMAC computation shall be used as the MAC. The output length, *t*, shall be no less than four bytes (i.e., $4 \leq t \leq L$). However, *t* shall be at least $\frac{L}{2}$ bytes (i.e., $\frac{L}{2} \leq t \leq L$) unless an application or protocol makes numerous trials impractical. For example, a low bandwidth channel might prevent numerous trials on a 4 byte MAC, or a protocol might allow only a small number of invalid MAC attempts.

5. HMAC SPECIFICATION

To compute a MAC over the data '*text*' using the HMAC function, the following operation is performed:

$$\boldsymbol{MAC(text)_t = HMAC(K, text)_t = H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))_t}$$

Table 1 illustrates the step by step process in the HMAC algorithm, which is depicted in Figure 1.

Table 1: The HMAC Algorithm

STEPS	STEP-BY-STEP DESCRIPTION
Step 1	If the length of $K = B$, set $K_0 = K$. Go to step 4.
Step 2	If the length of $K > B$, hash K to obtain an L byte string: $K = H(K)$.
Step 3	If the length of $K < B$, append zeros to the end of K to create a B-byte string K_0 (e.g., if K is 20 bytes in length and $B = 64$, then K will be appended with 44 zero bytes 0x00).
Step 4	Exclusive-Or K_0 with *ipad* to produce a B-byte string: $\boldsymbol{K_0 \oplus ipad}$.
Step 5	Append the stream of data '*text*' to the string resulting from step 4: $\boldsymbol{(K_0 \oplus ipad) \| text}$.
Step 6	Apply H to the stream generated in step 5: $\mathbf{H}(\boldsymbol{(K_0 \oplus ipad) \| text})$.
Step 7	Exclusive-Or K_0 with *opad*: $\boldsymbol{K_0 \oplus opad}$.
Step 8	Append the result from step 6 to step 7: $\boldsymbol{(K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text)}$.
Step 9	Apply H to the result from step 8: $\boldsymbol{H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))}$.
Step 10	Select the leftmost t bytes of the result of step 9 as the MAC.

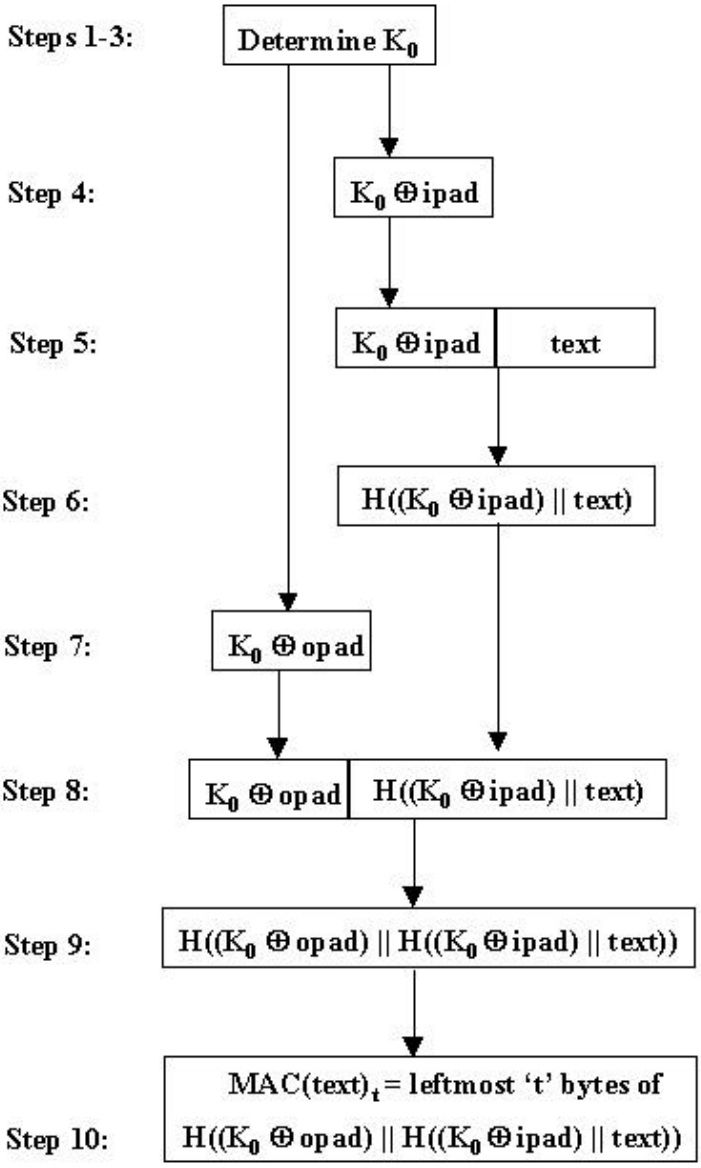


Figure 1: Illustration of the HMAC Construction

6. IMPLEMENTATION NOTE

The HMAC algorithm is specified for an arbitrary FIPS-approved cryptographic hash function, *H*. With minor modifications, an HMAC implementation can easily replace one hash function, *H*, with another hash function, *H'*.

Conceptually, the intermediate results of the compression function on the *B*-byte blocks ($K_0 \oplus ipad$) and ($K_0 \oplus opad$) can be precomputed once, at the time of generation of the

key *K*, or before its first use. These intermediate results can be stored and then used to initialize *H* each time that a message needs to be authenticated using the same key. For each authenticated message using the key *K*, this method saves the application of the hash function of *H* on two *B*-byte blocks (i.e., on ($K \oplus ipad$) and ($K \oplus opad$)). This saving may be significant when authenticating short streams of data. **These stored intermediate values shall be treated and protected in the same manner as secret keys.**

Choosing to implement HMAC in this manner has no effect on interoperability.

APPENDIX A: HMAC EXAMPLES

These examples are provided in order to promote correct implementations of HMAC.

A.1 SHA-1 Examples: B = 64 bytes; L = 20 bytes

[NOTE: These examples were taken from ANSI X9.71]

Test case 1 (20 byte key; 20 byte HMAC)**:**

key = x’0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b’

data = "Hi There"

HMAC = x’b617318655057264e28bc0b6fb378c8ef146be00’

Test case 2 (20 byte key; 20 byte HMAC):

key = x’aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa’

data = x’dd’ repeated 50 times

HMAC = x’125d7342b9ac11cd91a39af48aa17b4f63f175d3’

Test case 3 (25 byte key; 20 byte HMAC):

key = x’0102030405060708090a0b0c0d0e0f10111213141516171819’

data = x’cd’ repeated 50 times

HMAC = x’4c9007f4026250c6bc8414f9bf50c86c2d7235da’

Test case 4 (20 byte key; 20 byte MAC; 12 byte MAC):

key = x’0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c’

data = "Test With Truncation"

HMAC = x’4c1a03424b55e07fe7f27be1d58bb9324a9a5a04’

96-bit MAC = x’4c1a03424b55e07fe7f27be1’

APPENDIX B: REFERENCES

[ANSIX9.71] American Bankers Association, *Keyed Hash Message Authentication Code,* ANSI X9.71, Washington, D.C., 2000.

[FIPS113] National Institute of Standards and Technology, *Computer Data Authentication,* Federal Information Processing Standards Publication 113, 30 May 1985.

[FIPS140-2] National Institute of Standards and Technology, *Security Requirements for Cryptographic Modules,* Federal Information Processing Standards Publication 140-2, DD Month 2000.

[FIPS171] National Institute of Standards and Technology, *Key Management Using ANSI X9.17,* Federal Information Processing Standards Publication 171, 27 April 1992.

[FIPS180-1] National Institute of Standards and Technology, *Secure Hash Standard (SHS),* Federal Information Processing Standards Publication 180-1, 17 April 1995.

[ISO9797-2] Joint Technical Committee ISO/IEC JTC 1 Subcommittee SC 27, *Information technology – Security techniques – Message authentication codes (MACs) – Part 2: Mechanisms using a hash-function,* ISO/IEC FCD 9797-2, 15 July 1999.

[RFC2104] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication,* Internet Engineering Task Force, Request for Comments (RFC) 2104, February 1997.

[RFC2404] C. Madson and R. Glenn, *The Use of HMAC-SHA-1-96 within ESP and AH,* Internet Engineering Task Force, Request for Comments (RFC) 2404, November 1998.

Federal Information
Processing Standards Publication 180-2

2001 MONTH DAY

Announcing the

SECURE HASH STANDARD

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106), and the Computer Security Act of 1987 (Public Law 100-235).

1. Name of Standard: Secure Hash Signature Standard (SHS) (FIPS PUB 180-2).

2. Category of Standard: Computer Security Standard, Cryptography.

3. Explanation: This Standard specifies four secure hash algorithms - SHA-1, SHA-256, SHA-384, and SHA-512 - for computing a condensed representation of electronic data (message). When a message of any length $< 2^{64}$ bits (for SHA-1 and SHA-256) or $< 2^{128}$ bits (for SHA-384 and SHA-512) is input to an algorithm, the result is an output called a message digest. The message digests range in length from 160 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits).

The four hash algorithms specified in this standard are called secure because, for a given algorithm, it is computationally infeasible 1) to find a message that corresponds to a given message digest, or 2) to find two different messages that produce the same message digest. Any change to a message will, with a very high probability, result in a different message digest. This will result in a verification failure when the secure hash algorithm is used with a digital signature algorithm or a keyed-hash message authentication algorithm.

This standard supersedes FIPS 180-1, adding three algorithms that are capable of producing larger message digests. The SHA-1 algorithm specified herein is the same algorithm that was specified previously in FIPS 180-1, although some of the notation has been modified to be consistent with the notation used in the SHA-256, SHA-384, and SHA-512 algorithms.

4. Approving Authority: Secretary of Commerce.

5. Maintenance Agency: U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL).

6. Applicability: This standard is applicable to all Federal departments and agencies for the protection of sensitive unclassified information that is not subject to section 2315 of Title 10, United States Code, or section 3502(2) of Title 44, United States Code. This standard shall be implemented whenever a secure hash algorithm is required for Federal applications, including use by other cryptographic algorithms and protocols. The adoption and use of this standard is available to private and commercial organizations.

7. Specifications: Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard (SHS) (affixed).

8. Implementations: The secure hash algorithms specified herein may be implemented in software, firmware, hardware or any combination thereof. Only algorithm implementations that are validated by NIST will be considered as complying with this standard. Information about the planned validation program can be obtained at http://csrc.nist.gov/cryptval/ or from the National Institute of Standards and Technology, Information Technology Laboratory, Attn: SHS Validation, 100 Bureau Drive Stop 8930, Gaithersburg, MD 20899-8930.

9. Implementation Schedule: This standard becomes effective on [insert date: six months after approval by the Secretary of Commerce].

10. Patents: Implementations of the secure hash algorithms in this standard may be covered by U.S. or foreign patents.

11. Export Control: Certain cryptographic devices and technical data regarding them are subject to Federal export controls. Exports of cryptographic modules implementing this standard and technical data regarding them must comply with these Federal regulations and be licensed by the Bureau of Export Administration of the U.S. Department of Commerce. Applicable Federal government export controls are specified in Title 15, Code of Federal Regulations (CFR) Part 740.17; Title 15, CFR Part 742; and Title 15, CFR Part 774, Category 5, Part 2.

12. Qualifications: While it is the intent of this standard to specify general security requirements for generating a message digest, conformance to this standard does not assure that a particular implementation is secure. The responsible authority in each agency or department shall assure that an overall implementation provides an acceptable level of security. This standard will be reviewed every five years in order to assess its adequacy.

13. Waiver Procedure. Under certain exceptional circumstances, the heads of Federal agencies, or their delegates, may approve waivers to Federal Information Processing Standards (FIPS). The heads of such agencies may redelegate such authority only to a senior official designated pursuant to Section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when compliance with this standard would

a. adversely affect the accomplishment of the mission of an operator of a Federal computer system or
b. cause a major adverse financial impact on the operator that is not offset by government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision that explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decision, Information Technology Laboratory, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

In addition, a notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under Section 552(b) of Title 5, U.S. Code, shall be part of the procurement documentation and retained by the agency.

14. Where to Obtain Copies of the Standard: This publication is available electronically by accessing http://csrc.nist.gov/publications/. A list of other available computer security publications, including ordering information, can be obtained from NIST Publications List 91, which is available at the same web site. Alternatively, copies of NIST computer security publications are available from: National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, VA 22161.

Federal Information
Processing Standards Publication 180-2

2001 MONTH DAY

Specifications for the

SECURE HASH STANDARD

Table Of Contents

1. INTRODUCTION

This standard specifies four secure hash algorithms, SHA-1[1], SHA-256, SHA-384, and SHA-512. All four of the algorithms are iterative, one-way hash functions that can process a message to produce a condensed representation called a *message digest*. These algorithms enable the determination of a message's integrity: any change to the message will, with a very high probability, result in a different message digest. This property is useful in the generation and verification of digital signatures and message authentication codes, and in the generation of random numbers (bits).

Each algorithm can be described in two stages: preprocessing and hash computation. Preprocessing involves padding a message, parsing the padded message into *m*-bit blocks, and setting initialization values to be used in the hash computation. The hash computation generates a *message schedule* from the padded message and uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest.

The four algorithms differ most significantly in the number of bits of security that are provided for the data being hashed – this is directly related to the message digest length. When a secure hash algorithm is used in conjunction with another algorithm, there may be requirements specified elsewhere that require the use of a secure hash algorithm with a certain number of bits of security. For example, if a message is being signed with a digital signature algorithm that provides 128 bits of security, then that signature algorithm may require the use of a secure hash algorithm that also provides 128 bits of security (e.g., SHA-256).

Additionally, the four algorithms differ in terms of the size of the blocks and words of data that are used during hashing. Figure 1 presents the basic properties of all four secure hash algorithms.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)	Security[2] (bits)
SHA-1	$< 2^{64}$	512	32	160	80
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

Figure 1: Secure Hash Algorithm Properties

[1] The SHA-1 algorithm specified in this document is identical to the SHA-1 algorithm specified in FIPS 180-1 [180-1]. However, this specification, FIPS 180-2, uses $ROTL^n(X)$ instead of $S^n(X)$ [180-1] to denote "circular left shift by *n* bits" (i.e., "left rotation by *n* bits"). This is described in Sec. 3.2. Some other notational changes have been made in order to be consistent with the specifications for SHA-256, SHA-384, and SHA-512.

[2] In this context, "security" refers to the fact that a birthday attack [HAC] on a message digest of size *n* produces a collision with a workfactor of approximately $2^{n/2}$.

2. DEFINITIONS

2.1 Glossary of Terms and Acronyms

Bit	A binary digit having a value of 0 or 1.
Byte	A group of eight bits.
FIPS	Federal Information Processing Standard.
Word	A group of either 32 bits (4 bytes) or 64 bits (8 bytes), depending on the secure hash algorithm.

2.2 Algorithm Parameters, Symbols, and Terms

2.2.1 Parameters

The following parameters are used in the secure hash algorithm specifications in this standard.

$a, b, c, \ldots, h$	Working variables that are the w-bit words used in the computation of the hash values, $H^{(i)}$.
$H^{(i)}$	The i^{th} hash value. $H^{(0)}$ is the *initial* hash value; $H^{(N)}$ is the *final* hash value and is used to determine the message digest.
$H_j^{(i)}$	The j^{th} word of the i^{th} hash value, where $H_0^{(i)}$ is the left-most word of hash value i.
K_t	Constant value to be used for iteration t of the hash computation.
k	Number of zeroes appended to a message during the padding step.
ℓ	Length of the message, M, in bits.
m	Number of bits in a message block, $M^{(i)}$.
M	Message to be hashed.
$M^{(i)}$	Message block i, with a size of m bits.
$M_j^{(i)}$	The j^{th} word of the i^{th} message block, where $M_0^{(i)}$ is the left-most word of message block i.

n	Number of bits to be rotated or shifted when a word is operated upon.
N	Number of blocks in the padded message.
T	Temporary w-bit word used in the hash computation.
w	Number of bits in a word.
W_t	The t^{th} w-bit word of the message schedule.

2.2.2 Symbols

The following symbols are used in the secure hash algorithm specifications, and each operates on w-bit words.

$\wedge$	Bitwise AND operation.
$\vee$	Bitwise OR ("inclusive-OR") operation.
$\oplus$	Bitwise XOR ("exclusive-OR") operation.
$\neg$	Bitwise complement operation.
+	Addition modulo 2^w.
<<	Left-shift operation, where $x << n$ is obtained by discarding the left-most n bits of the word x and then padding the result with n zeroes on the right.
>>	Right-shift operation, where $x >> n$ is obtained by discarding the right-most n bits of the word x and then padding the result with n zeroes on the left.

3. NOTATION AND CONVENTIONS

3.1 Bit Strings and Integers

The following terminology related to bit strings and integers will be used.

1. A *hex digit* is an element of the set {`0, 1,…, 9, a,…, f`}. A hex digit is the representation of a 4-bit string. For example, the hex digit "`7`" represents the 4-bit string "`0111`", and the hex digit "`a`" represents the 4-bit string "`1010`".

2. A *word* is a *w*-bit string that may be represented as a sequence of hex digits. To convert a word to hex digits, each 4-bit string is converted to its hex digit equivalent, as described in (1) above. For example, the 32-bit string

   ```
   1010 0001 0000 0011 1111 1110 0010 0011
   ```

   can be expressed as "`a103fe23`", and the 64-bit string

   ```
   1010 0001 0000 0011 1111 1110 0010 0011
   0011 0010 1110 1111 0011 0000 0001 1010
   ```

   can be expressed as "`a103fe2332ef301a`".

   Throughout this specification, the "big-endian" convention is used when expressing both 32- and 64-bit words, so that within each word, the most significant bit is stored in the left-most bit position.

3. An *integer* may be represented as a word or pair of words. A word representation of the message length, ℓ, in bits, is required for the padding techniques of Sec. 5.1.

   An integer between 0 and 2^{32}-1 *inclusive* may be represented as a 32-bit word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. For example, the integer $291 = 2^8 + 2^5 + 2^1 + 2^0 = 256+32+2+1$ is represented by the hex word `00000123`.

   The same holds true for an integer between 0 and 2^{64}-1 *inclusive*, which may be represented as a 64-bit word.

   If Z is an integer, $0 \le Z < 2^{64}$, then $Z = 2^{32}X + Y$, where $0 \le X < 2^{32}$ and $0 \le Y < 2^{32}$. Since X and Y can be represented as 32-bit words x and y, respectively, the integer Z can be represented as the pair of words (x, y). This property is used for SHA-1 and SHA-256.

If Z is an integer, $0 \le Z < 2^{128}$, then $Z = 2^{64}X + Y$, where $0 \le X < 2^{64}$ and $0 \le Y < 2^{64}$. Since X and Y can be represented as 64-bit words x and y, respectively, the integer Z can be represented as the pair of words (x, y). This property is used for SHA-384 and SHA-512.

4. For the secure hash algorithms, the size of the *message block* - m bits - depends on the algorithm.

   a) For **SHA-1** and **SHA-256**, each message block has **512 bits**, which are represented as a sequence of sixteen **32-bit words**.

   b) For **SHA-384** and **SHA-512**, each message block has **1024 bits**, which are represented as a sequence of sixteen **64-bit words**.

3.2 Operations on Words

The following operations are applied to w-bit words in all four secure hash algorithms. SHA-1 and SHA-256 operate on 32-bit words ($w = 32$), and SHA-384 and SHA-512 operate on 64-bit words ($w = 64$).

1. Bitwise *logical* word operations: $\wedge$, $\vee$, $\oplus$, and $\neg$ (see Sec. 2.2.2).

2. Addition modulo 2^w.

   The operation $x + y$ is defined as follows. The words x and y represent integers X and Y, where $0 \le X < 2^w$ and $0 \le Y < 2^w$. For positive integers U and V, let $U \bmod V$ be the remainder upon dividing U by V. Compute

   $$Z = (X + Y) \bmod 2^w.$$

   Then $0 \le Z < 2^w$. Convert the integer Z to a word, z, and define $z = x + y$.

3. The *right shift* operation $\boldsymbol{SHR^n(x)}$, where x is a w-bit word and n is an integer with $0 \le n < w$, is defined by

   $$SHR^n(x) = x >> n.$$

   This operation is used in the SHA-256, SHA-384, and SHA-512 algorithms.

4. The *rotate right* (circular right shift) operation $\boldsymbol{ROTR^n(x)}$, where x is a w-bit word and n is an integer with $0 \le n < w$, is defined by

   $$ROTR^n(x) = (x >> n) \vee (x << w - n).$$

   Thus, $ROTR^n(x)$ is equivalent to a circular shift (rotation) of x by n positions to the right.

This operation is used by the SHA-256, SHA-384, and SHA-512 algorithms.

5. The *rotate left* (circular left shift) operation, $\boldsymbol{ROTL^n(x)}$, where x is a w-bit word and n is an integer with $0 \leq n < w$, is defined by

   $$ROTL^n(x) = (x << n) \vee (x >> w - n).$$

   Thus, $ROTL^n(x)$ is equivalent to a circular shift (rotation) of x by n positions to the left.

   This operation is used only in the SHA-1 algorithm. Note that in Ref. [180-1] this operation was referred to as "$S^n(X)$"; however, the notation has been modified for clarity and consistency with the notation used for operations in the other secure hash algorithms.

6. Note the following equivalence relationships, where w is fixed in each relationship:

   $$ROTL^n(x) \approx ROTR^{w-n}(x)$$

   $$ROTR^n(x) \approx ROTL^{w-n}(x).$$

4. FUNCTIONS AND CONSTANTS

4.1 Functions

This section defines the functions that are used by each of the algorithms. Although the SHA-256, SHA-384, and SHA-512 algorithms all use similar functions, their descriptions are separated into sections for SHA-256 (Sec. 4.1.2) and for SHA-384 and SHA-512 (Sec. 4.1.3), since the input and output for these functions are words of different sizes.

4.1.1 SHA-1 Functions

SHA-1 uses a sequence of logical functions, f_0, f_1,…, f_{79}. Each function f_t, where $0 \le t < 79$, operates on three 32-bit words, x, y, and z, and produces a 32-bit word as output. The function $f_t(x, y, z)$ is defined as follows:

$$f_t(x,y,z) = \begin{cases} (x \wedge y) \vee (\neg x \wedge z) & 0 \le t \le 19 \\ x \oplus y \oplus z & 20 \le t \le 39 \\ (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) & 40 \le t \le 59 \\ x \oplus y \oplus z & 60 \le t \le 79. \end{cases} \quad (4.1)$$

4.1.2 SHA-256 Functions

SHA-256 uses six logical functions, where *each function operates on 32-bit words*, which are represented as x, y, and z. The result of each function is a new 32-bit word.

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \quad (4.2)$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \quad (4.3)$$

$$\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \quad (4.4)$$

$$\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \quad (4.5)$$

$$s_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \quad (4.6)$$

$$s_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \quad (4.7)$$

4.1.3 SHA-384 and SHA-512 Functions

SHA-384 and SHA-512 each use six logical functions, where *each function operates on 64-bit words*, which are represented as x, y, and z. The result of each function is a new 64-bit word.

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \quad (4.8)$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \quad (4.9)$$

$$\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \quad (4.10)$$

$$\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \quad (4.11)$$

$$s_0^{\{512\}}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \quad (4.12)$$

$$s_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x) \quad (4.13)$$

4.2 Constants

4.2.1 SHA-1 Constants

SHA-1 uses a sequence of eighty constant 32-bit words, K_0, K_1,…, K_{79}, which are given by

$$K_t = \begin{cases} \texttt{5a827999} & 0 \le t \le 19 \\ \texttt{6ed9eba1} & 20 \le t \le 39 \\ \texttt{8f1bbcdc} & 40 \le t \le 59 \\ \texttt{ca62c1d6} & 60 \le t \le 79. \end{cases} \quad (4.14)$$

4.2.2 SHA-256 Constants

SHA-256 uses a sequence of sixty-four constant 32-bit words, $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex, these constant words are (from left to right)

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

4.2.3 SHA-384 and SHA-512 Constants

SHA-384 and SHA-512 use the same sequence of eighty constant 64-bit words, $K_0^{\{512\}}, K_1^{\{512\}}, \ldots, K_{79}^{\{512\}}$. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. In hex, these constant words are (from left to right)

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
```

```
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8
391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec
90befffa23631e28 a4506cebde82bde9 bef9a3f7b2c67915 c67178f2e372532b
ca273eceea26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178
06f067aa72176fba 0a637dc5a2c898a6 113f9804bef90dae 1b710b35131c471b
28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c
4cc5d4becb3e42b6 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817.
```

5. PREPROCESSING

Preprocessing shall take place before hash computation begins. This preprocessing consists of three steps: padding the message, M (Sec. 5.1), parsing the padded message into message blocks (Sec. 5.2), and setting the initial hash value, $H^{(0)}$ (Sec. 5.3).

5.1 Padding the Message

The message, M, shall be padded before hash computation begins. The purpose of this padding is to ensure that the padded message is a multiple of 512 or 1024 bits, depending on the algorithm.

5.1.1 SHA-1 and SHA-256

Suppose that the length of the message, M, is ℓ bits. Append the bit "1" to the end of the message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $\ell + 1 + k \equiv 448 \bmod 512$. Then append the 64-bit block that is equal to the number ℓ expressed using a binary representation. For example, the (8-bit ASCII) message "**abc**" has length $8 \times 3 = 24$, so the message is padded with a one bit, then $448 - (24 + 1) = 423$ zero bits, and then the message length, to become the 512-bit padded message

$$\underbrace{01100001}_{\text{“a”}}\ \underbrace{01100010}_{\text{“b”}}\ \underbrace{01100011}_{\text{“c”}}\ 1\ \overbrace{00\ldots00}^{423}\ \overbrace{00\ldots\underbrace{011000}_{\ell = 24}}^{64}.$$

The length of the padded message should now be a multiple of 512 bits.

5.1.2 SHA-384 and SHA-512

Suppose the length of the message M, in bits, is ℓ bits. Append the bit "1" to the end of the message, followed by k zero bits, where k is the smallest non-negative solution to the equation $\ell + 1 + k \equiv 896 \bmod 1024$. Then append the 128-bit block that is equal to the number ℓ expressed using a binary representation. For example, the (8-bit ASCII) message "**abc**" has length $8 \times 3 = 24$, so the message is padded with a one bit, then $896 - (24 + 1) = 871$ zero bits, and then the message length, to become the 1024-bit padded message

$$\underbrace{01100001}_{\text{“a”}}\ \underbrace{01100010}_{\text{“b”}}\ \underbrace{01100011}_{\text{“c”}}\ 1\ \overbrace{00\ldots00}^{871}\ \overbrace{00\ldots\underbrace{011000}_{\ell = 24}}^{128}.$$

The length of the padded message should now be a multiple of 1024 bits.

5.2 Parsing the Padded Message

After a message has been padded, it must be parsed into N m-bit blocks before the hash computation can begin.

5.2.1 SHA-1 and SHA-256

For SHA-1 and SHA-256, the padded message is parsed into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

5.2.2 SHA-384 and SHA-512

For SHA-384 and SHA-512, the padded message is parsed into N 1024-bit blocks, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$. Since the 1024 bits of the input block may be expressed as sixteen 64-bit words, the first 64 bits of message block i are denoted $M_0^{(i)}$, the next 64 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

5.3 Setting the Initial Hash Value ($H^{(0)}$)

Before hash computation begins for each of the secure hash algorithms, the initial hash value, $H^{(0)}$, must be set. The size and number of words in $H^{(0)}$ depends on the message digest size.

5.3.1 SHA-1

For SHA-1, the initial hash value, $H^{(0)}$, shall consist of the following five 32-bit words, in hex:

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

5.3.2 SHA-256

For SHA-256, the initial hash value, $H^{(0)}$, shall consist of the following eight 32-bit words, in hex:

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

These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

5.3.3 SHA-384

For SHA-384, the initial hash value, $H^{(0)}$, shall consist of the following eight 64-bit words, in hex:

$$H_0^{(0)} = \texttt{cbbb9d5dc1059ed8}$$
$$H_1^{(0)} = \texttt{629a292a367cd507}$$
$$H_2^{(0)} = \texttt{9159015a3070dd17}$$
$$H_3^{(0)} = \texttt{152fecd8f70e5939}$$
$$H_4^{(0)} = \texttt{67332667ffc00b31}$$
$$H_5^{(0)} = \texttt{8eb44a8768581511}$$
$$H_6^{(0)} = \texttt{db0c2e0d64f98fa7}$$
$$H_7^{(0)} = \texttt{47b5481dbefa4fa4}.$$

These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers.

5.3.4 SHA-512

For SHA-512, the initial hash value, $H^{(0)}$, shall consist of the following eight 64-bit words, in hex:

$$H_0^{(0)} = \texttt{6a09e667f3bcc908}$$
$$H_1^{(0)} = \texttt{bb67ae8584caa73b}$$
$$H_2^{(0)} = \texttt{3c6ef372fe94f82b}$$
$$H_3^{(0)} = \texttt{a54ff53a5f1d36f1}$$
$$H_4^{(0)} = \texttt{510e527fade682d1}$$
$$H_5^{(0)} = \texttt{9b05688c2b3e6c1f}$$
$$H_6^{(0)} = \texttt{1f83d9abfb41bd6b}$$
$$H_7^{(0)} = \texttt{5be0cd19137e2179}.$$

These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

6. SECURE HASH ALGORITHMS

In the following sections, SHA-512 is described before SHA-384. That is because the SHA-384 algorithm is identical to SHA-512, with the exception of using a different initial hash value and truncating the final hash value to 384 bits.

For each of the secure hash algorithms, there may exist alternate computation methods that yield identical results; one example is the alternative SHA-1 computation described in Sec. 6.1.3. Such alternate methods may be implemented in conformance to this standard.

6.1 SHA-1

SHA-1 may be used to hash a message, M, having a length of ℓ bits, where $0 \le \ell < 2^{64}$. The algorithm uses 1) a message schedule of eighty 32-bit words, 2) five working variables of 32 bits each, and 3) a hash value of five 32-bit words. The final result of SHA-1 is a 160-bit message digest.

The words of the message schedule are labeled W_0, W_1,…, W_{79}. The five working variables are labeled ***a***, ***b***, ***c***, ***d***, and ***e***. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, \ldots, H_4^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-1 also uses a single temporary word, T.

Appendix A gives several detailed examples of SHA-1.

6.1.1 SHA-1 Preprocessing

1. Pad the message, M, according to Sec. 5.1.1;

2. Parse the padded message into N 512-bit message blocks, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, according to Sec. 5.2.1; and

3. Set the initial hash value, $H^{(0)}$, as specified in Sec. 5.3.1.

6.1.2 SHA-1 Hash Computation

The SHA-1 hash computation uses functions and constants previously defined in Sec. 4.1.1 and Sec. 4.2.1, respectively. Addition (+) is performed modulo 2^{32}.

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, is processed in order, using the following steps:

For i = 1 to N:
{

1. Prepare the message schedule, $\{W_t\}$:

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

2. Initialize the five working variables, ***a***, ***b***, ***c***, ***d***, and ***e***, with the (i-1)st hash value:

$$a = H_0^{(i)}$$
$$b = H_1^{(i)}$$
$$c = H_2^{(i)}$$
$$d = H_3^{(i)}$$
$$e = H_4^{(i)}$$

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

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

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

}

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 160-bit message digest of the message, M, is

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

6.1.3 Alternate Method for Computing a SHA-1 Message Digest

The SHA-1 hash computation method described in Sec. 6.1.2 assumes that the message schedule W_0, W_1,…, W_{79} is implemented as an array of eighty 32-bit words. This is efficient from the standpoint of the minimization of execution time, since the addresses of W_{t-3},…, W_{t-16} in step (2) of Sec. 6.1.2 are easily computed.

However, if memory is limited, an alternative is to regard $\{W_t\}$ as a circular queue that may be implemented using an array of sixteen 32-bit words, W_0, W_1,…, W_{15}. The alternate method that is described in this section yields the same message digest as the SHA-1 computation method described in Sec. 6.1.2. Although this alternate method saves sixty-four 32-bit words of storage, it is likely to lengthen the execution time due to the increased complexity of the address computations for the $\{W_t\}$ in step (3).

For this alternate SHA-1 method, let *MASK* = `0000000f` (in hex). As in Sec. 6.1.1, addition is performed modulo 2^{32}. Assuming that the preprocessing as described in Sec. 6.1.1 has been performed, the processing of $M^{(i)}$ is as follows:

For $i = 1$ to N:
{

1. For $t = 0$ to 15:
   {
   $W_t = M_t^{(i)}$
   }

2. Initialize the five working variables, ***a***, ***b***, ***c***, ***d***, and ***e***, with the $(i-1)^{st}$ hash value:

   $a = H_0^{(i)}$
   $b = H_1^{(i)}$
   $c = H_2^{(i)}$
   $d = H_3^{(i)}$
   $e = H_4^{(i)}$

3. For $t = 0$ to 79:
   {
   $s = t \wedge MASK$

   If $t \geq 16$ then
   {
   $W_s = ROTL^1(W_{(s+13)\wedge MASK} \oplus W_{(s+8)\wedge MASK} \oplus W_{(s+2)\wedge MASK} \oplus W_s)$
   }

$$T = ROTL^{5}(a) + f_t(b,c,d) + e + K_t + W_s$$
$$e = d$$
$$d = c$$
$$c = ROTL^{30}(b)$$
$$b = a$$
$$a = T$$

}

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

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

}

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 160-bit message digest of the message, M, is

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

6.2 SHA-256

SHA-256 may be used to hash a message, M, having a length of ℓ bits, where $0 \le \ell < 2^{64}$. The algorithm uses 1) a message schedule of sixty-four 32-bit words, 2) eight working variables of 32 bits each, and 3) a hash value of eight 32-bit words. The final result of SHA-256 is a 256-bit message digest.

The words of the message schedule are labeled W_0, W_1,…, W_{63}. The eight working variables are labeled ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, \ldots, H_7^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-256 also uses two temporary words, T_1 and T_2.

Appendix B gives several detailed examples of SHA-256.

6.2.1 SHA-256 Preprocessing

1. Pad the message, M, according to Sec. 5.1.1;

2. Parse the padded message into N 512-bit message blocks, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, according to Sec. 5.2.1; and

3. Set the initial hash value, $H^{(0)}$, as specified in Sec. 5.3.2.

6.2.2 SHA-256 Hash Computation

The SHA-256 hash computation uses functions and constants previously defined in Sec. 4.1.2 and Sec. 4.2.2, respectively. Addition (+) is performed modulo 2^{32}.

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, is processed in order, using the following steps:

For i = 1 to N:
{

1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ s_1^{\{256\}}(W_{t-2}) + W_{t-7} + s_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

2. Initialize the eight working variables, $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{e}$, $\boldsymbol{f}$, $\boldsymbol{g}$, and $\boldsymbol{h}$, with the (i-1)st hash value:

$$a = H_0^{(i)}$$
$$b = H_1^{(i)}$$
$$c = H_2^{(i)}$$
$$d = H_3^{(i)}$$
$$e = H_4^{(i)}$$
$$f = H_5^{(i)}$$
$$g = H_6^{(i)}$$
$$h = H_7^{(i)}$$

3. For t = 0 to 63:
{

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$$
$$T_2 = \sum_0^{\{256\}}(a) + Maj(a, b, c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

}

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

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

}

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, M, is

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

6.3 SHA-512

SHA-512 may be used to hash a message, M, having a length of ℓ bits, where $0 \le \ell < 2^{128}$. The algorithm uses 1) a message schedule of eighty 64-bit words, 2) eight working variables of 64 bits each, and 3) a hash value of eight 64-bit words. The final result of SHA-512 is a 512-bit message digest.

The words of the message schedule are labeled W_0, W_1,…, W_{79}. The eight working variables are labeled ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, \ldots, H_7^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value

(after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-512 also uses two temporary words, T_1 and T_2.

Appendix C gives several detailed examples of SHA-512.

6.3.1 SHA-512 Preprocessing

1. Pad the message, M, according to Sec. 5.1.2;

2. Parse the padded message into N 1024-bit message blocks, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, according to Sec. 5.2.2; and

3. Set the initial hash value, $H^{(0)}$, as specified in Sec. 5.3.4.

6.3.2 SHA-512 Hash Computation

The SHA-512 hash computation uses functions and constants previously defined in Sec. 4.1.3 and Sec. 4.2.3, respectively. Addition (+) is performed modulo 2^{64}.

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, is processed in order, using the following steps:

For $i = 1$ to N:
{

1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ s_1^{\{512\}}(W_{t-2}) + W_{t-7} + s_0^{\{512\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 79 \end{cases}$$

2. Initialize the eight working variables, ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***, with the $(i\text{-}1)^{st}$ hash value:

$$a = H_0^{(i)}$$
$$b = H_1^{(i)}$$
$$c = H_2^{(i)}$$
$$d = H_3^{(i)}$$
$$e = H_4^{(i)}$$
$$f = H_5^{(i)}$$
$$g = H_6^{(i)}$$
$$h = H_7^{(i)}$$

3. For $t = 0$ to 79:

{

$T_1 = h + \sum_1^{\{512\}}(e) + Ch(e, f, g) + K_t^{\{512\}} + W_t$

$T_2 = \sum_0^{\{512\}}(a) + Maj(a, b, c)$

$h = g$

$g = f$

$f = e$

$e = d + T_1$

$d = c$

$c = b$

$b = a$

$a = T_1 + T_2$

}

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

$H_0^{(i)} = a + H_0^{(i-1)}$

$H_1^{(i)} = b + H_1^{(i-1)}$

$H_2^{(i)} = c + H_2^{(i-1)}$

$H_3^{(i)} = d + H_3^{(i-1)}$

$H_4^{(i)} = e + H_4^{(i-1)}$

$H_5^{(i)} = f + H_5^{(i-1)}$

$H_6^{(i)} = g + H_6^{(i-1)}$

$H_7^{(i)} = h + H_7^{(i-1)}$

}

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 512-bit message digest of the message, M, is

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

6.4 SHA-384

SHA-384 may be used to hash a message, M, having a length of ℓ bits, where $0 \le \ell < 2^{128}$. The algorithm is defined in the exact same manner as SHA-512 (Sec. 6.3), with the following two exceptions:

1. The initial hash value, $H^{(0)}$, shall be set as specified in Sec. 5.3.3; and

2. The 384-bit message digest is obtained by truncating the final hash value, $H^{(N)}$, to its left-most 384 bits:

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

Appendix D gives several detailed examples of SHA-384.

APPENDIX A: SHA-1 EXAMPLES

This appendix is for informational purposes only and is not required to meet the standard.

A.1 SHA-1 Example (One-Block Message)

Let the message, M, be the 24-bit ($\ell = 24$) ASCII string "**abc**", which is equivalent to the following binary string:

`01100001 01100010 01100011`.

The message is padded by appending a "`1`" bit, followed by 423 "`0`" bits, and ending with the hex value `00000000 00000018` (the two 32-bit word representation of the length, 24). Thus, the final padded message consists of one block ($N = 1$).

For SHA-1, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = `67452301`
$H_1^{(0)}$ = `efcdab89`
$H_2^{(0)}$ = `98badcfe`
$H_3^{(0)}$ = `10325476`
$H_4^{(0)}$ = `c3d2e1f0`.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = `61626380`
W_1 = `00000000`
W_2 = `00000000`
W_3 = `00000000`
W_4 = `00000000`
W_5 = `00000000`
W_6 = `00000000`
W_7 = `00000000`
W_8 = `00000000`
W_9 = `00000000`
W_{10} = `00000000`
W_{11} = `00000000`
W_{12} = `00000000`
W_{13} = `00000000`
W_{14} = `00000000`
W_{15} = `00000018`.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, and ***e*** *after* pass t of the "for $t = 0$ to 79" loop described in Sec. 6.1.2, step 4.

	a	***b***	***c***	***d***	***e***
$t = 0$:	`0116fc33`	`67452301`	`7bf36ae2`	`98badcfe`	`10325476`
$t = 1$:	`8990536d`	`0116fc33`	`59d148c0`	`7bf36ae2`	`98badcfe`
$t = 2$:	`a1390f08`	`8990536d`	`c045bf0c`	`59d148c0`	`7bf36ae2`

$t = 3$:	cdd8e11b	a1390f08	626414db	c045bf0c	59d148c0
$t = 4$:	cfd499de	cdd8e11b	284e43c2	626414db	c045bf0c
$t = 5$:	3fc7ca40	cfd499de	f3763846	284e43c2	626414db
$t = 6$:	993e30c1	3fc7ca40	b3f52677	f3763846	284e43c2
$t = 7$:	9e8c07d4	993e30c1	0ff1f290	b3f52677	f3763846
$t = 8$:	4b6ae328	9e8c07d4	664f8c30	0ff1f290	b3f52677
$t = 9$:	8351f929	4b6ae328	27a301f5	664f8c30	0ff1f290
$t = 10$:	fbda9e89	8351f929	12dab8ca	27a301f5	664f8c30
$t = 11$:	63188fe4	fbda9e89	60d47e4a	12dab8ca	27a301f5
$t = 12$:	4607b664	63188fe4	7ef6a7a2	60d47e4a	12dab8ca
$t = 13$:	9128f695	4607b664	18c623f9	7ef6a7a2	60d47e4a
$t = 14$:	196bee77	9128f695	1181ed99	18c623f9	7ef6a7a2
$t = 15$:	20bdd62f	196bee77	644a3da5	1181ed99	18c623f9
$t = 16$:	4e925823	20bdd62f	c65afb9d	644a3da5	1181ed99
$t = 17$:	82aa6728	4e925823	c82f758b	c65afb9d	644a3da5
$t = 18$:	dc64901d	82aa6728	d3a49608	c82f758b	c65afb9d
$t = 19$:	fd9e1d7d	dc64901d	20aa99ca	d3a49608	c82f758b
$t = 20$:	1a37b0ca	fd9e1d7d	77192407	20aa99ca	d3a49608
$t = 21$:	33a23bfc	1a37b0ca	7f67875f	77192407	20aa99ca
$t = 22$:	21283486	33a23bfc	868dec32	7f67875f	77192407
$t = 23$:	d541f12d	21283486	0ce88eff	868dec32	7f67875f
$t = 24$:	c7567dc6	d541f12d	884a0d21	0ce88eff	868dec32
$t = 25$:	48413ba4	c7567dc6	75507c4b	884a0d21	0ce88eff
$t = 26$:	be35fbd5	48413ba4	b1d59f71	75507c4b	884a0d21
$t = 27$:	4aa84d97	be35fbd5	12104ee9	b1d59f71	75507c4b
$t = 28$:	8370b52e	4aa84d97	6f8d7ef5	12104ee9	b1d59f71
$t = 29$:	c5fbaf5d	8370b52e	d2aa1365	6f8d7ef5	12104ee9
$t = 30$:	1267b407	c5fbaf5d	a0dc2d4b	d2aa1365	6f8d7ef5
$t = 31$:	3b845d33	1267b407	717eebd7	a0dc2d4b	d2aa1365
$t = 32$:	046faa0a	3b845d33	c499ed01	717eebd7	a0dc2d4b
$t = 33$:	2c0ebc11	046faa0a	cee1174c	c499ed01	717eebd7
$t = 34$:	21796ad4	2c0ebc11	811bea82	cee1174c	c499ed01
$t = 35$:	dcbbb0cb	21796ad4	4b03af04	811bea82	cee1174c
$t = 36$:	0f511fd8	dcbbb0cb	085e5ab5	4b03af04	811bea82
$t = 37$:	dc63973f	0f511fd8	f72eec32	085e5ab5	4b03af04
$t = 38$:	4c986405	dc63973f	03d447f6	f72eec32	085e5ab5
$t = 39$:	32de1cba	4c986405	f718e5cf	03d447f6	f72eec32
$t = 40$:	fc87dedf	32de1cba	53261901	f718e5cf	03d447f6
$t = 41$:	970a0d5c	fc87dedf	8cb7872e	53261901	f718e5cf
$t = 42$:	7f193dc5	970a0d5c	ff21f7b7	8cb7872e	53261901
$t = 43$:	ee1b1aaf	7f193dc5	25c28357	ff21f7b7	8cb7872e
$t = 44$:	40f28e09	ee1b1aaf	5fc64f71	25c28357	ff21f7b7
$t = 45$:	1c51e1f2	40f28e09	fb86c6ab	5fc64f71	25c28357
$t = 46$:	a01b846c	1c51e1f2	503ca382	fb86c6ab	5fc64f71
$t = 47$:	bead02ca	a01b846c	8714787c	503ca382	fb86c6ab
$t = 48$:	baf39337	bead02ca	2806e11b	8714787c	503ca382
$t = 49$:	120731c5	baf39337	afab40b2	2806e11b	8714787c
$t = 50$:	641db2ce	120731c5	eebce4cd	afab40b2	2806e11b
$t = 51$:	3847ad66	641db2ce	4481cc71	eebce4cd	afab40b2
$t = 52$:	e490436d	3847ad66	99076cb3	4481cc71	eebce4cd
$t = 53$:	27e9f1d8	e490436d	8e11eb59	99076cb3	4481cc71
$t = 54$:	7b71f76d	27e9f1d8	792410db	8e11eb59	99076cb3
$t = 55$:	5e6456af	7b71f76d	09fa7c76	792410db	8e11eb59
$t = 56$:	c846093f	5e6456af	5edc7ddb	09fa7c76	792410db
$t = 57$:	d262ff50	c846093f	d79915ab	5edc7ddb	09fa7c76
$t = 58$:	09d785fd	d262ff50	f211824f	d79915ab	5edc7ddb

$t = 59$:	3f52de5a	09d785fd	3498bfd4	f211824f	d79915ab
$t = 60$:	d756c147	3f52de5a	4275e17f	3498bfd4	f211824f
$t = 61$:	548c9cb2	d756c147	8fd4b796	4275e17f	3498bfd4
$t = 62$:	b66c020b	548c9cb2	f5d5b051	8fd4b796	4275e17f
$t = 63$:	6b61c9e1	b66c020b	9523272c	f5d5b051	8fd4b796
$t = 64$:	19dfa7ac	6b61c9e1	ed9b0082	9523272c	f5d5b051
$t = 65$:	101655f9	19dfa7ac	5ad87278	ed9b0082	9523272c
$t = 66$:	0c3df2b4	101655f9	0677e9eb	5ad87278	ed9b0082
$t = 67$:	78dd4d2b	0c3df2b4	4405957e	0677e9eb	5ad87278
$t = 68$:	497093c0	78dd4d2b	030f7cad	4405957e	0677e9eb
$t = 69$:	3f2588c2	497093c0	de37534a	030f7cad	4405957e
$t = 70$:	c199f8c7	3f2588c2	125c24f0	de37534a	030f7cad
$t = 71$:	39859de7	c199f8c7	8fc96230	125c24f0	de37534a
$t = 72$:	edb42de4	39859de7	f0667e31	8fc96230	125c24f0
$t = 73$:	11793f6f	edb42de4	ce616779	f0667e31	8fc96230
$t = 74$:	5ee76897	11793f6f	3b6d0b79	ce616779	f0667e31
$t = 75$:	63f7dab7	5ee76897	c45e4fdb	3b6d0b79	ce616779
$t = 76$:	a079b7d9	63f7dab7	d7b9da25	c45e4fdb	3b6d0b79
$t = 77$:	860d21cc	a079b7d9	d8fdf6ad	d7b9da25	c45e4fdb
$t = 78$:	5738d5e1	860d21cc	681e6df6	d8fdf6ad	d7b9da25
$t = 79$:	42541b35	5738d5e1	21834873	681e6df6	d8fdf6ad

That completes the processing of the first and only message block, $M^{(1)}$. The final hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{67452301} + \texttt{42541b35} = \texttt{a9993e36}$$
$$H_1^{(1)} = \texttt{efcdab89} + \texttt{5738d5e1} = \texttt{4706816a}$$
$$H_2^{(1)} = \texttt{98badcfe} + \texttt{21834873} = \texttt{ba3e2571}$$
$$H_3^{(1)} = \texttt{10325476} + \texttt{681e6df6} = \texttt{7850c26c}$$
$$H_4^{(1)} = \texttt{c3d2e1f0} + \texttt{d8fdf6ad} = \texttt{9cd0d89d}.$$

The resulting 160-bit message digest is

```
a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d.
```

A.2 SHA-1 Example (Multi-Block Message)

Let the message, *M*, be the 448-bit ($\ell = 448$) ASCII string

"**abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq**".

The message is padded by appending a "`1`" bit, followed by 511 "`0`" bits, and ending with the hex value `00000000 000001c0` (the two 32-bit word representation of the length, 448). Thus, the final padded message consists of two blocks ($N = 2$).

For SHA-1, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = 67452301
$H_1^{(0)}$ = efcdab89
$H_2^{(0)}$ = 98badcfe
$H_3^{(0)}$ = 10325476
$H_4^{(0)}$ = c3d2e1f0.

The words of the first padded message block, $M^{(1)}$, are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 61626364
W_1 = 62636465
W_2 = 63646566
W_3 = 64656667
W_4 = 65666768
W_5 = 66676869
W_6 = 6768696a
W_7 = 68696a6b
W_8 = 696a6b6c
W_9 = 6a6b6c6d
W_{10} = 6b6c6d6e
W_{11} = 6c6d6e6f
W_{12} = 6d6e6f70
W_{13} = 6e6f7071
W_{14} = 80000000
W_{15} = 00000000.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, and ***e*** *after* pass t of the "for t = 0 to 79" loop described in Sec. 6.1.2, step 4.

	a	***b***	***c***	***d***	***e***
t = 0:	0116fc17	67452301	7bf36ae2	98badcfe	10325476
t = 1:	ebf3b452	0116fc17	59d148c0	7bf36ae2	98badcfe
t = 2:	5109913a	ebf3b452	c045bf05	59d148c0	7bf36ae2
t = 3:	2c4f6eac	5109913a	bafced14	c045bf05	59d148c0
t = 4:	33f4ae5b	2c4f6eac	9442644e	bafced14	c045bf05
t = 5:	96b85189	33f4ae5b	0b13dbab	9442644e	bafced14
t = 6:	db04cb58	96b85189	ccfd2b96	0b13dbab	9442644e
t = 7:	45833f0f	db04cb58	65ae1462	ccfd2b96	0b13dbab
t = 8:	c565c35e	45833f0f	36c132d6	65ae1462	ccfd2b96
t = 9:	6350afda	c565c35e	d160cfc3	36c132d6	65ae1462
t = 10:	8993ea77	6350afda	b15970d7	d160cfc3	36c132d6
t = 11:	e19ecaa2	8993ea77	98d42bf6	b15970d7	d160cfc3
t = 12:	8603481e	e19ecaa2	e264fa9d	98d42bf6	b15970d7
t = 13:	32f94a85	8603481e	b867b2a8	e264fa9d	98d42bf6
t = 14:	b2e7a8be	32f94a85	a180d207	b867b2a8	e264fa9d
t = 15:	42637e39	b2e7a8be	4cbe52a1	a180d207	b867b2a8
t = 16:	6b068048	42637e39	acb9ea2f	4cbe52a1	a180d207
t = 17:	426b9c35	6b068048	5098df8e	acb9ea2f	4cbe52a1
t = 18:	944b1bd1	426b9c35	1ac1a012	5098df8e	acb9ea2f
t = 19:	6c445652	944b1bd1	509ae70d	1ac1a012	5098df8e
t = 20:	95836da5	6c445652	6512c6f4	509ae70d	1ac1a012
t = 21:	09511177	95836da5	9b111594	6512c6f4	509ae70d
t = 22:	e2b92dc4	09511177	6560db69	9b111594	6512c6f4
t = 23:	fd224575	e2b92dc4	c254445d	6560db69	9b111594
t = 24:	eeb82d9a	fd224575	38ae4b71	c254445d	6560db69
t = 25:	5a142c1a	eeb82d9a	7f48915d	38ae4b71	c254445d

$t = 26$:	2972f7c7	5a142c1a	bbae0b66	7f48915d	38ae4b71
$t = 27$:	d526a644	2972f7c7	96850b06	bbae0b66	7f48915d
$t = 28$:	e1122421	d526a644	ca5cbdf1	96850b06	bbae0b66
$t = 29$:	05b457b2	e1122421	3549a991	ca5cbdf1	96850b06
$t = 30$:	a9c84bec	05b457b2	78448908	3549a991	ca5cbdf1
$t = 31$:	52e31f60	a9c84bec	816d15ec	78448908	3549a991
$t = 32$:	5af3242c	52e31f60	2a7212fb	816d15ec	78448908
$t = 33$:	31c756a9	5af3242c	14b8c7d8	2a7212fb	816d15ec
$t = 34$:	e9ac987c	31c756a9	16bcc90b	14b8c7d8	2a7212fb
$t = 35$:	ab7c32ee	e9ac987c	4c71d5aa	16bcc90b	14b8c7d8
$t = 36$:	5933fc99	ab7c32ee	3a6b261f	4c71d5aa	16bcc90b
$t = 37$:	43f87ae9	5933fc99	aadf0cbb	3a6b261f	4c71d5aa
$t = 38$:	24957f22	43f87ae9	564cff26	aadf0cbb	3a6b261f
$t = 39$:	adeb7478	24957f22	50fe1eba	564cff26	aadf0cbb
$t = 40$:	d70e5010	adeb7478	89255fc8	50fe1eba	564cff26
$t = 41$:	79bcfb08	d70e5010	2b7add1e	89255fc8	50fe1eba
$t = 42$:	f9bcb8de	79bcfb08	35c39404	2b7add1e	89255fc8
$t = 43$:	633e9561	f9bcb8de	1e6f3ec2	35c39404	2b7add1e
$t = 44$:	98c1ea64	633e9561	be6f2e37	1e6f3ec2	35c39404
$t = 45$:	c6ea241e	98c1ea64	58cfa558	be6f2e37	1e6f3ec2
$t = 46$:	a2ad4f02	c6ea241e	26307a99	58cfa558	be6f2e37
$t = 47$:	c8a69090	a2ad4f02	b1ba8907	26307a99	58cfa558
$t = 48$:	88341600	c8a69090	a8ab53c0	b1ba8907	26307a99
$t = 49$:	7e846f58	88341600	3229a424	a8ab53c0	b1ba8907
$t = 50$:	86e358ba	7e846f58	220d0580	3229a424	a8ab53c0
$t = 51$:	8d2e76c8	86e358ba	1fa11bd6	220d0580	3229a424
$t = 52$:	ce892e10	8d2e76c8	a1b8d62e	1fa11bd6	220d0580
$t = 53$:	edea95b1	ce892e10	234b9db2	a1b8d62e	1fa11bd6
$t = 54$:	36d1230a	edea95b1	33a24b84	234b9db2	a1b8d62e
$t = 55$:	776c3910	36d1230a	7b7aa56c	33a24b84	234b9db2
$t = 56$:	a681b723	776c3910	8db448c2	7b7aa56c	33a24b84
$t = 57$:	ac0a794f	a681b723	1ddb0e44	8db448c2	7b7aa56c
$t = 58$:	f03d3782	ac0a794f	e9a06dc8	1ddb0e44	8db448c2
$t = 59$:	9ef775c3	f03d3782	eb029e53	e9a06dc8	1ddb0e44
$t = 60$:	36254b13	9ef775c3	bc0f4de0	eb029e53	e9a06dc8
$t = 61$:	4080d4dc	36254b13	e7bddd70	bc0f4de0	eb029e53
$t = 62$:	2bfaf7a8	4080d4dc	cd8952c4	e7bddd70	bc0f4de0
$t = 63$:	513f9ca0	2bfaf7a8	10203537	cd8952c4	e7bddd70
$t = 64$:	e5895c81	513f9ca0	0afebdea	10203537	cd8952c4
$t = 65$:	1037d2d5	e5895c81	144fe728	0afebdea	10203537
$t = 66$:	14a82da9	1037d2d5	79625720	144fe728	0afebdea
$t = 67$:	6d17c9fd	14a82da9	440df4b5	79625720	144fe728
$t = 68$:	2c7b07bd	6d17c9fd	452a0b6a	440df4b5	79625720
$t = 69$:	fdf6efff	2c7b07bd	5b45f27f	452a0b6a	440df4b5
$t = 70$:	112b96e3	fdf6efff	4b1ec1ef	5b45f27f	452a0b6a
$t = 71$:	84065712	112b96e3	ff7dbbff	4b1ec1ef	5b45f27f
$t = 72$:	ab89fb71	84065712	c44ae5b8	ff7dbbff	4b1ec1ef
$t = 73$:	c5210e35	ab89fb71	a10195c4	c44ae5b8	ff7dbbff
$t = 74$:	352d9f4b	c5210e35	6ae27edc	a10195c4	c44ae5b8
$t = 75$:	1a0e0e0a	352d9f4b	7148438d	6ae27edc	a10195c4
$t = 76$:	d0d47349	1a0e0e0a	cd4b67d2	7148438d	6ae27edc
$t = 77$:	ad38620d	d0d47349	86838382	cd4b67d2	7148438d
$t = 78$:	d3ad7c25	ad38620d	74351cd2	86838382	cd4b67d2
$t = 79$:	8ce34517	d3ad7c25	6b4e1883	74351cd2	86838382

That completes the processing of the first message block, $M^{(1)}$. The first intermediate hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{67452301} + \texttt{8ce34517} = \texttt{f4286818}$$
$$H_1^{(1)} = \texttt{efcdab89} + \texttt{d3ad7c25} = \texttt{c37b27ae}$$
$$H_2^{(1)} = \texttt{98badcfe} + \texttt{6b4e1883} = \texttt{0408f581}$$
$$H_3^{(1)} = \texttt{10325476} + \texttt{74351cd2} = \texttt{84677148}$$
$$H_4^{(1)} = \texttt{c3d2e1f0} + \texttt{86838382} = \texttt{4a566572}.$$

The words of the *second* padded message block, $M^{(2)}$, are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 00000000
W_1 = 00000000
W_2 = 00000000
W_3 = 00000000
W_4 = 00000000
W_5 = 00000000
W_6 = 00000000
W_7 = 00000000
W_8 = 00000000
W_9 = 00000000
W_{10} = 00000000
W_{11} = 00000000
W_{12} = 00000000
W_{13} = 00000000
W_{14} = 00000000
W_{15} = 000001c0.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, and ***e*** *after* pass t of the "for t = 0 to 79" loop described in Sec. 6.1.2, step 4.

	a	***b***	***c***	***d***	***e***
t = 0:	2df257e9	f4286818	b0dec9eb	0408f581	84677148
t = 1:	4d3dc58f	2df257e9	3d0a1a06	b0dec9eb	0408f581
t = 2:	c352bb05	4d3dc58f	4b7c95fa	3d0a1a06	b0dec9eb
t = 3:	eef743c6	c352bb05	d34f7163	4b7c95fa	3d0a1a06
t = 4:	41e34277	eef743c6	70d4aec1	d34f7163	4b7c95fa
t = 5:	5443915c	41e34277	bbbdd0f1	70d4aec1	d34f7163
t = 6:	e7fa0377	5443915c	d078d09d	bbbdd0f1	70d4aec1
t = 7:	c6946813	e7fa0377	1510e457	d078d09d	bbbdd0f1
t = 8:	fdde1de1	c6946813	f9fe80dd	1510e457	d078d09d
t = 9:	b8538aca	fdde1de1	f1a51a04	f9fe80dd	1510e457
t = 10:	6ba94f63	b8538aca	7f778778	f1a51a04	f9fe80dd
t = 11:	43a2792f	6ba94f63	ae14e2b2	7f778778	f1a51a04
t = 12:	fecd7bbf	43a2792f	daea53d8	ae14e2b2	7f778778
t = 13:	a2604ca8	fecd7bbf	d0e89e4b	daea53d8	ae14e2b2
t = 14:	258b0baa	a2604ca8	ffb35eef	d0e89e4b	daea53d8
t = 15:	d9772360	258b0baa	2898132a	ffb35eef	d0e89e4b
t = 16:	5507db6e	d9772360	8962c2ea	2898132a	ffb35eef
t = 17:	a51b58bc	5507db6e	365dc8d8	8962c2ea	2898132a
t = 18:	c2eb709f	a51b58bc	9541f6db	365dc8d8	8962c2ea
t = 19:	d8992153	c2eb709f	2946d62f	9541f6db	365dc8d8
t = 20:	37482f5f	d8992153	f0badc27	2946d62f	9541f6db
t = 21:	ee8700bd	37482f5f	f6264854	f0badc27	2946d62f

$t = 22$:	9ad594b9	ee8700bd	cdd20bd7	f6264854	f0badc27
$t = 23$:	8fbaa5b9	9ad594b9	7ba1c02f	cdd20bd7	f6264854
$t = 24$:	88fb5867	8fbaa5b9	66b5652e	7ba1c02f	cdd20bd7
$t = 25$:	eec50521	88fb5867	63eea96e	66b5652e	7ba1c02f
$t = 26$:	50bce434	eec50521	e23ed619	63eea96e	66b5652e
$t = 27$:	5c416daf	50bce434	7bb14148	e23ed619	63eea96e
$t = 28$:	2429be5f	5c416daf	142f390d	7bb14148	e23ed619
$t = 29$:	0a2fb108	2429be5f	d7105b6b	142f390d	7bb14148
$t = 30$:	17986223	0a2fb108	c90a6f97	d7105b6b	142f390d
$t = 31$:	8a4af384	17986223	028bec42	c90a6f97	d7105b6b
$t = 32$:	6b629993	8a4af384	c5e61888	028bec42	c90a6f97
$t = 33$:	f15f04f3	6b629993	2292bce1	c5e61888	028bec42
$t = 34$:	295cc25b	f15f04f3	dad8a664	2292bce1	c5e61888
$t = 35$:	696da404	295cc25b	fc57c13c	dad8a664	2292bce1
$t = 36$:	cef5ae12	696da404	ca573096	fc57c13c	dad8a664
$t = 37$:	87d5b80c	cef5ae12	1a5b6901	ca573096	fc57c13c
$t = 38$:	84e2a5f2	87d5b80c	b3bd6b84	1a5b6901	ca573096
$t = 39$:	03bb6310	84e2a5f2	21f56e03	b3bd6b84	1a5b6901
$t = 40$:	c2d8f75f	03bb6310	a138a97c	21f56e03	b3bd6b84
$t = 41$:	bfb25768	c2d8f75f	00eed8c4	a138a97c	21f56e03
$t = 42$:	28589152	bfb25768	f0b63dd7	00eed8c4	a138a97c
$t = 43$:	ec1d3d61	28589152	2fec95da	f0b63dd7	00eed8c4
$t = 44$:	3caed7af	ec1d3d61	8a162454	2fec95da	f0b63dd7
$t = 45$:	c3d033ea	3caed7af	7b074f58	8a162454	2fec95da
$t = 46$:	7316056a	c3d033ea	cf2bb5eb	7b074f58	8a162454
$t = 47$:	46f93b68	7316056a	b0f40cfa	cf2bb5eb	7b074f58
$t = 48$:	dc8e7f26	46f93b68	9cc5815a	b0f40cfa	cf2bb5eb
$t = 49$:	850d411c	dc8e7f26	11be4eda	9cc5815a	b0f40cfa
$t = 50$:	7e4672c0	850d411c	b7239fc9	11be4eda	9cc5815a
$t = 51$:	89fbd41d	7e4672c0	21435047	b7239fc9	11be4eda
$t = 52$:	1797e228	89fbd41d	1f919cb0	21435047	b7239fc9
$t = 53$:	431d65bc	1797e228	627ef507	1f919cb0	21435047
$t = 54$:	2bdbb8cb	431d65bc	05e5f88a	627ef507	1f919cb0
$t = 55$:	6da72e7f	2bdbb8cb	10c7596f	05e5f88a	627ef507
$t = 56$:	a8495a9b	6da72e7f	caf6ee32	10c7596f	05e5f88a
$t = 57$:	e785655a	a8495a9b	db69cb9f	caf6ee32	10c7596f
$t = 58$:	5b086c42	e785655a	ea1256a6	db69cb9f	caf6ee32
$t = 59$:	a65818f7	5b086c42	b9e15956	ea1256a6	db69cb9f
$t = 60$:	7aab101b	a65818f7	96c21b10	b9e15956	ea1256a6
$t = 61$:	93614c9c	7aab101b	e996063d	96c21b10	b9e15956
$t = 62$:	f66d9bf4	93614c9c	deaac406	e996063d	96c21b10
$t = 63$:	d504902b	f66d9bf4	24d85327	deaac406	e996063d
$t = 64$:	60a9da62	d504902b	3d9b66fd	24d85327	deaac406
$t = 65$:	8b687819	60a9da62	f541240a	3d9b66fd	24d85327
$t = 66$:	083e90c3	8b687819	982a7698	f541240a	3d9b66fd
$t = 67$:	f6226bbf	083e90c3	62da1e06	982a7698	f541240a
$t = 68$:	76c0563b	f6226bbf	c20fa430	62da1e06	982a7698
$t = 69$:	989dd165	76c0563b	fd889aef	c20fa430	62da1e06
$t = 70$:	8b2c7573	989dd165	ddb0158e	fd889aef	c20fa430
$t = 71$:	ae1b8e7b	8b2c7573	66277459	ddb0158e	fd889aef
$t = 72$:	ca1840de	ae1b8e7b	e2cb1d5c	66277459	ddb0158e
$t = 73$:	16f3babb	ca1840de	eb86e39e	e2cb1d5c	66277459
$t = 74$:	d28d83ad	16f3babb	b2861037	eb86e39e	e2cb1d5c
$t = 75$:	6bc02dfe	d28d83ad	c5bceeae	b2861037	eb86e39e
$t = 76$:	d3a6e275	6bc02dfe	74a360eb	c5bceeae	b2861037
$t = 77$:	da955482	d3a6e275	9af00b7f	74a360eb	c5bceeae

$t = 78:$	58c0aac0	da955482	74e9b89d	9af00b7f	74a360eb
$t = 79:$	906fd62c	58c0aac0	b6a55520	74e9b89d	9af00b7f

That completes the processing of the second and final message block, $M^{(2)}$. The final hash value, $H^{(2)}$, is calculated to be

$$H_0^{(1)} = \texttt{f4286818} + \texttt{906fd62c} = \texttt{84983e44}$$
$$H_1^{(1)} = \texttt{c37b27ae} + \texttt{58c0aac0} = \texttt{1c3bd26e}$$
$$H_2^{(1)} = \texttt{0408f581} + \texttt{b6a55520} = \texttt{baae4aa1}$$
$$H_3^{(1)} = \texttt{84677148} + \texttt{74e9b89d} = \texttt{f95129e5}$$
$$H_4^{(1)} = \texttt{4a566572} + \texttt{9af00b7f} = \texttt{e54670f1}.$$

The resulting 160-bit message digest is

```
84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1.
```

A.3 SHA-1 Example (Long Message)

Let the message *M* be the binary-coded form of the ASCII string which consists of 1,000,000 repetitions of the character "**a**". The resulting SHA-1 message digest is

```
34aa973c d4c4daa4 f61eeb2b dbad2731 6534016f.
```

APPENDIX B: SHA-256 EXAMPLES

This appendix is for informational purposes only and is not required to meet the standard.

B.1 SHA-256 Example (One-Block Message)

Let the message, M, be the 24-bit ($\ell = 24$) ASCII string "**abc**", which is equivalent to the following binary string:

`01100001 01100010 01100011`.

The message is padded by appending a "`1`" bit, followed by 423 "`0`" bits, and ending with the hex value `00000000 00000018` (the two 32-bit word representation of the length, 24). Thus, the final padded message consists of one block ($N = 1$).

For SHA-256, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = `6a09e667`
$H_1^{(0)}$ = `bb67ae85`
$H_2^{(0)}$ = `3c6ef372`
$H_3^{(0)}$ = `a54ff53a`
$H_4^{(0)}$ = `510e527f`
$H_5^{(0)}$ = `9b05688c`
$H_6^{(0)}$ = `1f83d9ab`
$H_7^{(0)}$ = `5be0cd19`.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = `61626380`
W_1 = `00000000`
W_2 = `00000000`
W_3 = `00000000`
W_4 = `00000000`
W_5 = `00000000`
W_6 = `00000000`
W_7 = `00000000`
W_8 = `00000000`
W_9 = `00000000`
W_{10} = `00000000`
W_{11} = `00000000`
W_{12} = `00000000`
W_{13} = `00000000`
W_{14} = `00000000`
W_{15} = `00000018`.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass t of the "for t = 0 to 63" loop described in Sec. 6.2.2, step 4.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*
$t = 0$:	5d6aebcd	6a09e667	bb67ae85	3c6ef372	fa2a4622	510e527f	9b05688c	1f83d9ab
$t = 1$:	5a6ad9ad	5d6aebcd	6a09e667	bb67ae85	78ce7989	fa2a4622	510e527f	9b05688c
$t = 2$:	c8c347a7	5a6ad9ad	5d6aebcd	6a09e667	f92939eb	78ce7989	fa2a4622	510e527f
$t = 3$:	d550f666	c8c347a7	5a6ad9ad	5d6aebcd	24e00850	f92939eb	78ce7989	fa2a4622
$t = 4$:	04409a6a	d550f666	c8c347a7	5a6ad9ad	43ada245	24e00850	f92939eb	78ce7989
$t = 5$:	2b4209f5	04409a6a	d550f666	c8c347a7	714260ad	43ada245	24e00850	f92939eb
$t = 6$:	e5030380	2b4209f5	04409a6a	d550f666	9b27a401	714260ad	43ada245	24e00850
$t = 7$:	85a07b5f	e5030380	2b4209f5	04409a6a	0c657a79	9b27a401	714260ad	43ada245
$t = 8$:	8e04ecb9	85a07b5f	e5030380	2b4209f5	32ca2d8c	0c657a79	9b27a401	714260ad
$t = 9$:	8c87346b	8e04ecb9	85a07b5f	e5030380	1cc92596	32ca2d8c	0c657a79	9b27a401
$t = 10$:	4798a3f4	8c87346b	8e04ecb9	85a07b5f	436b23e8	1cc92596	32ca2d8c	0c657a79
$t = 11$:	f71fc5a9	4798a3f4	8c87346b	8e04ecb9	816fd6e9	436b23e8	1cc92596	32ca2d8c
$t = 12$:	87912990	f71fc5a9	4798a3f4	8c87346b	1e578218	816fd6e9	436b23e8	1cc92596
$t = 13$:	d932eb16	87912990	f71fc5a9	4798a3f4	745a48de	1e578218	816fd6e9	436b23e8
$t = 14$:	c0645fde	d932eb16	87912990	f71fc5a9	0b92f20c	745a48de	1e578218	816fd6e9
$t = 15$:	b0fa238e	c0645fde	d932eb16	87912990	07590dcd	0b92f20c	745a48de	1e578218
$t = 16$:	21da9a9b	b0fa238e	c0645fde	d932eb16	8034229c	07590dcd	0b92f20c	745a48de
$t = 17$:	c2fbd9d1	21da9a9b	b0fa238e	c0645fde	846ee454	8034229c	07590dcd	0b92f20c
$t = 18$:	fe777bbf	c2fbd9d1	21da9a9b	b0fa238e	cc899961	846ee454	8034229c	07590dcd
$t = 19$:	e1f20c33	fe777bbf	c2fbd9d1	21da9a9b	b0638179	cc899961	846ee454	8034229c
$t = 20$:	9dc68b63	e1f20c33	fe777bbf	c2fbd9d1	8ada8930	b0638179	cc899961	846ee454
$t = 21$:	c2606d6d	9dc68b63	e1f20c33	fe777bbf	e1257970	8ada8930	b0638179	cc899961
$t = 22$:	a7a3623f	c2606d6d	9dc68b63	e1f20c33	49f5114a	e1257970	8ada8930	b0638179
$t = 23$:	c5d53d8d	a7a3623f	c2606d6d	9dc68b63	aa47c347	49f5114a	e1257970	8ada8930
$t = 24$:	1c2c2838	c5d53d8d	a7a3623f	c2606d6d	2823ef91	aa47c347	49f5114a	e1257970
$t = 25$:	cde8037d	1c2c2838	c5d53d8d	a7a3623f	14383d8e	2823ef91	aa47c347	49f5114a
$t = 26$:	b62ec4bc	cde8037d	1c2c2838	c5d53d8d	c74c6516	14383d8e	2823ef91	aa47c347
$t = 27$:	77d37528	b62ec4bc	cde8037d	1c2c2838	edffbff8	c74c6516	14383d8e	2823ef91
$t = 28$:	363482c9	77d37528	b62ec4bc	cde8037d	6112a3b7	edffbff8	c74c6516	14383d8e
$t = 29$:	a0060b30	363482c9	77d37528	b62ec4bc	ade79437	6112a3b7	edffbff8	c74c6516
$t = 30$:	ea992a22	a0060b30	363482c9	77d37528	0109ab3a	ade79437	6112a3b7	edffbff8
$t = 31$:	73b33bf5	ea992a22	a0060b30	363482c9	ba591112	0109ab3a	ade79437	6112a3b7
$t = 32$:	98e12507	73b33bf5	ea992a22	a0060b30	9cd9f5f6	ba591112	0109ab3a	ade79437
$t = 33$:	fe604df5	98e12507	73b33bf5	ea992a22	59249dd3	9cd9f5f6	ba591112	0109ab3a
$t = 34$:	a9a7738c	fe604df5	98e12507	73b33bf5	085f3833	59249dd3	9cd9f5f6	ba591112
$t = 35$:	65a0cfe4	a9a7738c	fe604df5	98e12507	f4b002d6	085f3833	59249dd3	9cd9f5f6
$t = 36$:	41a65cb1	65a0cfe4	a9a7738c	fe604df5	0772a26b	f4b002d6	085f3833	59249dd3
$t = 37$:	34df1604	41a65cb1	65a0cfe4	a9a7738c	a507a53d	0772a26b	f4b002d6	085f3833
$t = 38$:	6dc57a8a	34df1604	41a65cb1	65a0cfe4	f0781bc8	a507a53d	0772a26b	f4b002d6
$t = 39$:	79ea687a	6dc57a8a	34df1604	41a65cb1	1efbc0a0	f0781bc8	a507a53d	0772a26b
$t = 40$:	d6670766	79ea687a	6dc57a8a	34df1604	26352d63	1efbc0a0	f0781bc8	a507a53d
$t = 41$:	df46652f	d6670766	79ea687a	6dc57a8a	838b2711	26352d63	1efbc0a0	f0781bc8
$t = 42$:	17aa0dfe	df46652f	d6670766	79ea687a	decd4715	838b2711	26352d63	1efbc0a0
$t = 43$:	9d4baf93	17aa0dfe	df46652f	d6670766	fda24c2e	decd4715	838b2711	26352d63
$t = 44$:	26628815	9d4baf93	17aa0dfe	df46652f	a80f11f0	fda24c2e	decd4715	838b2711
$t = 45$:	72ab4b91	26628815	9d4baf93	17aa0dfe	b7755da1	a80f11f0	fda24c2e	decd4715
$t = 46$:	a14c14b0	72ab4b91	26628815	9d4baf93	d57b94a9	b7755da1	a80f11f0	fda24c2e
$t = 47$:	4172328d	a14c14b0	72ab4b91	26628815	fecf0bc6	d57b94a9	b7755da1	a80f11f0
$t = 48$:	05757ceb	4172328d	a14c14b0	72ab4b91	bd714038	fecf0bc6	d57b94a9	b7755da1
$t = 49$:	f11bfaa8	05757ceb	4172328d	a14c14b0	6e5c390c	bd714038	fecf0bc6	d57b94a9
$t = 50$:	7a0508a1	f11bfaa8	05757ceb	4172328d	52f1ccf7	6e5c390c	bd714038	fecf0bc6
$t = 51$:	886e7a22	7a0508a1	f11bfaa8	05757ceb	49231c1e	52f1ccf7	6e5c390c	bd714038

$t = 52$:	101fd28f	886e7a22	7a0508a1	f11bfaa8	529e7d00	49231c1e	52f1ccf7	6e5c390c
$t = 53$:	f5702fdb	101fd28f	886e7a22	7a0508a1	9f4787c3	529e7d00	49231c1e	52f1ccf7
$t = 54$:	3ec45cdb	f5702fdb	101fd28f	886e7a22	e50e1b4f	9f4787c3	529e7d00	49231c1e
$t = 55$:	38cc9913	3ec45cdb	f5702fdb	101fd28f	54cb266b	e50e1b4f	9f4787c3	529e7d00
$t = 56$:	fcd1887b	38cc9913	3ec45cdb	f5702fdb	9b5e906c	54cb266b	e50e1b4f	9f4787c3
$t = 57$:	c062d46f	fcd1887b	38cc9913	3ec45cdb	7e44008e	9b5e906c	54cb266b	e50e1b4f
$t = 58$:	ffb70472	c062d46f	fcd1887b	38cc9913	6d83bfc6	7e44008e	9b5e906c	54cb266b
$t = 59$:	b6ae8fff	ffb70472	c062d46f	fcd1887b	b21bad3d	6d83bfc6	7e44008e	9b5e906c
$t = 60$:	b85e2ce9	b6ae8fff	ffb70472	c062d46f	961f4894	b21bad3d	6d83bfc6	7e44008e
$t = 61$:	04d24d6c	b85e2ce9	b6ae8fff	ffb70472	948d25b6	961f4894	b21bad3d	6d83bfc6
$t = 62$:	d39a2165	04d24d6c	b85e2ce9	b6ae8fff	fb121210	948d25b6	961f4894	b21bad3d
$t = 63$:	506e3058	d39a2165	04d24d6c	b85e2ce9	5ef50f24	fb121210	948d25b6	961f4894

That completes the processing of the first and only message block, $M^{(1)}$. The final hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{6a09e667} + \texttt{506e3058} = \texttt{ba7816bf}$$
$$H_1^{(1)} = \texttt{bb67ae85} + \texttt{d39a2165} = \texttt{8f01cfea}$$
$$H_2^{(1)} = \texttt{3c6ef372} + \texttt{04d24d6c} = \texttt{414140de}$$
$$H_3^{(1)} = \texttt{a54ff53a} + \texttt{b85e2ce9} = \texttt{5dae2223}$$
$$H_4^{(1)} = \texttt{510e527f} + \texttt{5ef50f24} = \texttt{b00361a3}$$
$$H_5^{(1)} = \texttt{9b05688c} + \texttt{fb121210} = \texttt{96177a9c}$$
$$H_6^{(1)} = \texttt{1f83d9ab} + \texttt{948d25b6} = \texttt{b410ff61}$$
$$H_7^{(1)} = \texttt{5be0cd19} + \texttt{961f4894} = \texttt{f20015ad.}$$

The resulting 256-bit message digest is

```
ba7816bf 8f01cfea 414140de 5dae2223 b00361a3 96177a9c b410ff61 f20015ad.
```

B.2 SHA-256 Example (Multi-Block Message)

Let the message, *M*, be the 448-bit ($\ell = 448$) ASCII string

"**abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq**".

The message is padded by appending a "`1`" bit, followed by 511 "`0`" bits, and ending with the hex value `00000000 000001c0` (the two 32-bit word representation of the length, 448). Thus, the final padded message consists of two blocks ($N = 2$).

For SHA-256, the initial hash value, $H^{(0)}$, is

$$H_0^{(0)} = \texttt{6a09e667}$$
$$H_1^{(0)} = \texttt{bb67ae85}$$
$$H_2^{(0)} = \texttt{3c6ef372}$$

$H_3^{(0)}$ = a54ff53a
$H_4^{(0)}$ = 510e527f
$H_5^{(0)}$ = 9b05688c
$H_6^{(0)}$ = 1f83d9ab
$H_7^{(0)}$ = 5be0cd19.

The words of the first padded message block, $M^{(1)}$, are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 61626364
W_1 = 62636465
W_2 = 63646566
W_3 = 64656667
W_4 = 65666768
W_5 = 66676869
W_6 = 6768696a
W_7 = 68696a6b
W_8 = 696a6b6c
W_9 = 6a6b6c6d
W_{10} = 6b6c6d6e
W_{11} = 6c6d6e6f
W_{12} = 6d6e6f70
W_{13} = 6e6f7071
W_{14} = 80000000
W_{15} = 00000000.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 63" loop described in Sec. 6.2.2, step 4.

	a	***b***	***c***	***d***	***e***	***f***	***g***	***h***
t = 0:	5d6aebb1	6a09e667	bb67ae85	3c6ef372	fa2a4606	510e527f	9b05688c	1f83d9ab
t = 1:	2f2d5fcf	5d6aebb1	6a09e667	bb67ae85	4eb1cfce	fa2a4606	510e527f	9b05688c
t = 2:	97651825	2f2d5fcf	5d6aebb1	6a09e667	62d5c49e	4eb1cfce	fa2a4606	510e527f
t = 3:	4a8d64d5	97651825	2f2d5fcf	5d6aebb1	6494841b	62d5c49e	4eb1cfce	fa2a4606
t = 4:	f921c212	4a8d64d5	97651825	2f2d5fcf	05c4f88a	6494841b	62d5c49e	4eb1cfce
t = 5:	55c8ef48	f921c212	4a8d64d5	97651825	7ff91c94	05c4f88a	6494841b	62d5c49e
t = 6:	485835b7	55c8ef48	f921c212	4a8d64d5	39a5b2ca	7ff91c94	05c4f88a	6494841b
t = 7:	d237e6db	485835b7	55c8ef48	f921c212	a401d211	39a5b2ca	7ff91c94	05c4f88a
t = 8:	359f2bce	d237e6db	485835b7	55c8ef48	c09ffec4	a401d211	39a5b2ca	7ff91c94
t = 9:	3a474b2b	359f2bce	d237e6db	485835b7	9037b3b8	c09ffec4	a401d211	39a5b2ca
t = 10:	b8e2b4cb	3a474b2b	359f2bce	d237e6db	443ed29e	9037b3b8	c09ffec4	a401d211
t = 11:	1762215c	b8e2b4cb	3a474b2b	359f2bce	ee1c97a8	443ed29e	9037b3b8	c09ffec4
t = 12:	101a4861	1762215c	b8e2b4cb	3a474b2b	839a0fc9	ee1c97a8	443ed29e	9037b3b8
t = 13:	d68e6457	101a4861	1762215c	b8e2b4cb	9243f8af	839a0fc9	ee1c97a8	443ed29e
t = 14:	dd16cbb3	d68e6457	101a4861	1762215c	9162aded	9243f8af	839a0fc9	ee1c97a8
t = 15:	c3486194	dd16cbb3	d68e6457	101a4861	1496a54f	9162aded	9243f8af	839a0fc9
t = 16:	b9dcacb1	c3486194	dd16cbb3	d68e6457	d4f64250	1496a54f	9162aded	9243f8af
t = 17:	046a193e	b9dcacb1	c3486194	dd16cbb3	885370b6	d4f64250	1496a54f	9162aded
t = 18:	f402f058	046a193e	b9dcacb1	c3486194	6f433549	885370b6	d4f64250	1496a54f
t = 19:	2139187b	f402f058	046a193e	b9dcacb1	7c304206	6f433549	885370b6	d4f64250
t = 20:	d70ac17d	2139187b	f402f058	046a193e	7cc6b262	7c304206	6f433549	885370b6
t = 21:	1b2b66b8	d70ac17d	2139187b	f402f058	d560b028	7cc6b262	7c304206	6f433549
t = 22:	ae2e2d4f	1b2b66b8	d70ac17d	2139187b	f074fc95	d560b028	7cc6b262	7c304206
t = 23:	59fce6b9	ae2e2d4f	1b2b66b8	d70ac17d	a2c7d51d	f074fc95	d560b028	7cc6b262
t = 24:	4a885065	59fce6b9	ae2e2d4f	1b2b66b8	763597fb	a2c7d51d	f074fc95	d560b028

$t = 25$:	573221da	4a885065	59fce6b9	ae2e2d4f	36e74eb4	763597fb	a2c7d51d	f074fc95
$t = 26$:	128661da	573221da	4a885065	59fce6b9	1162d575	36e74eb4	763597fb	a2c7d51d
$t = 27$:	73f858af	128661da	573221da	4a885065	e77c797f	1162d575	36e74eb4	763597fb
$t = 28$:	74bcf468	73f858af	128661da	573221da	72abaecd	e77c797f	1162d575	36e74eb4
$t = 29$:	df7151a0	74bcf468	73f858af	128661da	7629c961	72abaecd	e77c797f	1162d575
$t = 30$:	eb43f3ed	df7151a0	74bcf468	73f858af	0635d880	7629c961	72abaecd	e77c797f
$t = 31$:	5581ab07	eb43f3ed	df7151a0	74bcf468	df980085	0635d880	7629c961	72abaecd
$t = 32$:	9fc905c8	5581ab07	eb43f3ed	df7151a0	a94d2af1	df980085	0635d880	7629c961
$t = 33$:	9ce5a62f	9fc905c8	5581ab07	eb43f3ed	6ef3b6bd	a94d2af1	df980085	0635d880
$t = 34$:	1df8e885	9ce5a62f	9fc905c8	5581ab07	2a9e048e	6ef3b6bd	a94d2af1	df980085
$t = 35$:	0786dce8	1df8e885	9ce5a62f	9fc905c8	de2a21d1	2a9e048e	6ef3b6bd	a94d2af1
$t = 36$:	2c55d3a6	0786dce8	1df8e885	9ce5a62f	b067c1af	de2a21d1	2a9e048e	6ef3b6bd
$t = 37$:	a985b4be	2c55d3a6	0786dce8	1df8e885	f72bf353	b067c1af	de2a21d1	2a9e048e
$t = 38$:	91ac9d5d	a985b4be	2c55d3a6	0786dce8	68d8d590	f72bf353	b067c1af	de2a21d1
$t = 39$:	7e4d30b8	91ac9d5d	a985b4be	2c55d3a6	9f5b9b6d	68d8d590	f72bf353	b067c1af
$t = 40$:	7e056794	7e4d30b8	91ac9d5d	a985b4be	423b26c0	9f5b9b6d	68d8d590	f72bf353
$t = 41$:	508a16ab	7e056794	7e4d30b8	91ac9d5d	45459d97	423b26c0	9f5b9b6d	68d8d590
$t = 42$:	b62c7013	508a16ab	7e056794	7e4d30b8	80a92a00	45459d97	423b26c0	9f5b9b6d
$t = 43$:	167361de	b62c7013	508a16ab	7e056794	41dd3844	80a92a00	45459d97	423b26c0
$t = 44$:	de71e2f2	167361de	b62c7013	508a16ab	ff61c636	41dd3844	80a92a00	45459d97
$t = 45$:	18f0d19d	de71e2f2	167361de	b62c7013	6b88472c	ff61c636	41dd3844	80a92a00
$t = 46$:	165be9cd	18f0d19d	de71e2f2	167361de	a483f080	6b88472c	ff61c636	41dd3844
$t = 47$:	13d82741	165be9cd	18f0d19d	de71e2f2	a7802a4d	a483f080	6b88472c	ff61c636
$t = 48$:	017b9d99	13d82741	165be9cd	18f0d19d	aeb10b60	a7802a4d	a483f080	6b88472c
$t = 49$:	543c99a1	017b9d99	13d82741	165be9cd	16f134b6	aeb10b60	a7802a4d	a483f080
$t = 50$:	758ca97a	543c99a1	017b9d99	13d82741	100cf2ea	16f134b6	aeb10b60	a7802a4d
$t = 51$:	81c1cde0	758ca97a	543c99a1	017b9d99	5c47eb7b	100cf2ea	16f134b6	aeb10b60
$t = 52$:	b8d55619	81c1cde0	758ca97a	543c99a1	1c806a61	5c47eb7b	100cf2ea	16f134b6
$t = 53$:	1d6de87a	b8d55619	81c1cde0	758ca97a	3443bed4	1c806a61	5c47eb7b	100cf2ea
$t = 54$:	f907b313	1d6de87a	b8d55619	81c1cde0	61a41711	3443bed4	1c806a61	5c47eb7b
$t = 55$:	9e57c4a0	f907b313	1d6de87a	b8d55619	eec13548	61a41711	3443bed4	1c806a61
$t = 56$:	71629856	9e57c4a0	f907b313	1d6de87a	2f6c8c4e	eec13548	61a41711	3443bed4
$t = 57$:	7c015a2c	71629856	9e57c4a0	f907b313	cb9d3dd0	2f6c8c4e	eec13548	61a41711
$t = 58$:	921fccb6	7c015a2c	71629856	9e57c4a0	43d8a034	cb9d3dd0	2f6c8c4e	eec13548
$t = 59$:	e18f259a	921fccb6	7c015a2c	71629856	51e15869	43d8a034	cb9d3dd0	2f6c8c4e
$t = 60$:	bcfce922	e18f259a	921fccb6	7c015a2c	962d8621	51e15869	43d8a034	cb9d3dd0
$t = 61$:	f6f443f8	bcfce922	e18f259a	921fccb6	acc75916	962d8621	51e15869	43d8a034
$t = 62$:	86126910	f6f443f8	bcfce922	e18f259a	2fc08f85	acc75916	962d8621	51e15869
$t = 63$:	1bdc6f6f	86126910	f6f443f8	bcfce922	25d2430a	2fc08f85	acc75916	962d8621

That completes the processing of the first message block, $M^{(1)}$. The first intermediate hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{6a09e667} + \texttt{1bdc6f6f} = \texttt{85e655d6}$$
$$H_1^{(1)} = \texttt{bb67ae85} + \texttt{86126910} = \texttt{417a1795}$$
$$H_2^{(1)} = \texttt{3c6ef372} + \texttt{f6f443f8} = \texttt{3363376a}$$
$$H_3^{(1)} = \texttt{a54ff53a} + \texttt{bcfce922} = \texttt{624cde5c}$$
$$H_4^{(1)} = \texttt{510e527f} + \texttt{25d2430a} = \texttt{76e09589}$$
$$H_5^{(1)} = \texttt{9b05688c} + \texttt{2fc08f85} = \texttt{cac5f811}$$
$$H_6^{(1)} = \texttt{1f83d9ab} + \texttt{acc75916} = \texttt{cc4b32c1}$$

$H_7^{(1)}$ = 5be0cd19 + 962d8621 = f20e533a.

The words of the *second* padded message block, $M^{(2)}$, are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 00000000
W_1 = 00000000
W_2 = 00000000
W_3 = 00000000
W_4 = 00000000
W_5 = 00000000
W_6 = 00000000
W_7 = 00000000
W_8 = 00000000
W_9 = 00000000
W_{10} = 00000000
W_{11} = 00000000
W_{12} = 00000000
W_{13} = 00000000
W_{14} = 00000000
W_{15} = 000001c0.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 63" loop described in Sec. 6.2.2, step 4.

	a	***b***	***c***	***d***	***e***	***f***	***g***	***h***
t = 0:	7c20c838	85e655d6	417a1795	3363376a	4670ae6e	76e09589	cac5f811	cc4b32c1
t = 1:	7c3c0f86	7c20c838	85e655d6	417a1795	8c51be64	4670ae6e	76e09589	cac5f811
t = 2:	fd1eebdc	7c3c0f86	7c20c838	85e655d6	af71b9ea	8c51be64	4670ae6e	76e09589
t = 3:	f268faa9	fd1eebdc	7c3c0f86	7c20c838	e20362ef	af71b9ea	8c51be64	4670ae6e
t = 4:	185a5d79	f268faa9	fd1eebdc	7c3c0f86	8dff3001	e20362ef	af71b9ea	8c51be64
t = 5:	3eeb6c06	185a5d79	f268faa9	fd1eebdc	fe20cda6	8dff3001	e20362ef	af71b9ea
t = 6:	89bba3f1	3eeb6c06	185a5d79	f268faa9	0a34df03	fe20cda6	8dff3001	e20362ef
t = 7:	bf9a93a0	89bba3f1	3eeb6c06	185a5d79	059abdd1	0a34df03	fe20cda6	8dff3001
t = 8:	2c096744	bf9a93a0	89bba3f1	3eeb6c06	abfa465b	059abdd1	0a34df03	fe20cda6
t = 9:	2d964e86	2c096744	bf9a93a0	89bba3f1	aa27ed82	abfa465b	059abdd1	0a34df03
t = 10:	5b35025b	2d964e86	2c096744	bf9a93a0	10e77723	aa27ed82	abfa465b	059abdd1
t = 11:	5eb4ec40	5b35025b	2d964e86	2c096744	e11b4548	10e77723	aa27ed82	abfa465b
t = 12:	35ee996d	5eb4ec40	5b35025b	2d964e86	5c24e2a2	e11b4548	10e77723	aa27ed82
t = 13:	d74080fa	35ee996d	5eb4ec40	5b35025b	68aa893f	5c24e2a2	e11b4548	10e77723
t = 14:	0cea5cbc	d74080fa	35ee996d	5eb4ec40	60356548	68aa893f	5c24e2a2	e11b4548
t = 15:	16a8cc79	0cea5cbc	d74080fa	35ee996d	0fcb1f6f	60356548	68aa893f	5c24e2a2
t = 16:	f16f634e	16a8cc79	0cea5cbc	d74080fa	8b21cdc1	0fcb1f6f	60356548	68aa893f
t = 17:	23dcb6c2	f16f634e	16a8cc79	0cea5cbc	ca9182d3	8b21cdc1	0fcb1f6f	60356548
t = 18:	dcff40fd	23dcb6c2	f16f634e	16a8cc79	69bf7b95	ca9182d3	8b21cdc1	0fcb1f6f
t = 19:	76f1a2bc	dcff40fd	23dcb6c2	f16f634e	0dc84bb1	69bf7b95	ca9182d3	8b21cdc1
t = 20:	20aad899	76f1a2bc	dcff40fd	23dcb6c2	cc4769f2	0dc84bb1	69bf7b95	ca9182d3
t = 21:	d44dc81a	20aad899	76f1a2bc	dcff40fd	5bace62d	cc4769f2	0dc84bb1	69bf7b95
t = 22:	f13ae55b	d44dc81a	20aad899	76f1a2bc	966aa287	5bace62d	cc4769f2	0dc84bb1
t = 23:	a4195b91	f13ae55b	d44dc81a	20aad899	eddbd6ed	966aa287	5bace62d	cc4769f2
t = 24:	4984fa79	a4195b91	f13ae55b	d44dc81a	a530d939	eddbd6ed	966aa287	5bace62d
t = 25:	aa6cb982	4984fa79	a4195b91	f13ae55b	0b5eeea4	a530d939	eddbd6ed	966aa287
t = 26:	9450fbbc	aa6cb982	4984fa79	a4195b91	09166dda	0b5eeea4	a530d939	eddbd6ed
t = 27:	0d936bab	9450fbbc	aa6cb982	4984fa79	6e495d4b	09166dda	0b5eeea4	a530d939
t = 28:	d958b529	0d936bab	9450fbbc	aa6cb982	c2fa99b1	6e495d4b	09166dda	0b5eeea4
t = 29:	1cfa5eb0	d958b529	0d936bab	9450fbbc	6c49db9f	c2fa99b1	6e495d4b	09166dda
t = 30:	02ef3a5f	1cfa5eb0	d958b529	0d936bab	5da10665	6c49db9f	c2fa99b1	6e495d4b
t = 31:	b0eab1c5	02ef3a5f	1cfa5eb0	d958b529	f6d93952	5da10665	6c49db9f	c2fa99b1

$t = 32$:	0bfba73c	b0eab1c5	02ef3a5f	1cfa5eb0	8b99e3a9	f6d93952	5da10665	6c49db9f
$t = 33$:	4bd1df96	0bfba73c	b0eab1c5	02ef3a5f	905e44ac	8b99e3a9	f6d93952	5da10665
$t = 34$:	9907f1b6	4bd1df96	0bfba73c	b0eab1c5	66c3043d	905e44ac	8b99e3a9	f6d93952
$t = 35$:	ecde4e0d	9907f1b6	4bd1df96	0bfba73c	5dc119e6	66c3043d	905e44ac	8b99e3a9
$t = 36$:	2f11c939	ecde4e0d	9907f1b6	4bd1df96	fed4ce1d	5dc119e6	66c3043d	905e44ac
$t = 37$:	d949682b	2f11c939	ecde4e0d	9907f1b6	32d99008	fed4ce1d	5dc119e6	66c3043d
$t = 38$:	adca7a96	d949682b	2f11c939	ecde4e0d	c6cce4ff	32d99008	fed4ce1d	5dc119e6
$t = 39$:	221b8a5a	adca7a96	d949682b	2f11c939	0b82c5eb	c6cce4ff	32d99008	fed4ce1d
$t = 40$:	12d97845	221b8a5a	adca7a96	d949682b	e4213ca2	0b82c5eb	c6cce4ff	32d99008
$t = 41$:	2c794876	12d97845	221b8a5a	adca7a96	ff6759ba	e4213ca2	0b82c5eb	c6cce4ff
$t = 42$:	8300fca2	2c794876	12d97845	221b8a5a	e0e3457c	ff6759ba	e4213ca2	0b82c5eb
$t = 43$:	f2ad6322	8300fca2	2c794876	12d97845	cc48c7f3	e0e3457c	ff6759ba	e4213ca2
$t = 44$:	0f154e11	f2ad6322	8300fca2	2c794876	6f9517cb	cc48c7f3	e0e3457c	ff6759ba
$t = 45$:	104a7db4	0f154e11	f2ad6322	8300fca2	5348e8f6	6f9517cb	cc48c7f3	e0e3457c
$t = 46$:	0b3303a7	104a7db4	0f154e11	f2ad6322	bbe1c39a	5348e8f6	6f9517cb	cc48c7f3
$t = 47$:	d7354d5b	0b3303a7	104a7db4	0f154e11	aad55b6b	bbe1c39a	5348e8f6	6f9517cb
$t = 48$:	b736d7a6	d7354d5b	0b3303a7	104a7db4	68f25260	aad55b6b	bbe1c39a	5348e8f6
$t = 49$:	2748e5ec	b736d7a6	d7354d5b	0b3303a7	d4b58576	68f25260	aad55b6b	bbe1c39a
$t = 50$:	d8aabcf9	2748e5ec	b736d7a6	d7354d5b	27844711	d4b58576	68f25260	aad55b6b
$t = 51$:	1a6bcf6a	d8aabcf9	2748e5ec	b736d7a6	ff5e99d0	27844711	d4b58576	68f25260
$t = 52$:	4eca6fa0	1a6bcf6a	d8aabcf9	2748e5ec	989ed071	ff5e99d0	27844711	d4b58576
$t = 53$:	ec02560a	4eca6fa0	1a6bcf6a	d8aabcf9	7151df8e	989ed071	ff5e99d0	27844711
$t = 54$:	d9f0c115	ec02560a	4eca6fa0	1a6bcf6a	624150c4	7151df8e	989ed071	ff5e99d0
$t = 55$:	92952710	d9f0c115	ec02560a	4eca6fa0	226806d6	624150c4	7151df8e	989ed071
$t = 56$:	20d4d0e4	92952710	d9f0c115	ec02560a	4e515a4d	226806d6	624150c4	7151df8e
$t = 57$:	4348eb1f	20d4d0e4	92952710	d9f0c115	c21eddf9	4e515a4d	226806d6	624150c4
$t = 58$:	286fe5f0	4348eb1f	20d4d0e4	92952710	54076664	c21eddf9	4e515a4d	226806d6
$t = 59$:	1c4cddd9	286fe5f0	4348eb1f	20d4d0e4	f487a853	54076664	c21eddf9	4e515a4d
$t = 60$:	a9f181dd	1c4cddd9	286fe5f0	4348eb1f	27ccb387	f487a853	54076664	c21eddf9
$t = 61$:	b25cef29	a9f181dd	1c4cddd9	286fe5f0	2aa1bb13	27ccb387	f487a853	54076664
$t = 62$:	908c2123	b25cef29	a9f181dd	1c4cddd9	9a392956	2aa1bb13	27ccb387	f487a853
$t = 63$:	9ea7148b	908c2123	b25cef29	a9f181dd	2c5c4ed0	9a392956	2aa1bb13	27ccb387

That completes the processing of the second and final message block, $M^{(2)}$. The final hash value, $H^{(2)}$, is calculated to be

$$H_0^{(2)} = \texttt{85e655d6} + \texttt{9ea7148b} = \texttt{248d6a61}$$
$$H_1^{(2)} = \texttt{417a1795} + \texttt{908c2123} = \texttt{d20638b8}$$
$$H_2^{(2)} = \texttt{3363376a} + \texttt{b25cef29} = \texttt{e5c02693}$$
$$H_3^{(2)} = \texttt{624cde5c} + \texttt{a9f181dd} = \texttt{0c3e6039}$$
$$H_4^{(2)} = \texttt{76e09589} + \texttt{2c5c4ed0} = \texttt{a33ce459}$$
$$H_5^{(2)} = \texttt{cac5f811} + \texttt{9a392956} = \texttt{64ff2167}$$
$$H_6^{(2)} = \texttt{cc4b32c1} + \texttt{2aa1bb13} = \texttt{f6ecedd4}$$
$$H_7^{(2)} = \texttt{f20e533a} + \texttt{27ccb387} = \texttt{19db06c1}.$$

The resulting 256-bit message digest is

```
248d6a61 d20638b8 e5c02693 0c3e6039 a33ce459 64ff2167 f6ecedd4 19db06c1.
```

B.3 SHA-256 Example (Long Message)

Let the message *M* be the binary-coded form of the ASCII string which consists of 1,000,000 repetitions of the character "**a**". The resulting SHA-256 message digest is

```
cdc76e5c 9914fb92 81a1c7e2 84d73e67 f1809a48 a497200e 046d39cc c7112cd0.
```

APPENDIX C: SHA-512 EXAMPLES

This appendix is for informational purposes only and is not required to meet the standard.

C.1 SHA-512 Example (One-Block Message)

Let the message, *M*, be the 24-bit ($\ell = 24$) ASCII string "**abc**", which is equivalent to the following binary string:

01100001 01100010 01100011.

The message is padded by appending a "1" bit, followed by 871 "0" bits, and ending with the hex value

0000000000000000 0000000000000018

(the two 64-bit word representation of the length, 24). Thus, the final padded message consists of one block ($N = 1$).

For SHA-512, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = 6a09e667f3bcc908
$H_1^{(0)}$ = bb67ae8584caa73b
$H_2^{(0)}$ = 3c6ef372fe94f82b
$H_3^{(0)}$ = a54ff53a5f1d36f1
$H_4^{(0)}$ = 510e527fade682d1
$H_5^{(0)}$ = 9b05688c2b3e6c1f
$H_6^{(0)}$ = 1f83d9abfb41bd6b
$H_7^{(0)}$ = 5be0cd19137e2179.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 6162638000000000
W_1 = 0000000000000000
W_2 = 0000000000000000
W_3 = 0000000000000000
W_4 = 0000000000000000
W_5 = 0000000000000000
W_6 = 0000000000000000
W_7 = 0000000000000000
W_8 = 0000000000000000
W_9 = 0000000000000000
W_{10} = 0000000000000000
W_{11} = 0000000000000000
W_{12} = 0000000000000000
W_{13} = 0000000000000000
W_{14} = 0000000000000000
W_{15} = 0000000000000018.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 79" loop described in Sec. 6.3.2, step 4.

	a / ***e***	***b*** / ***f***	***c*** / ***g***	***d*** / ***h***
t = 0 :	f6afceb8bcfcddf5	6a09e667f3bcc908	bb67ae8584caa73b	3c6ef372fe94f82b
	58cb02347ab51f91	510e527fade682d1	9b05688c2b3e6c1f	1f83d9abfb41bd6b
t = 1 :	1320f8c9fb872cc0	f6afceb8bcfcddf5	6a09e667f3bcc908	bb67ae8584caa73b
	c3d4ebfd48650ffa	58cb02347ab51f91	510e527fade682d1	9b05688c2b3e6c1f
t = 2 :	ebcffc07203d91f3	1320f8c9fb872cc0	f6afceb8bcfcddf5	6a09e667f3bcc908
	dfa9b239f2697812	c3d4ebfd48650ffa	58cb02347ab51f91	510e527fade682d1
t = 3 :	5a83cb3e80050e82	ebcffc07203d91f3	1320f8c9fb872cc0	f6afceb8bcfcddf5
	0b47b4bb1928990e	dfa9b239f2697812	c3d4ebfd48650ffa	58cb02347ab51f91
t = 4 :	b680953951604860	5a83cb3e80050e82	ebcffc07203d91f3	1320f8c9fb872cc0
	745aca4a342ed2e2	0b47b4bb1928990e	dfa9b239f2697812	c3d4ebfd48650ffa
t = 5 :	af573b02403e89cd	b680953951604860	5a83cb3e80050e82	ebcffc07203d91f3
	96f60209b6dc35ba	745aca4a342ed2e2	0b47b4bb1928990e	dfa9b239f2697812
t = 6 :	c4875b0c7abc076b	af573b02403e89cd	b680953951604860	5a83cb3e80050e82
	5a6c781f54dcc00c	96f60209b6dc35ba	745aca4a342ed2e2	0b47b4bb1928990e
t = 7 :	8093d195e0054fa3	c4875b0c7abc076b	af573b02403e89cd	b680953951604860
	86f67263a0f0ec0a	5a6c781f54dcc00c	96f60209b6dc35ba	745aca4a342ed2e2
t = 8 :	f1eca5544cb89225	8093d195e0054fa3	c4875b0c7abc076b	af573b02403e89cd
	d0403c398fc40002	86f67263a0f0ec0a	5a6c781f54dcc00c	96f60209b6dc35ba
t = 9 :	81782d4a5db48f03	f1eca5544cb89225	8093d195e0054fa3	c4875b0c7abc076b
	00091f460be46c52	d0403c398fc40002	86f67263a0f0ec0a	5a6c781f54dcc00c
t = 10 :	69854c4aa0f25b59	81782d4a5db48f03	f1eca5544cb89225	8093d195e0054fa3
	d375471bde1ba3f4	00091f460be46c52	d0403c398fc40002	86f67263a0f0ec0a
t = 11 :	db0a9963f80c2eaa	69854c4aa0f25b59	81782d4a5db48f03	f1eca5544cb89225
	475975b91a7a462c	d375471bde1ba3f4	00091f460be46c52	d0403c398fc40002
t = 12 :	5e41214388186c14	db0a9963f80c2eaa	69854c4aa0f25b59	81782d4a5db48f03
	cdf3bff2883fc9d9	475975b91a7a462c	d375471bde1ba3f4	00091f460be46c52
t = 13 :	44249631255d2ca0	5e41214388186c14	db0a9963f80c2eaa	69854c4aa0f25b59
	860acf9effba6f61	cdf3bff2883fc9d9	475975b91a7a462c	d375471bde1ba3f4
t = 14 :	fa967eed85a08028	44249631255d2ca0	5e41214388186c14	db0a9963f80c2eaa
	874bfe5f6aae9f2f	860acf9effba6f61	cdf3bff2883fc9d9	475975b91a7a462c
t = 15 :	0ae07c86b1181c75	fa967eed85a08028	44249631255d2ca0	5e41214388186c14
	a77b7c035dd4c161	874bfe5f6aae9f2f	860acf9effba6f61	cdf3bff2883fc9d9
t = 16 :	caf81a425d800537	0ae07c86b1181c75	fa967eed85a08028	44249631255d2ca0
	2deecc6b39d64d78	a77b7c035dd4c161	874bfe5f6aae9f2f	860acf9effba6f61
t = 17 :	4725be249ad19e6b	caf81a425d800537	0ae07c86b1181c75	fa967eed85a08028
	f47e8353f8047455	2deecc6b39d64d78	a77b7c035dd4c161	874bfe5f6aae9f2f
t = 18 :	3c4b4104168e3edb	4725be249ad19e6b	caf81a425d800537	0ae07c86b1181c75
	29695fd88d81dbd0	f47e8353f8047455	2deecc6b39d64d78	a77b7c035dd4c161
t = 19 :	9a3fb4d38ab6cf06	3c4b4104168e3edb	4725be249ad19e6b	caf81a425d800537
	f14998dd5f70767e	29695fd88d81dbd0	f47e8353f8047455	2deecc6b39d64d78

$t = 20$:	8dc5ae65569d3855	9a3fb4d38ab6cf06	3c4b4104168e3edb	4725be249ad19e6b
	4bb9e66d1145bfdc	f14998dd5f70767e	29695fd88d81dbd0	f47e8353f8047455
$t = 21$:	da34d6673d452dcf	8dc5ae65569d3855	9a3fb4d38ab6cf06	3c4b4104168e3edb
	8e30ff09ad488753	4bb9e66d1145bfdc	f14998dd5f70767e	29695fd88d81dbd0
$t = 22$:	3e2644567b709a78	da34d6673d452dcf	8dc5ae65569d3855	9a3fb4d38ab6cf06
	0ac2b11da8f571c6	8e30ff09ad488753	4bb9e66d1145bfdc	f14998dd5f70767e
$t = 23$:	4f6877b58fe55484	3e2644567b709a78	da34d6673d452dcf	8dc5ae65569d3855
	c66005f87db55233	0ac2b11da8f571c6	8e30ff09ad488753	4bb9e66d1145bfdc
$t = 24$:	9aff71163fa3a940	4f6877b58fe55484	3e2644567b709a78	da34d6673d452dcf
	d3ecf13769180e6f	c66005f87db55233	0ac2b11da8f571c6	8e30ff09ad488753
$t = 25$:	0bc5f791f8e6816b	9aff71163fa3a940	4f6877b58fe55484	3e2644567b709a78
	6ddf1fd7edcce336	d3ecf13769180e6f	c66005f87db55233	0ac2b11da8f571c6
$t = 26$:	884c3bc27bc4f941	0bc5f791f8e6816b	9aff71163fa3a940	4f6877b58fe55484
	e6e48c9a8e948365	6ddf1fd7edcce336	d3ecf13769180e6f	c66005f87db55233
$t = 27$:	eab4a9e5771b8d09	884c3bc27bc4f941	0bc5f791f8e6816b	9aff71163fa3a940
	09068a4e255a0dac	e6e48c9a8e948365	6ddf1fd7edcce336	d3ecf13769180e6f
$t = 28$:	e62349090f47d30a	eab4a9e5771b8d09	884c3bc27bc4f941	0bc5f791f8e6816b
	0fcdf99710f21584	09068a4e255a0dac	e6e48c9a8e948365	6ddf1fd7edcce336
$t = 29$:	74bf40f869094c63	e62349090f47d30a	eab4a9e5771b8d09	884c3bc27bc4f941
	f0aec2fe1437f085	0fcdf99710f21584	09068a4e255a0dac	e6e48c9a8e948365
$t = 30$:	4c4fbbb75f1873a6	74bf40f869094c63	e62349090f47d30a	eab4a9e5771b8d09
	73e025d91b9efea3	f0aec2fe1437f085	0fcdf99710f21584	09068a4e255a0dac
$t = 31$:	ff4d3f1f0d46a736	4c4fbbb75f1873a6	74bf40f869094c63	e62349090f47d30a
	3cd388e119e8162e	73e025d91b9efea3	f0aec2fe1437f085	0fcdf99710f21584
$t = 32$:	a0509015ca08c8d4	ff4d3f1f0d46a736	4c4fbbb75f1873a6	74bf40f869094c63
	e1034573654a106f	3cd388e119e8162e	73e025d91b9efea3	f0aec2fe1437f085
$t = 33$:	60d4e6995ed91fe6	a0509015ca08c8d4	ff4d3f1f0d46a736	4c4fbbb75f1873a6
	efabbd8bf47c041a	e1034573654a106f	3cd388e119e8162e	73e025d91b9efea3
$t = 34$:	2c59ec7743632621	60d4e6995ed91fe6	a0509015ca08c8d4	ff4d3f1f0d46a736
	0fbae670fa780fd3	efabbd8bf47c041a	e1034573654a106f	3cd388e119e8162e
$t = 35$:	1a081afc59fdbc2c	2c59ec7743632621	60d4e6995ed91fe6	a0509015ca08c8d4
	f098082f502b44cd	0fbae670fa780fd3	efabbd8bf47c041a	e1034573654a106f
$t = 36$:	88df85b0bbe77514	1a081afc59fdbc2c	2c59ec7743632621	60d4e6995ed91fe6
	8fbfd0162bbf4675	f098082f502b44cd	0fbae670fa780fd3	efabbd8bf47c041a
$t = 37$:	002bb8e4cd989567	88df85b0bbe77514	1a081afc59fdbc2c	2c59ec7743632621
	66adcfa249ac7bbd	8fbfd0162bbf4675	f098082f502b44cd	0fbae670fa780fd3
$t = 38$:	b3bb8542b3376de5	002bb8e4cd989567	88df85b0bbe77514	1a081afc59fdbc2c
	b49596c20feba7de	66adcfa249ac7bbd	8fbfd0162bbf4675	f098082f502b44cd
$t = 39$:	8e01e125b855d225	b3bb8542b3376de5	002bb8e4cd989567	88df85b0bbe77514
	0c710a47ba6a567b	b49596c20feba7de	66adcfa249ac7bbd	8fbfd0162bbf4675
$t = 40$:	b01521dd6a6be12c	8e01e125b855d225	b3bb8542b3376de5	002bb8e4cd989567
	169008b3a4bb170b	0c710a47ba6a567b	b49596c20feba7de	66adcfa249ac7bbd
$t = 41$:	e96f89dd48cbd851	b01521dd6a6be12c	8e01e125b855d225	b3bb8542b3376de5
	f0996439e7b50cb1	169008b3a4bb170b	0c710a47ba6a567b	b49596c20feba7de
$t = 42$:	bc05ba8de5d3c480	e96f89dd48cbd851	b01521dd6a6be12c	8e01e125b855d225
	639cb938e14dc190	f0996439e7b50cb1	169008b3a4bb170b	0c710a47ba6a567b
$t = 43$:	35d7e7f41defcbd5	bc05ba8de5d3c480	e96f89dd48cbd851	b01521dd6a6be12c

	cc5100997f5710f2	639cb938e14dc190	f0996439e7b50cb1	169008b3a4bb170b
$t = 44$:	c47c9d5c7ea8a234	35d7e7f41defcbd5	bc05ba8de5d3c480	e96f89dd48cbd851
	858d832ae0e8911c	cc5100997f5710f2	639cb938e14dc190	f0996439e7b50cb1
$t = 45$:	021fbadbabab5ac6	c47c9d5c7ea8a234	35d7e7f41defcbd5	bc05ba8de5d3c480
	e95c2a57572d64d9	858d832ae0e8911c	cc5100997f5710f2	639cb938e14dc190
$t = 46$:	f61e672694de2d67	021fbadbabab5ac6	c47c9d5c7ea8a234	35d7e7f41defcbd5
	c6bc35740d8daa9a	e95c2a57572d64d9	858d832ae0e8911c	cc5100997f5710f2
$t = 47$:	6b69fc1bb482feac	f61e672694de2d67	021fbadbabab5ac6	c47c9d5c7ea8a234
	35264334c03ac8ad	c6bc35740d8daa9a	e95c2a57572d64d9	858d832ae0e8911c
$t = 48$:	571f323d96b3a047	6b69fc1bb482feac	f61e672694de2d67	021fbadbabab5ac6
	271580ed6c3e5650	35264334c03ac8ad	c6bc35740d8daa9a	e95c2a57572d64d9
$t = 49$:	ca9bd862c5050918	571f323d96b3a047	6b69fc1bb482feac	f61e672694de2d67
	dfe091dab182e645	271580ed6c3e5650	35264334c03ac8ad	c6bc35740d8daa9a
$t = 50$:	813a43dd2c502043	ca9bd862c5050918	571f323d96b3a047	6b69fc1bb482feac
	07a0d8ef821c5e1a	dfe091dab182e645	271580ed6c3e5650	35264334c03ac8ad
$t = 51$:	d43f83727325dd77	813a43dd2c502043	ca9bd862c5050918	571f323d96b3a047
	483f80a82eaee23e	07a0d8ef821c5e1a	dfe091dab182e645	271580ed6c3e5650
$t = 52$:	03df11b32d42e203	d43f83727325dd77	813a43dd2c502043	ca9bd862c5050918
	504f94e40591cffa	483f80a82eaee23e	07a0d8ef821c5e1a	dfe091dab182e645
$t = 53$:	d63f68037ddf06aa	03df11b32d42e203	d43f83727325dd77	813a43dd2c502043
	a6781efe1aa1ce02	504f94e40591cffa	483f80a82eaee23e	07a0d8ef821c5e1a
$t = 54$:	f650857b5babda4d	d63f68037ddf06aa	03df11b32d42e203	d43f83727325dd77
	9ccfb31a86df0f86	a6781efe1aa1ce02	504f94e40591cffa	483f80a82eaee23e
$t = 55$:	63b460e42748817e	f650857b5babda4d	d63f68037ddf06aa	03df11b32d42e203
	c6b4dd2a9931c509	9ccfb31a86df0f86	a6781efe1aa1ce02	504f94e40591cffa
$t = 56$:	7a52912943d52b05	63b460e42748817e	f650857b5babda4d	d63f68037ddf06aa
	d2e89bbd91e00be0	c6b4dd2a9931c509	9ccfb31a86df0f86	a6781efe1aa1ce02
$t = 57$:	4b81c3aec976ea4b	7a52912943d52b05	63b460e42748817e	f650857b5babda4d
	70505988124351ac	d2e89bbd91e00be0	c6b4dd2a9931c509	9ccfb31a86df0f86
$t = 58$:	581ecb3355dcd9b8	4b81c3aec976ea4b	7a52912943d52b05	63b460e42748817e
	6a3c9b0f71c8bf36	70505988124351ac	d2e89bbd91e00be0	c6b4dd2a9931c509
$t = 59$:	2c074484ef1eac8c	581ecb3355dcd9b8	4b81c3aec976ea4b	7a52912943d52b05
	4797cde4ed370692	6a3c9b0f71c8bf36	70505988124351ac	d2e89bbd91e00be0
$t = 60$:	3857dfd2fc37d3ba	2c074484ef1eac8c	581ecb3355dcd9b8	4b81c3aec976ea4b
	a6af4e9c9f807e51	4797cde4ed370692	6a3c9b0f71c8bf36	70505988124351ac
$t = 61$:	cfcd928c5424e2b6	3857dfd2fc37d3ba	2c074484ef1eac8c	581ecb3355dcd9b8
	09aee5bda1644de5	a6af4e9c9f807e51	4797cde4ed370692	6a3c9b0f71c8bf36
$t = 62$:	a81dedbb9f19e643	cfcd928c5424e2b6	3857dfd2fc37d3ba	2c074484ef1eac8c
	84058865d60a05fa	09aee5bda1644de5	a6af4e9c9f807e51	4797cde4ed370692
$t = 63$:	ab44e86276478d85	a81dedbb9f19e643	cfcd928c5424e2b6	3857dfd2fc37d3ba
	cd881ee59ca6bc53	84058865d60a05fa	09aee5bda1644de5	a6af4e9c9f807e51
$t = 64$:	5a806d7e9821a501	ab44e86276478d85	a81dedbb9f19e643	cfcd928c5424e2b6
	aa84b086688a5c45	cd881ee59ca6bc53	84058865d60a05fa	09aee5bda1644de5
$t = 65$:	eeb9c21bb0102598	5a806d7e9821a501	ab44e86276478d85	a81dedbb9f19e643
	3b5fed0d6a1f96e1	aa84b086688a5c45	cd881ee59ca6bc53	84058865d60a05fa
$t = 66$:	46c4210ab2cc155d	eeb9c21bb0102598	5a806d7e9821a501	ab44e86276478d85
	29fab5a7bff53366	3b5fed0d6a1f96e1	aa84b086688a5c45	cd881ee59ca6bc53

$t = 67$:	54ba35cf56a0340e	46c4210ab2cc155d	eeb9c21bb0102598	5a806d7e9821a501
	1c66f46d95690bcf	29fab5a7bff53366	3b5fed0d6a1f96e1	aa84b086688a5c45
$t = 68$:	181839d609c79748	54ba35cf56a0340e	46c4210ab2cc155d	eeb9c21bb0102598
	0ada78ba2d446140	1c66f46d95690bcf	29fab5a7bff53366	3b5fed0d6a1f96e1
$t = 69$:	fb6aaae5d0b6a447	181839d609c79748	54ba35cf56a0340e	46c4210ab2cc155d
	e3711cb6564d112d	0ada78ba2d446140	1c66f46d95690bcf	29fab5a7bff53366
$t = 70$:	7652c579cb60f19c	fb6aaae5d0b6a447	181839d609c79748	54ba35cf56a0340e
	aff62c9665ff80fa	e3711cb6564d112d	0ada78ba2d446140	1c66f46d95690bcf
$t = 71$:	f15e9664b2803575	7652c579cb60f19c	fb6aaae5d0b6a447	181839d609c79748
	947c3dfafee570ef	aff62c9665ff80fa	e3711cb6564d112d	0ada78ba2d446140
$t = 72$:	358406d165aee9ab	f15e9664b2803575	7652c579cb60f19c	fb6aaae5d0b6a447
	8c7b5fd91a794ca0	947c3dfafee570ef	aff62c9665ff80fa	e3711cb6564d112d
$t = 73$:	20878dcd29cdfaf5	358406d165aee9ab	f15e9664b2803575	7652c579cb60f19c
	054d3536539948d0	8c7b5fd91a794ca0	947c3dfafee570ef	aff62c9665ff80fa
$t = 74$:	33d48dabb5521de2	20878dcd29cdfaf5	358406d165aee9ab	f15e9664b2803575
	2ba18245b50de4cf	054d3536539948d0	8c7b5fd91a794ca0	947c3dfafee570ef
$t = 75$:	c8960e6be864b916	33d48dabb5521de2	20878dcd29cdfaf5	358406d165aee9ab
	995019a6ff3ba3de	2ba18245b50de4cf	054d3536539948d0	8c7b5fd91a794ca0
$t = 76$:	654ef9abec389ca9	c8960e6be864b916	33d48dabb5521de2	20878dcd29cdfaf5
	ceb9fc3691ce8326	995019a6ff3ba3de	2ba18245b50de4cf	054d3536539948d0
$t = 77$:	d67806db8b148677	654ef9abec389ca9	c8960e6be864b916	33d48dabb5521de2
	25c96a7768fb2aa3	ceb9fc3691ce8326	995019a6ff3ba3de	2ba18245b50de4cf
$t = 78$:	10d9c4c4295599f6	d67806db8b148677	654ef9abec389ca9	c8960e6be864b916
	9bb4d39778c07f9e	25c96a7768fb2aa3	ceb9fc3691ce8326	995019a6ff3ba3de
$t = 79$:	73a54f399fa4b1b2	10d9c4c4295599f6	d67806db8b148677	654ef9abec389ca9
	d08446aa79693ed7	9bb4d39778c07f9e	25c96a7768fb2aa3	ceb9fc3691ce8326

That completes the processing of the first and only message block, $M^{(1)}$. The final hash value, $H^{(1)}$, is calculated to be

$$
\begin{aligned}
H_0^{(1)} &= \texttt{6a09e667f3bcc908} + \texttt{73a54f399fa4b1b2} = \texttt{ddaf35a193617aba} \\
H_1^{(1)} &= \texttt{bb67ae8584caa73b} + \texttt{10d9c4c4295599f6} = \texttt{cc417349ae204131} \\
H_2^{(1)} &= \texttt{3c6ef372fe94f82b} + \texttt{d67806db8b148677} = \texttt{12e6fa4e89a97ea2} \\
H_3^{(1)} &= \texttt{a54ff53a5f1d36f1} + \texttt{654ef9abec389ca9} = \texttt{0a9eeee64b55d39a} \\
H_4^{(1)} &= \texttt{510e527fade682d1} + \texttt{d08446aa79693ed7} = \texttt{2192992a274fc1a8} \\
H_5^{(1)} &= \texttt{9b05688c2b3e6c1f} + \texttt{9bb4d39778c07f9e} = \texttt{36ba3c23a3feebbd} \\
H_6^{(1)} &= \texttt{1f83d9abfb41bd6b} + \texttt{25c96a7768fb2aa3} = \texttt{454d4423643ce80e} \\
H_7^{(1)} &= \texttt{5be0cd19137e2179} + \texttt{ceb9fc3691ce8326} = \texttt{2a9ac94fa54ca49f.}
\end{aligned}
$$

The resulting 512-bit message digest is

```
ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a
2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f.
```

C.2 SHA-512 Example (Multi-Block Message)

Let the message, *M*, be the 896-bit ($\ell = 896$) ASCII string

"**abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmn**
hijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu".

The message is padded by appending a "`1`" bit, followed by 1023 "`0`" bits, and ending with the hex value

`0000000000000000 0000000000000380`

(the two 64-bit word representation of the length, 24). Thus, the final padded message consists of two blocks ($N = 2$).

For SHA-512, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = `6a09e667f3bcc908`
$H_1^{(0)}$ = `bb67ae8584caa73b`
$H_2^{(0)}$ = `3c6ef372fe94f82b`
$H_3^{(0)}$ = `a54ff53a5f1d36f1`
$H_4^{(0)}$ = `510e527fade682d1`
$H_5^{(0)}$ = `9b05688c2b3e6c1f`
$H_6^{(0)}$ = `1f83d9abfb41bd6b`
$H_7^{(0)}$ = `5be0cd19137e2179`.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = `6162636465666768`
W_1 = `6263646566676869`
W_2 = `636465666768696a`
W_3 = `6465666768696a6b`
W_4 = `65666768696a6b6c`
W_5 = `666768696a6b6c6d`
W_6 = `6768696a6b6c6d6e`
W_7 = `68696a6b6c6d6e6f`
W_8 = `696a6b6c6d6e6f70`
W_9 = `6a6b6c6d6e6f7071`
W_{10} = `6b6c6d6e6f707172`
W_{11} = `6c6d6e6f70717273`
W_{12} = `6d6e6f7071727374`
W_{13} = `6e6f707172737475`
W_{14} = `8000000000000000`
W_{15} = `0000000000000000`.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 79" loop described in Sec. 6.3.2, step 4.

	a / ***e***	***b*** / ***f***	***c*** / ***g***	***d*** / ***h***
$t = 0$:	f6afce9d2263455d 58cb0218e01b86f9	6a09e667f3bcc908 510e527fade682d1	bb67ae8584caa73b 9b05688c2b3e6c1f	3c6ef372fe94f82b 1f83d9abfb41bd6b
$t = 1$:	0b7056a534ae5f62 f8c7198fe39e4c8c	f6afce9d2263455d 58cb0218e01b86f9	6a09e667f3bcc908 510e527fade682d1	bb67ae8584caa73b 9b05688c2b3e6c1f
$t = 2$:	2ca82233760c9942 303eccccd65953de	0b7056a534ae5f62 f8c7198fe39e4c8c	f6afce9d2263455d 58cb0218e01b86f9	6a09e667f3bcc908 510e527fade682d1
$t = 3$:	a023f17ce52cda7b ffdee5eedcc9ca42	2ca82233760c9942 303eccccd65953de	0b7056a534ae5f62 f8c7198fe39e4c8c	f6afce9d2263455d 58cb0218e01b86f9
$t = 4$:	8f0a67d9d591a1a7 cb4cfbb166505f2f	a023f17ce52cda7b ffdee5eedcc9ca42	2ca82233760c9942 303eccccd65953de	0b7056a534ae5f62 f8c7198fe39e4c8c
$t = 5$:	b466267371acc493 73d6c84c54d399ee	8f0a67d9d591a1a7 cb4cfbb166505f2f	a023f17ce52cda7b ffdee5eedcc9ca42	2ca82233760c9942 303eccccd65953de
$t = 6$:	658269f1a312fccd cdc40314975fb275	b466267371acc493 73d6c84c54d399ee	8f0a67d9d591a1a7 cb4cfbb166505f2f	a023f17ce52cda7b ffdee5eedcc9ca42
$t = 7$:	65e3519c5b88181b a657850ab3970c5a	658269f1a312fccd cdc40314975fb275	b466267371acc493 73d6c84c54d399ee	8f0a67d9d591a1a7 cb4cfbb166505f2f
$t = 8$:	56604fbb4b6393ec e8b3be22fbe64df7	65e3519c5b88181b a657850ab3970c5a	658269f1a312fccd cdc40314975fb275	b466267371acc493 73d6c84c54d399ee
$t = 9$:	c4562769a37d02c0 0062e70a1ef705c1	56604fbb4b6393ec e8b3be22fbe64df7	65e3519c5b88181b a657850ab3970c5a	658269f1a312fccd cdc40314975fb275
$t = 10$:	27c0b4c9186e1736 bc9740477a18ae2d	c4562769a37d02c0 0062e70a1ef705c1	56604fbb4b6393ec e8b3be22fbe64df7	65e3519c5b88181b a657850ab3970c5a
$t = 11$:	f17f52fb02f4eb74 be58522cb9590ee1	27c0b4c9186e1736 bc9740477a18ae2d	c4562769a37d02c0 0062e70a1ef705c1	56604fbb4b6393ec e8b3be22fbe64df7
$t = 12$:	f2c245ac903d4a35 49d5fa3a16dcd502	f17f52fb02f4eb74 be58522cb9590ee1	27c0b4c9186e1736 bc9740477a18ae2d	c4562769a37d02c0 0062e70a1ef705c1
$t = 13$:	9b04175ea8090daa ec9c5e98ff98760d	f2c245ac903d4a35 49d5fa3a16dcd502	f17f52fb02f4eb74 be58522cb9590ee1	27c0b4c9186e1736 bc9740477a18ae2d
$t = 14$:	481b8a6ee5e07031 e4d35b613a5ac420	9b04175ea8090daa ec9c5e98ff98760d	f2c245ac903d4a35 49d5fa3a16dcd502	f17f52fb02f4eb74 be58522cb9590ee1
$t = 15$:	9356ac3ec3e51459 701f17d27582443b	481b8a6ee5e07031 e4d35b613a5ac420	9b04175ea8090daa ec9c5e98ff98760d	f2c245ac903d4a35 49d5fa3a16dcd502
$t = 16$:	b889ed34abd7aa37 1d05d9ba779a1a78	9356ac3ec3e51459 701f17d27582443b	481b8a6ee5e07031 e4d35b613a5ac420	9b04175ea8090daa ec9c5e98ff98760d
$t = 17$:	bf537b1f3edc7381 c362ff9cf932951d	b889ed34abd7aa37 1d05d9ba779a1a78	9356ac3ec3e51459 701f17d27582443b	481b8a6ee5e07031 e4d35b613a5ac420
$t = 18$:	d4e44d54e8242ad8 459e4e6888919f36	bf537b1f3edc7381 c362ff9cf932951d	b889ed34abd7aa37 1d05d9ba779a1a78	9356ac3ec3e51459 701f17d27582443b
$t = 19$:	05f3fba454e5de3d caed4b5fa322b984	d4e44d54e8242ad8 459e4e6888919f36	bf537b1f3edc7381 c362ff9cf932951d	b889ed34abd7aa37 1d05d9ba779a1a78
$t = 20$:	cdb73772dc0248bf dc8049afa6acd502	05f3fba454e5de3d caed4b5fa322b984	d4e44d54e8242ad8 459e4e6888919f36	bf537b1f3edc7381 c362ff9cf932951d
$t = 21$:	1d47a3268ff677ed	cdb73772dc0248bf	05f3fba454e5de3d	d4e44d54e8242ad8

	8407818e9b28cc12	dc8049afa6acd502	caed4b5fa322b984	459e4e6888919f36
$t = 22$:	af4e23eb622d0df4	1d47a3268ff677ed	cdb73772dc0248bf	05f3fba454e5de3d
	64b5ae5424598428	8407818e9b28cc12	dc8049afa6acd502	caed4b5fa322b984
$t = 23$:	be50606778de14a6	af4e23eb622d0df4	1d47a3268ff677ed	cdb73772dc0248bf
	0a5d727cc92e7adb	64b5ae5424598428	8407818e9b28cc12	dc8049afa6acd502
$t = 24$:	821e44f6678ac478	be50606778de14a6	af4e23eb622d0df4	1d47a3268ff677ed
	f367e596d0a038a5	0a5d727cc92e7adb	64b5ae5424598428	8407818e9b28cc12
$t = 25$:	0c852b1359a77c18	821e44f6678ac478	be50606778de14a6	af4e23eb622d0df4
	6dec8a3396a80c3f	f367e596d0a038a5	0a5d727cc92e7adb	64b5ae5424598428
$t = 26$:	ebb574fad4b7a7e4	0c852b1359a77c18	821e44f6678ac478	be50606778de14a6
	a241e7efc1eb6ff9	6dec8a3396a80c3f	f367e596d0a038a5	0a5d727cc92e7adb
$t = 27$:	a092821c3cdf08da	ebb574fad4b7a7e4	0c852b1359a77c18	821e44f6678ac478
	c84e849917a7c08e	a241e7efc1eb6ff9	6dec8a3396a80c3f	f367e596d0a038a5
$t = 28$:	82ba2e1a2df2a4f1	a092821c3cdf08da	ebb574fad4b7a7e4	0c852b1359a77c18
	61845f6924789851	c84e849917a7c08e	a241e7efc1eb6ff9	6dec8a3396a80c3f
$t = 29$:	1959ad991c63d06a	82ba2e1a2df2a4f1	a092821c3cdf08da	ebb574fad4b7a7e4
	231faf24910a891a	61845f6924789851	c84e849917a7c08e	a241e7efc1eb6ff9
$t = 30$:	9b32d4cacd9a625b	1959ad991c63d06a	82ba2e1a2df2a4f1	a092821c3cdf08da
	533066919d608799	231faf24910a891a	61845f6924789851	c84e849917a7c08e
$t = 31$:	dc55339f4d841965	9b32d4cacd9a625b	1959ad991c63d06a	82ba2e1a2df2a4f1
	e2517f359998a58d	533066919d608799	231faf24910a891a	61845f6924789851
$t = 32$:	fdebb1283b12514f	dc55339f4d841965	9b32d4cacd9a625b	1959ad991c63d06a
	b1989170a183c661	e2517f359998a58d	533066919d608799	231faf24910a891a
$t = 33$:	b44c7975a83e3334	fdebb1283b12514f	dc55339f4d841965	9b32d4cacd9a625b
	009ad175b8d588a4	b1989170a183c661	e2517f359998a58d	533066919d608799
$t = 34$:	0bac61bfc53d18b7	b44c7975a83e3334	fdebb1283b12514f	dc55339f4d841965
	a7d5416d690557b8	009ad175b8d588a4	b1989170a183c661	e2517f359998a58d
$t = 35$:	392893c22e75856a	0bac61bfc53d18b7	b44c7975a83e3334	fdebb1283b12514f
	7a7c9eb7bc813248	a7d5416d690557b8	009ad175b8d588a4	b1989170a183c661
$t = 36$:	824408631432e09b	392893c22e75856a	0bac61bfc53d18b7	b44c7975a83e3334
	5e696a9fda56d6bf	7a7c9eb7bc813248	a7d5416d690557b8	009ad175b8d588a4
$t = 37$:	a64162f151a8c1cb	824408631432e09b	392893c22e75856a	0bac61bfc53d18b7
	0f57062401dc680b	5e696a9fda56d6bf	7a7c9eb7bc813248	a7d5416d690557b8
$t = 38$:	922537abad1e95a1	a64162f151a8c1cb	824408631432e09b	392893c22e75856a
	4f4c193d435ff721	0f57062401dc680b	5e696a9fda56d6bf	7a7c9eb7bc813248
$t = 39$:	b80591f6fbfadcde	922537abad1e95a1	a64162f151a8c1cb	824408631432e09b
	00f4407c0f37237e	4f4c193d435ff721	0f57062401dc680b	5e696a9fda56d6bf
$t = 40$:	08f151f4b8d0fa2e	b80591f6fbfadcde	922537abad1e95a1	a64162f151a8c1cb
	ec8b96fe402094cd	00f4407c0f37237e	4f4c193d435ff721	0f57062401dc680b
$t = 41$:	12b5fcc2b68f65c0	08f151f4b8d0fa2e	b80591f6fbfadcde	922537abad1e95a1
	d688101dfd24a148	ec8b96fe402094cd	00f4407c0f37237e	4f4c193d435ff721
$t = 42$:	a71bf5bd64289948	12b5fcc2b68f65c0	08f151f4b8d0fa2e	b80591f6fbfadcde
	e052bfb7a6945939	d688101dfd24a148	ec8b96fe402094cd	00f4407c0f37237e
$t = 43$:	890c2cd670c4aea3	a71bf5bd64289948	12b5fcc2b68f65c0	08f151f4b8d0fa2e
	dd13e4edeeff00e7	e052bfb7a6945939	d688101dfd24a148	ec8b96fe402094cd
$t = 44$:	ca61990b43297ffc	890c2cd670c4aea3	a71bf5bd64289948	12b5fcc2b68f65c0

	139aa55c51d9ee5f	dd13e4edeeff00e7	e052bfb7a6945939	d688101dfd24a148
$t = 45$:	7196e8fa538ba4bf	ca61990b43297ffc	890c2cd670c4aea3	a71bf5bd64289948
	046735513cdd14d3	139aa55c51d9ee5f	dd13e4edeeff00e7	e052bfb7a6945939
$t = 46$:	1f0720944dbeb6a4	7196e8fa538ba4bf	ca61990b43297ffc	890c2cd670c4aea3
	a41eb7e5a27588e3	046735513cdd14d3	139aa55c51d9ee5f	dd13e4edeeff00e7
$t = 47$:	d6d4f8608b8ab199	1f0720944dbeb6a4	7196e8fa538ba4bf	ca61990b43297ffc
	24b9c216f915da60	a41eb7e5a27588e3	046735513cdd14d3	139aa55c51d9ee5f
$t = 48$:	88761eb67845978e	d6d4f8608b8ab199	1f0720944dbeb6a4	7196e8fa538ba4bf
	9fe22e39448d50ed	24b9c216f915da60	a41eb7e5a27588e3	046735513cdd14d3
$t = 49$:	7d40e6be47d85702	88761eb67845978e	d6d4f8608b8ab199	1f0720944dbeb6a4
	d9c900e01968c33e	9fe22e39448d50ed	24b9c216f915da60	a41eb7e5a27588e3
$t = 50$:	7d0d988df5768598	7d40e6be47d85702	88761eb67845978e	d6d4f8608b8ab199
	2ec2e522a7c7d12c	d9c900e01968c33e	9fe22e39448d50ed	24b9c216f915da60
$t = 51$:	48a8b60575b37f31	7d0d988df5768598	7d40e6be47d85702	88761eb67845978e
	7059f9bc8c88a373	2ec2e522a7c7d12c	d9c900e01968c33e	9fe22e39448d50ed
$t = 52$:	6bc425af294bbf79	48a8b60575b37f31	7d0d988df5768598	7d40e6be47d85702
	6a8143b1716ee33d	7059f9bc8c88a373	2ec2e522a7c7d12c	d9c900e01968c33e
$t = 53$:	307a456158ee8849	6bc425af294bbf79	48a8b60575b37f31	7d0d988df5768598
	4372e85c16ee4440	6a8143b1716ee33d	7059f9bc8c88a373	2ec2e522a7c7d12c
$t = 54$:	af36382c8fd716be	307a456158ee8849	6bc425af294bbf79	48a8b60575b37f31
	a8f8b0033187a916	4372e85c16ee4440	6a8143b1716ee33d	7059f9bc8c88a373
$t = 55$:	810ebee951c64ca1	af36382c8fd716be	307a456158ee8849	6bc425af294bbf79
	16a64f5997b9cca6	a8f8b0033187a916	4372e85c16ee4440	6a8143b1716ee33d
$t = 56$:	2dd7659f1b4d13cd	810ebee951c64ca1	af36382c8fd716be	307a456158ee8849
	5da6793bb7286a4b	16a64f5997b9cca6	a8f8b0033187a916	4372e85c16ee4440
$t = 57$:	5ac712acff4b98be	2dd7659f1b4d13cd	810ebee951c64ca1	af36382c8fd716be
	91f6395b301adbfd	5da6793bb7286a4b	16a64f5997b9cca6	a8f8b0033187a916
$t = 58$:	c1af358833cb03c0	5ac712acff4b98be	2dd7659f1b4d13cd	810ebee951c64ca1
	d4883c0c21dda190	91f6395b301adbfd	5da6793bb7286a4b	16a64f5997b9cca6
$t = 59$:	88a306074d388c7d	c1af358833cb03c0	5ac712acff4b98be	2dd7659f1b4d13cd
	9fc52468b897f9c8	d4883c0c21dda190	91f6395b301adbfd	5da6793bb7286a4b
$t = 60$:	f11bfd0cf67d3040	88a306074d388c7d	c1af358833cb03c0	5ac712acff4b98be
	47efb6407f74d318	9fc52468b897f9c8	d4883c0c21dda190	91f6395b301adbfd
$t = 61$:	1f065e7828ed4e1b	f11bfd0cf67d3040	88a306074d388c7d	c1af358833cb03c0
	7481899904a4ce23	47efb6407f74d318	9fc52468b897f9c8	d4883c0c21dda190
$t = 62$:	aebde39f2bc42ec1	1f065e7828ed4e1b	f11bfd0cf67d3040	88a306074d388c7d
	62ab526ff177a988	7481899904a4ce23	47efb6407f74d318	9fc52468b897f9c8
$t = 63$:	d35a94706e3e5df2	aebde39f2bc42ec1	1f065e7828ed4e1b	f11bfd0cf67d3040
	53f92b648d5d815c	62ab526ff177a988	7481899904a4ce23	47efb6407f74d318
$t = 64$:	d72d727c53e09ab9	d35a94706e3e5df2	aebde39f2bc42ec1	1f065e7828ed4e1b
	10746426ba9824f4	53f92b648d5d815c	62ab526ff177a988	7481899904a4ce23
$t = 65$:	3a7235e5a4051d94	d72d727c53e09ab9	d35a94706e3e5df2	aebde39f2bc42ec1
	afe455daec5c2b00	10746426ba9824f4	53f92b648d5d815c	62ab526ff177a988
$t = 66$:	f7f510fe73ef7e76	3a7235e5a4051d94	d72d727c53e09ab9	d35a94706e3e5df2
	f1202c0bb7c4583f	afe455daec5c2b00	10746426ba9824f4	53f92b648d5d815c
$t = 67$:	23c2acfb393523e9	f7f510fe73ef7e76	3a7235e5a4051d94	d72d727c53e09ab9
	a0bc2a61044ac12e	f1202c0bb7c4583f	afe455daec5c2b00	10746426ba9824f4

$t = 68$:	0307d241a1ed7121	23c2acfb393523e9	f7f510fe73ef7e76	3a7235e5a4051d94
	fad5f38f1e0aea12	a0bc2a61044ac12e	f1202c0bb7c4583f	afe455daec5c2b00
$t = 69$:	191814d82f0a16fb	0307d241a1ed7121	23c2acfb393523e9	f7f510fe73ef7e76
	39d325086e66e200	fad5f38f1e0aea12	a0bc2a61044ac12e	f1202c0bb7c4583f
$t = 70$:	0a1ed41b6da18c01	191814d82f0a16fb	0307d241a1ed7121	23c2acfb393523e9
	b3d3521e166e5df1	39d325086e66e200	fad5f38f1e0aea12	a0bc2a61044ac12e
$t = 71$:	8a3f07db93f6c827	0a1ed41b6da18c01	191814d82f0a16fb	0307d241a1ed7121
	6b370074be040ed7	b3d3521e166e5df1	39d325086e66e200	fad5f38f1e0aea12
$t = 72$:	002744d87ef80d28	8a3f07db93f6c827	0a1ed41b6da18c01	191814d82f0a16fb
	8c5a245de2d72fe6	6b370074be040ed7	b3d3521e166e5df1	39d325086e66e200
$t = 73$:	778dc7880a4a2aa0	002744d87ef80d28	8a3f07db93f6c827	0a1ed41b6da18c01
	45a375b466e5e342	8c5a245de2d72fe6	6b370074be040ed7	b3d3521e166e5df1
$t = 74$:	a3f11de5ede05b11	778dc7880a4a2aa0	002744d87ef80d28	8a3f07db93f6c827
	f5bbf52f1ab7cc05	45a375b466e5e342	8c5a245de2d72fe6	6b370074be040ed7
$t = 75$:	629c8ae6ecd8af4b	a3f11de5ede05b11	778dc7880a4a2aa0	002744d87ef80d28
	5a8fe5919d3cf136	f5bbf52f1ab7cc05	45a375b466e5e342	8c5a245de2d72fe6
$t = 76$:	c9a8c1e2d063ce94	629c8ae6ecd8af4b	a3f11de5ede05b11	778dc7880a4a2aa0
	aacd089bfae8faf9	5a8fe5919d3cf136	f5bbf52f1ab7cc05	45a375b466e5e342
$t = 77$:	c517cba6a09bb26a	c9a8c1e2d063ce94	629c8ae6ecd8af4b	a3f11de5ede05b11
	e1682bd33c8f8e23	aacd089bfae8faf9	5a8fe5919d3cf136	f5bbf52f1ab7cc05
$t = 78$:	11e3570e06e3b74e	c517cba6a09bb26a	c9a8c1e2d063ce94	629c8ae6ecd8af4b
	075aabbade34fd01	e1682bd33c8f8e23	aacd089bfae8faf9	5a8fe5919d3cf136
$t = 79$:	d90f1b1237b3a561	11e3570e06e3b74e	c517cba6a09bb26a	c9a8c1e2d063ce94
	867983f69d3a3ad1	075aabbade34fd01	e1682bd33c8f8e23	aacd089bfae8faf9

That completes the processing of the first message block, $M^{(1)}$. The intermediate hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{6a09e667f3bcc908} + \texttt{d90f1b1237b3a561} = \texttt{4319017a2b706e69}$$
$$H_1^{(1)} = \texttt{bb67ae8584caa73b} + \texttt{11e3570e06e3b74e} = \texttt{cd4b05938bae5e89}$$
$$H_2^{(1)} = \texttt{3c6ef372fe94f82b} + \texttt{c517cba6a09bb26a} = \texttt{0186bf199f30aa95}$$
$$H_3^{(1)} = \texttt{a54ff53a5f1d36f1} + \texttt{c9a8c1e2d063ce94} = \texttt{6ef8b71d2f810585}$$
$$H_4^{(1)} = \texttt{510e527fade682d1} + \texttt{867983f69d3a3ad1} = \texttt{d787d6764b20bda2}$$
$$H_5^{(1)} = \texttt{9b05688c2b3e6c1f} + \texttt{075aabbade34fd01} = \texttt{a260144709736920}$$
$$H_6^{(1)} = \texttt{1f83d9abfb41bd6b} + \texttt{e1682bd33c8f8e23} = \texttt{00ec057f37d14b8e}$$
$$H_7^{(1)} = \texttt{5be0cd19137e2179} + \texttt{aacd089bfae8faf9} = \texttt{06add5b50e671c72}.$$

The words of the *second* padded message block, $M^{(2)}$, are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 =	0000000000000000	W_8 =	0000000000000000
W_1 =	0000000000000000	W_9 =	0000000000000000
W_2 =	0000000000000000	W_{10} =	0000000000000000
W_3 =	0000000000000000	W_{11} =	0000000000000000
W_4 =	0000000000000000	W_{12} =	0000000000000000
W_5 =	0000000000000000	W_{13} =	0000000000000000
W_6 =	0000000000000000	W_{14} =	0000000000000000
W_7 =	0000000000000000	W_{15} =	0000000000000380.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 79" loop described in Sec. 6.1.2, step 4.

	a / ***e***	***b*** / ***f***	***c*** / ***g***	***d*** / ***h***
t = 0 :	b8fdb92bdfb187e8 1d5f4d5ad031b8e6	4319017a2b706e69 d787d6764b20bda2	cd4b05938bae5e89 a260144709736920	0186bf199f30aa95 00ec057f37d14b8e
t = 1 :	6eb90718369c5cd7 4b9b4877d987b0fe	b8fdb92bdfb187e8 1d5f4d5ad031b8e6	4319017a2b706e69 d787d6764b20bda2	cd4b05938bae5e89 a260144709736920
t = 2 :	c83451f2335d5144 d6b67350e0781e99	6eb90718369c5cd7 4b9b4877d987b0fe	b8fdb92bdfb187e8 1d5f4d5ad031b8e6	4319017a2b706e69 d787d6764b20bda2
t = 3 :	28ec1deb2a9ee6e3 25e3136be5999b8c	c83451f2335d5144 d6b67350e0781e99	6eb90718369c5cd7 4b9b4877d987b0fe	b8fdb92bdfb187e8 1d5f4d5ad031b8e6
t = 4 :	806abd86c0479e5b 1b8f7670eab1cf89	28ec1deb2a9ee6e3 25e3136be5999b8c	c83451f2335d5144 d6b67350e0781e99	6eb90718369c5cd7 4b9b4877d987b0fe
t = 5 :	234788f8a54aed38 4fabe51c67d5d156	806abd86c0479e5b 1b8f7670eab1cf89	28ec1deb2a9ee6e3 25e3136be5999b8c	c83451f2335d5144 d6b67350e0781e99
t = 6 :	01264f18257b5e2c 1c3506096b99de50	234788f8a54aed38 4fabe51c67d5d156	806abd86c0479e5b 1b8f7670eab1cf89	28ec1deb2a9ee6e3 25e3136be5999b8c
t = 7 :	5b14f38104dde991 13f8bfdc4001c362	01264f18257b5e2c 1c3506096b99de50	234788f8a54aed38 4fabe51c67d5d156	806abd86c0479e5b 1b8f7670eab1cf89
t = 8 :	f522574a41b2aac6 63a5f09617622ed2	5b14f38104dde991 13f8bfdc4001c362	01264f18257b5e2c 1c3506096b99de50	234788f8a54aed38 4fabe51c67d5d156
t = 9 :	6ec258b855afae5a 211e271d92770b36	f522574a41b2aac6 63a5f09617622ed2	5b14f38104dde991 13f8bfdc4001c362	01264f18257b5e2c 1c3506096b99de50
t = 10 :	9364214ba48b416c d64dcb6ec0fe5bac	6ec258b855afae5a 211e271d92770b36	f522574a41b2aac6 63a5f09617622ed2	5b14f38104dde991 13f8bfdc4001c362
t = 11 :	082ba62147ecbbd5 34fe78473b61266e	9364214ba48b416c d64dcb6ec0fe5bac	6ec258b855afae5a 211e271d92770b36	f522574a41b2aac6 63a5f09617622ed2
t = 12 :	5790f6ba82bba809 d491e309141dcaa3	082ba62147ecbbd5 34fe78473b61266e	9364214ba48b416c d64dcb6ec0fe5bac	6ec258b855afae5a 211e271d92770b36
t = 13 :	a6b8aefd086d33ce 044943c2992cc0f0	5790f6ba82bba809 d491e309141dcaa3	082ba62147ecbbd5 34fe78473b61266e	9364214ba48b416c d64dcb6ec0fe5bac
t = 14 :	bf2324a9a363abe7 0cf5f4bde5977c54	a6b8aefd086d33ce 044943c2992cc0f0	5790f6ba82bba809 d491e309141dcaa3	082ba62147ecbbd5 34fe78473b61266e
t = 15 :	00e8e32076a61aff	bf2324a9a363abe7	a6b8aefd086d33ce	5790f6ba82bba809

	43bf4eb269a2650c	0cf5f4bde5977c54	044943c2992cc0f0	d491e309141dcaa3
$t = 16$:	f0376dff66fff4a7	00e8e32076a61aff	bf2324a9a363abe7	a6b8aefd086d33ce
	69fa5896969e85b8	43bf4eb269a2650c	0cf5f4bde5977c54	044943c2992cc0f0
$t = 17$:	2fad194272cda857	f0376dff66fff4a7	00e8e32076a61aff	bf2324a9a363abe7
	ddb519d663b7b6ec	69fa5896969e85b8	43bf4eb269a2650c	0cf5f4bde5977c54
$t = 18$:	9ae56936e95325ac	2fad194272cda857	f0376dff66fff4a7	00e8e32076a61aff
	04ceb04676619057	ddb519d663b7b6ec	69fa5896969e85b8	43bf4eb269a2650c
$t = 19$:	d94ccb853f53433b	9ae56936e95325ac	2fad194272cda857	f0376dff66fff4a7
	dcdc0f45813fb5a2	04ceb04676619057	ddb519d663b7b6ec	69fa5896969e85b8
$t = 20$:	837f8075d2945995	d94ccb853f53433b	9ae56936e95325ac	2fad194272cda857
	272b5f79a91419d8	dcdc0f45813fb5a2	04ceb04676619057	ddb519d663b7b6ec
$t = 21$:	786bde689f7aa62d	837f8075d2945995	d94ccb853f53433b	9ae56936e95325ac
	566586e69ad3f487	272b5f79a91419d8	dcdc0f45813fb5a2	04ceb04676619057
$t = 22$:	276457f01812aa6f	786bde689f7aa62d	837f8075d2945995	d94ccb853f53433b
	e78fb8b0dfbbc62f	566586e69ad3f487	272b5f79a91419d8	dcdc0f45813fb5a2
$t = 23$:	0de519f5d6c2c298	276457f01812aa6f	786bde689f7aa62d	837f8075d2945995
	5ca3e5cd1a30b954	e78fb8b0dfbbc62f	566586e69ad3f487	272b5f79a91419d8
$t = 24$:	54314dff825e2b22	0de519f5d6c2c298	276457f01812aa6f	786bde689f7aa62d
	b81a51e0c96ccf77	5ca3e5cd1a30b954	e78fb8b0dfbbc62f	566586e69ad3f487
$t = 25$:	5d3f98dd7b29c363	54314dff825e2b22	0de519f5d6c2c298	276457f01812aa6f
	95d49494f5a0d14a	b81a51e0c96ccf77	5ca3e5cd1a30b954	e78fb8b0dfbbc62f
$t = 26$:	5e9da426aa7d4a58	5d3f98dd7b29c363	54314dff825e2b22	0de519f5d6c2c298
	d22cccad2e391cd4	95d49494f5a0d14a	b81a51e0c96ccf77	5ca3e5cd1a30b954
$t = 27$:	3b62dd973298ea43	5e9da426aa7d4a58	5d3f98dd7b29c363	54314dff825e2b22
	aceb5d06101e514e	d22cccad2e391cd4	95d49494f5a0d14a	b81a51e0c96ccf77
$t = 28$:	fd258ff809b2253d	3b62dd973298ea43	5e9da426aa7d4a58	5d3f98dd7b29c363
	26c991e85352da6f	aceb5d06101e514e	d22cccad2e391cd4	95d49494f5a0d14a
$t = 29$:	b462a20846af417d	fd258ff809b2253d	3b62dd973298ea43	5e9da426aa7d4a58
	291eee54c034c326	26c991e85352da6f	aceb5d06101e514e	d22cccad2e391cd4
$t = 30$:	d5471e3dc7171224	b462a20846af417d	fd258ff809b2253d	3b62dd973298ea43
	0aaf99c59e7fadbd	291eee54c034c326	26c991e85352da6f	aceb5d06101e514e
$t = 31$:	9ace856ba1290e6e	d5471e3dc7171224	b462a20846af417d	fd258ff809b2253d
	658f0bea63804d05	0aaf99c59e7fadbd	291eee54c034c326	26c991e85352da6f
$t = 32$:	80a0d154506b37c4	9ace856ba1290e6e	d5471e3dc7171224	b462a20846af417d
	bbe6e3b3bb7fefab	658f0bea63804d05	0aaf99c59e7fadbd	291eee54c034c326
$t = 33$:	fb90a8a76dea1bfe	80a0d154506b37c4	9ace856ba1290e6e	d5471e3dc7171224
	65234d5b5049e665	bbe6e3b3bb7fefab	658f0bea63804d05	0aaf99c59e7fadbd
$t = 34$:	f517b690d940a294	fb90a8a76dea1bfe	80a0d154506b37c4	9ace856ba1290e6e
	e4dd663f44d313bc	65234d5b5049e665	bbe6e3b3bb7fefab	658f0bea63804d05
$t = 35$:	b70883992932880d	f517b690d940a294	fb90a8a76dea1bfe	80a0d154506b37c4
	dc5dd7c12b1cb6e3	e4dd663f44d313bc	65234d5b5049e665	bbe6e3b3bb7fefab
$t = 36$:	b2a2be77b0fcf3bf	b70883992932880d	f517b690d940a294	fb90a8a76dea1bfe
	50fca57291e19874	dc5dd7c12b1cb6e3	e4dd663f44d313bc	65234d5b5049e665
$t = 37$:	8575839b0f08472b	b2a2be77b0fcf3bf	b70883992932880d	f517b690d940a294
	bd7176bd099bb2f2	50fca57291e19874	dc5dd7c12b1cb6e3	e4dd663f44d313bc
$t = 38$:	4405d2765de0adfc	8575839b0f08472b	b2a2be77b0fcf3bf	b70883992932880d

	7ca4916f2cd8db10	bd7176bd099bb2f2	50fca57291e19874	dc5dd7c12b1cb6e3
$t = 39$:	eec6fca5aa657661	4405d2765de0adfc	8575839b0f08472b	b2a2be77b0fcf3bf
	7be0b7e70bdabe53	7ca4916f2cd8db10	bd7176bd099bb2f2	50fca57291e19874
$t = 40$:	bb3fcd7585b59e32	eec6fca5aa657661	4405d2765de0adfc	8575839b0f08472b
	2201c7cbd34e31fe	7be0b7e70bdabe53	7ca4916f2cd8db10	bd7176bd099bb2f2
$t = 41$:	0e109efc47927341	bb3fcd7585b59e32	eec6fca5aa657661	4405d2765de0adfc
	d43e5686506fa05d	2201c7cbd34e31fe	7be0b7e70bdabe53	7ca4916f2cd8db10
$t = 42$:	55c0dba83bcdc6e0	0e109efc47927341	bb3fcd7585b59e32	eec6fca5aa657661
	5b634502f1671535	d43e5686506fa05d	2201c7cbd34e31fe	7be0b7e70bdabe53
$t = 43$:	f5756f847bfaef67	55c0dba83bcdc6e0	0e109efc47927341	bb3fcd7585b59e32
	e2d307fd94f4818a	5b634502f1671535	d43e5686506fa05d	2201c7cbd34e31fe
$t = 44$:	f1438c9cf271c06e	f5756f847bfaef67	55c0dba83bcdc6e0	0e109efc47927341
	ad8ac1ed966b2dc6	e2d307fd94f4818a	5b634502f1671535	d43e5686506fa05d
$t = 45$:	a7dcaffdbefb9d4a	f1438c9cf271c06e	f5756f847bfaef67	55c0dba83bcdc6e0
	9e46e9f915099c34	ad8ac1ed966b2dc6	e2d307fd94f4818a	5b634502f1671535
$t = 46$:	985ba373680b8e94	a7dcaffdbefb9d4a	f1438c9cf271c06e	f5756f847bfaef67
	7d4c0abc676b1a8b	9e46e9f915099c34	ad8ac1ed966b2dc6	e2d307fd94f4818a
$t = 47$:	807f45784852303f	985ba373680b8e94	a7dcaffdbefb9d4a	f1438c9cf271c06e
	082ee70d3f352aac	7d4c0abc676b1a8b	9e46e9f915099c34	ad8ac1ed966b2dc6
$t = 48$:	d9c523173b1a1e05	807f45784852303f	985ba373680b8e94	a7dcaffdbefb9d4a
	e301dca32c44ca05	082ee70d3f352aac	7d4c0abc676b1a8b	9e46e9f915099c34
$t = 49$:	b6df019ca515cafb	d9c523173b1a1e05	807f45784852303f	985ba373680b8e94
	754b3a461a665640	e301dca32c44ca05	082ee70d3f352aac	7d4c0abc676b1a8b
$t = 50$:	427a642921b2e645	b6df019ca515cafb	d9c523173b1a1e05	807f45784852303f
	08a30fefe981f2ec	754b3a461a665640	e301dca32c44ca05	082ee70d3f352aac
$t = 51$:	7aab58dbe1b9df7b	427a642921b2e645	b6df019ca515cafb	d9c523173b1a1e05
	2749c52d0b3d1225	08a30fefe981f2ec	754b3a461a665640	e301dca32c44ca05
$t = 52$:	974ddd552aec16ce	7aab58dbe1b9df7b	427a642921b2e645	b6df019ca515cafb
	a9e6cbfb416a591f	2749c52d0b3d1225	08a30fefe981f2ec	754b3a461a665640
$t = 53$:	55e0b99d4404f6ca	974ddd552aec16ce	7aab58dbe1b9df7b	427a642921b2e645
	6c24ad697b41b1b9	a9e6cbfb416a591f	2749c52d0b3d1225	08a30fefe981f2ec
$t = 54$:	901f632579ee1eee	55e0b99d4404f6ca	974ddd552aec16ce	7aab58dbe1b9df7b
	4ee99476db1bb7a9	6c24ad697b41b1b9	a9e6cbfb416a591f	2749c52d0b3d1225
$t = 55$:	f90db9f292a60463	901f632579ee1eee	55e0b99d4404f6ca	974ddd552aec16ce
	5401644992a1f8b8	4ee99476db1bb7a9	6c24ad697b41b1b9	a9e6cbfb416a591f
$t = 56$:	9b906a7df1007357	f90db9f292a60463	901f632579ee1eee	55e0b99d4404f6ca
	f5e402ee21db8915	5401644992a1f8b8	4ee99476db1bb7a9	6c24ad697b41b1b9
$t = 57$:	71a0a998fb48c0fc	9b906a7df1007357	f90db9f292a60463	901f632579ee1eee
	96bece755cd203cb	f5e402ee21db8915	5401644992a1f8b8	4ee99476db1bb7a9
$t = 58$:	c25e798e50752535	71a0a998fb48c0fc	9b906a7df1007357	f90db9f292a60463
	9d548440d8e110f2	96bece755cd203cb	f5e402ee21db8915	5401644992a1f8b8
$t = 59$:	1ce4f2591812e6ae	c25e798e50752535	71a0a998fb48c0fc	9b906a7df1007357
	b27252537a83cf27	9d548440d8e110f2	96bece755cd203cb	f5e402ee21db8915
$t = 60$:	c1700e250dc6ffed	1ce4f2591812e6ae	c25e798e50752535	71a0a998fb48c0fc
	970088839126bda5	b27252537a83cf27	9d548440d8e110f2	96bece755cd203cb
$t = 61$:	f8e6924412fd0c64	c1700e250dc6ffed	1ce4f2591812e6ae	c25e798e50752535
	d50cf4f73910e3ee	970088839126bda5	b27252537a83cf27	9d548440d8e110f2

$t = 62$:	d53e0a39eee47528	f8e6924412fd0c64	c1700e250dc6ffed	1ce4f2591812e6ae
	1b6d7234ace15d7d	d50cf4f73910e3ee	970088839126bda5	b27252537a83cf27
$t = 63$:	3960545ab926c0d5	d53e0a39eee47528	f8e6924412fd0c64	c1700e250dc6ffed
	9eabb5618b4fcd13	1b6d7234ace15d7d	d50cf4f73910e3ee	970088839126bda5
$t = 64$:	b2c164d71abb92fe	3960545ab926c0d5	d53e0a39eee47528	f8e6924412fd0c64
	f1736fbbfb6ebe72	9eabb5618b4fcd13	1b6d7234ace15d7d	d50cf4f73910e3ee
$t = 65$:	4d979e985b067e75	b2c164d71abb92fe	3960545ab926c0d5	d53e0a39eee47528
	d1fb300f35992350	f1736fbbfb6ebe72	9eabb5618b4fcd13	1b6d7234ace15d7d
$t = 66$:	59d0238ce137abd7	4d979e985b067e75	b2c164d71abb92fe	3960545ab926c0d5
	5f3c64b7546e2cec	d1fb300f35992350	f1736fbbfb6ebe72	9eabb5618b4fcd13
$t = 67$:	bf8d9453b9876b0a	59d0238ce137abd7	4d979e985b067e75	b2c164d71abb92fe
	6c27893a31b0e07e	5f3c64b7546e2cec	d1fb300f35992350	f1736fbbfb6ebe72
$t = 68$:	c45dd4a2d2fea059	bf8d9453b9876b0a	59d0238ce137abd7	4d979e985b067e75
	48253e21b26d8cf9	6c27893a31b0e07e	5f3c64b7546e2cec	d1fb300f35992350
$t = 69$:	e08471946c17b0b6	c45dd4a2d2fea059	bf8d9453b9876b0a	59d0238ce137abd7
	714e2adf4e23ff24	48253e21b26d8cf9	6c27893a31b0e07e	5f3c64b7546e2cec
$t = 70$:	b4838c1c28fee7bc	e08471946c17b0b6	c45dd4a2d2fea059	bf8d9453b9876b0a
	371f12f333f7e5b9	714e2adf4e23ff24	48253e21b26d8cf9	6c27893a31b0e07e
$t = 71$:	851cf60a77f6e6d1	b4838c1c28fee7bc	e08471946c17b0b6	c45dd4a2d2fea059
	a2a475deac0e8b42	371f12f333f7e5b9	714e2adf4e23ff24	48253e21b26d8cf9
$t = 72$:	f53d23c50249af2d	851cf60a77f6e6d1	b4838c1c28fee7bc	e08471946c17b0b6
	1e99cae9d4cf0409	a2a475deac0e8b42	371f12f333f7e5b9	714e2adf4e23ff24
$t = 73$:	b81e85d427045550	f53d23c50249af2d	851cf60a77f6e6d1	b4838c1c28fee7bc
	f5794711faa60f63	1e99cae9d4cf0409	a2a475deac0e8b42	371f12f333f7e5b9
$t = 74$:	ae70c7d11ea84a83	b81e85d427045550	f53d23c50249af2d	851cf60a77f6e6d1
	dc0d633411c289b2	f5794711faa60f63	1e99cae9d4cf0409	a2a475deac0e8b42
$t = 75$:	5c54592e13c76135	ae70c7d11ea84a83	b81e85d427045550	f53d23c50249af2d
	1620dd5479e94b9b	dc0d633411c289b2	f5794711faa60f63	1e99cae9d4cf0409
$t = 76$:	03a0f79087078a93	5c54592e13c76135	ae70c7d11ea84a83	b81e85d427045550
	57e90fa678e4cc97	1620dd5479e94b9b	dc0d633411c289b2	f5794711faa60f63
$t = 77$:	8df0baad4c6ed50c	03a0f79087078a93	5c54592e13c76135	ae70c7d11ea84a83
	c6e7246f7f0bdac6	57e90fa678e4cc97	1620dd5479e94b9b	dc0d633411c289b2
$t = 78$:	bfa9f194894db5b6	8df0baad4c6ed50c	03a0f79087078a93	5c54592e13c76135
	90bb8597bb41da1a	c6e7246f7f0bdac6	57e90fa678e4cc97	1620dd5479e94b9b
$t = 79$:	4b7c99fbaf72a571	bfa9f194894db5b6	8df0baad4c6ed50c	03a0f79087078a93
	78955227fde03a42	90bb8597bb41da1a	c6e7246f7f0bdac6	57e90fa678e4cc97

That completes the processing of the second and final message block, $M^{(2)}$. The final hash value, $H^{(2)}$, is calculated to be

$$H_0^{(2)} = \texttt{4319017a2b706e69} + \texttt{4b7c99fbaf72a571} = \texttt{8e959b75dae313da}$$
$$H_1^{(2)} = \texttt{cd4b05938bae5e89} + \texttt{bfa9f194894db5b6} = \texttt{8cf4f72814fc143f}$$
$$H_2^{(2)} = \texttt{0186bf199f30aa95} + \texttt{8df0baad4c6ed50c} = \texttt{8f7779c6eb9f7fa1}$$
$$H_3^{(2)} = \texttt{6ef8b71d2f810585} + \texttt{03a0f79087078a93} = \texttt{7299aeadb6889018}$$
$$H_4^{(2)} = \texttt{d787d6764b20bda2} + \texttt{78955227fde03a42} = \texttt{501d289e4900f7e4}$$

$H_5^{(2)}$ = a260144709736920 + 90bb8597bb41da1a = 331b99dec4b5433a

$H_6^{(2)}$ = 00ec057f37d14b8e + c6e7246f7f0bdac6 = c7d329eeb6dd2654

$H_7^{(2)}$ = 06add5b50e671c72 + 57e90fa678e4cc97 = 5e96e55b874be909.

The resulting 512-bit message digest is

```
8e959b75dae313da 8cf4f72814fc143f 8f7779c6eb9f7fa1 7299aeadb6889018
501d289e4900f7e4 331b99dec4b5433a c7d329eeb6dd2654 5e96e55b874be909.
```

C.3 SHA-512 Example (Long Message)

Let the message *M* be the binary-coded form of the ASCII string which consists of 1,000,000 repetitions of the character "**a**". The resulting SHA-512 message digest is

```
e718483d0ce76964 4e2e42c7bc15b463 8e1f98b13b204428 5632a803afa973eb
de0ff244877ea60a 4cb0432ce577c31b eb009c5c2c49aa2e 4eadb217ad8cc09b.
```

APPENDIX D: SHA-384 EXAMPLES

This appendix is for informational purposes only and is not required to meet the standard.

D.1 SHA-384 Example (One-Block Message)

Let the message, M, be the 24-bit ($\ell = 24$) ASCII string "**abc**", which is equivalent to the following binary string:

01100001 01100010 01100011.

The message is padded by appending a "1" bit, followed by 871 "0" bits, and ending with the hex value

0000000000000000 0000000000000018

(the two 64-bit word representation of the length, 24). Thus, the final padded message consists of one block ($N = 1$).

For SHA-384, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = cbbb9d5dc1059ed8
$H_1^{(0)}$ = 629a292a367cd507
$H_2^{(0)}$ = 9159015a3070dd17
$H_3^{(0)}$ = 152fecd8f70e5939
$H_4^{(0)}$ = 67332667ffc00b31
$H_5^{(0)}$ = 8eb44a8768581511
$H_6^{(0)}$ = db0c2e0d64f98fa7
$H_7^{(0)}$ = 47b5481dbefa4fa4.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 6162638000000000
W_1 = 0000000000000000
W_2 = 0000000000000000
W_3 = 0000000000000000
W_4 = 0000000000000000
W_5 = 0000000000000000
W_6 = 0000000000000000
W_7 = 0000000000000000
W_8 = 0000000000000000
W_9 = 0000000000000000
W_{10} = 0000000000000000
W_{11} = 0000000000000000
W_{12} = 0000000000000000
W_{13} = 0000000000000000
W_{14} = 0000000000000000
W_{15} = 0000000000000018.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 79" loop described in Sec. 6.3.2, step 4.

	a / ***e***	***b*** / ***f***	***c*** / ***g***	***d*** / ***h***
t = 0 :	470994ad30873f88 bd03f724be6075f9	cbbb9d5dc1059ed8 67332667ffc00b31	629a292a367cd507 8eb44a8768581511	9159015a3070dd17 db0c2e0d64f98fa7
t = 1 :	2e91230306a12ae0 5e1b4e1695372b9e	470994ad30873f88 bd03f724be6075f9	cbbb9d5dc1059ed8 67332667ffc00b31	629a292a367cd507 8eb44a8768581511
t = 2 :	eebe5d379be707ad 54074a65aef34336	2e91230306a12ae0 5e1b4e1695372b9e	470994ad30873f88 bd03f724be6075f9	cbbb9d5dc1059ed8 67332667ffc00b31
t = 3 :	e308483153e15ad6 086c5b2d36a89178	eebe5d379be707ad 54074a65aef34336	2e91230306a12ae0 5e1b4e1695372b9e	470994ad30873f88 bd03f724be6075f9
t = 4 :	3a7a023c593d8479 8aa1144850633794	e308483153e15ad6 086c5b2d36a89178	eebe5d379be707ad 54074a65aef34336	2e91230306a12ae0 5e1b4e1695372b9e
t = 5 :	333199a85f92b052 7a6316f0ef047ce7	3a7a023c593d8479 8aa1144850633794	e308483153e15ad6 086c5b2d36a89178	eebe5d379be707ad 54074a65aef34336
t = 6 :	76f0741213dd2ef6 74063cba385f0675	333199a85f92b052 7a6316f0ef047ce7	3a7a023c593d8479 8aa1144850633794	e308483153e15ad6 086c5b2d36a89178
t = 7 :	02f2a04d3aab1629 1688b9bf14980fc0	76f0741213dd2ef6 74063cba385f0675	333199a85f92b052 7a6316f0ef047ce7	3a7a023c593d8479 8aa1144850633794
t = 8 :	73e5b2a1704a0349 fd00139f705907d0	02f2a04d3aab1629 1688b9bf14980fc0	76f0741213dd2ef6 74063cba385f0675	333199a85f92b052 7a6316f0ef047ce7
t = 9 :	bf3f67ba12882648 652e311d4f0a4257	73e5b2a1704a0349 fd00139f705907d0	02f2a04d3aab1629 1688b9bf14980fc0	76f0741213dd2ef6 74063cba385f0675
t = 10 :	33254508bb2ea48d 9e18991c4f39f0ba	bf3f67ba12882648 652e311d4f0a4257	73e5b2a1704a0349 fd00139f705907d0	02f2a04d3aab1629 1688b9bf14980fc0
t = 11 :	c1fdb2a0205ea0e5 04732e8bc4044582	33254508bb2ea48d 9e18991c4f39f0ba	bf3f67ba12882648 652e311d4f0a4257	73e5b2a1704a0349 fd00139f705907d0
t = 12 :	185f9ff038a50f39 8b4acfc4d2b8afe6	c1fdb2a0205ea0e5 04732e8bc4044582	33254508bb2ea48d 9e18991c4f39f0ba	bf3f67ba12882648 652e311d4f0a4257
t = 13 :	e5f06744c0d7563a 2fa93d1ce9523015	185f9ff038a50f39 8b4acfc4d2b8afe6	c1fdb2a0205ea0e5 04732e8bc4044582	33254508bb2ea48d 9e18991c4f39f0ba
t = 14 :	7e32dc0e9f414783 3a9950aaa5e75884	e5f06744c0d7563a 2fa93d1ce9523015	185f9ff038a50f39 8b4acfc4d2b8afe6	c1fdb2a0205ea0e5 04732e8bc4044582
t = 15 :	1eab6159ae87ef6d 153b895cfbc436c5	7e32dc0e9f414783 3a9950aaa5e75884	e5f06744c0d7563a 2fa93d1ce9523015	185f9ff038a50f39 8b4acfc4d2b8afe6
t = 16 :	33ef2cebbf1739aa 9d1a64baf1d366aa	1eab6159ae87ef6d 153b895cfbc436c5	7e32dc0e9f414783 3a9950aaa5e75884	e5f06744c0d7563a 2fa93d1ce9523015
t = 17 :	7df1b65f1b87d6ca 5b6e369d36e8e181	33ef2cebbf1739aa 9d1a64baf1d366aa	1eab6159ae87ef6d 153b895cfbc436c5	7e32dc0e9f414783 3a9950aaa5e75884
t = 18 :	63a24014a34bb0f6 e13e610eae680d85	7df1b65f1b87d6ca 5b6e369d36e8e181	33ef2cebbf1739aa 9d1a64baf1d366aa	1eab6159ae87ef6d 153b895cfbc436c5
t = 19 :	f1aabd313309509b 674385f0d87db94f	63a24014a34bb0f6 e13e610eae680d85	7df1b65f1b87d6ca 5b6e369d36e8e181	33ef2cebbf1739aa 9d1a64baf1d366aa

$t = 20$:	9ba737ae88a72c64	f1aabd313309509b	63a24014a34bb0f6	7df1b65f1b87d6ca
	3fc2614c43906c0f	674385f0d87db94f	e13e610eae680d85	5b6e369d36e8e181
$t = 21$:	042c2dc9a5bf558a	9ba737ae88a72c64	f1aabd313309509b	63a24014a34bb0f6
	19316bebc88e01f2	3fc2614c43906c0f	674385f0d87db94f	e13e610eae680d85
$t = 22$:	7799c75acc748c0f	042c2dc9a5bf558a	9ba737ae88a72c64	f1aabd313309509b
	a7bbd65bf64f58c8	19316bebc88e01f2	3fc2614c43906c0f	674385f0d87db94f
$t = 23$:	ccf99a80f92bf002	7799c75acc748c0f	042c2dc9a5bf558a	9ba737ae88a72c64
	e52a24fae4e8fc9b	a7bbd65bf64f58c8	19316bebc88e01f2	3fc2614c43906c0f
$t = 24$:	ae993474363efe68	ccf99a80f92bf002	7799c75acc748c0f	042c2dc9a5bf558a
	587f308d58681928	e52a24fae4e8fc9b	a7bbd65bf64f58c8	19316bebc88e01f2
$t = 25$:	335063d1a2aec92f	ae993474363efe68	ccf99a80f92bf002	7799c75acc748c0f
	c2d6d65e38c6ea79	587f308d58681928	e52a24fae4e8fc9b	a7bbd65bf64f58c8
$t = 26$:	53a78b0cca01ba37	335063d1a2aec92f	ae993474363efe68	ccf99a80f92bf002
	3b65a26c3c92c8f3	c2d6d65e38c6ea79	587f308d58681928	e52a24fae4e8fc9b
$t = 27$:	ab7ffa529f622930	53a78b0cca01ba37	335063d1a2aec92f	ae993474363efe68
	b9d8a2f2762901ea	3b65a26c3c92c8f3	c2d6d65e38c6ea79	587f308d58681928
$t = 28$:	e428bb43afe3d63e	ab7ffa529f622930	53a78b0cca01ba37	335063d1a2aec92f
	6a8527525f898726	b9d8a2f2762901ea	3b65a26c3c92c8f3	c2d6d65e38c6ea79
$t = 29$:	bbed541a5128088c	e428bb43afe3d63e	ab7ffa529f622930	53a78b0cca01ba37
	7973aadbde294be9	6a8527525f898726	b9d8a2f2762901ea	3b65a26c3c92c8f3
$t = 30$:	4c5c38df7ec8baf4	bbed541a5128088c	e428bb43afe3d63e	ab7ffa529f622930
	422ceea0200e9ee4	7973aadbde294be9	6a8527525f898726	b9d8a2f2762901ea
$t = 31$:	4ba456ec244033ed	4c5c38df7ec8baf4	bbed541a5128088c	e428bb43afe3d63e
	7cf40857056d86b0	422ceea0200e9ee4	7973aadbde294be9	6a8527525f898726
$t = 32$:	aa4a6ab2ac5f5dd8	4ba456ec244033ed	4c5c38df7ec8baf4	bbed541a5128088c
	ad2b1ecfb5bfc556	7cf40857056d86b0	422ceea0200e9ee4	7973aadbde294be9
$t = 33$:	9cb941f2ced774b3	aa4a6ab2ac5f5dd8	4ba456ec244033ed	4c5c38df7ec8baf4
	029f66c7b4569bf0	ad2b1ecfb5bfc556	7cf40857056d86b0	422ceea0200e9ee4
$t = 34$:	39265f358594de27	9cb941f2ced774b3	aa4a6ab2ac5f5dd8	4ba456ec244033ed
	3f7b1c260c82e54f	029f66c7b4569bf0	ad2b1ecfb5bfc556	7cf40857056d86b0
$t = 35$:	09cca487d39b02a1	39265f358594de27	9cb941f2ced774b3	aa4a6ab2ac5f5dd8
	4a22b37b58a5b1b0	3f7b1c260c82e54f	029f66c7b4569bf0	ad2b1ecfb5bfc556
$t = 36$:	d48d97ce438cf4f0	09cca487d39b02a1	39265f358594de27	9cb941f2ced774b3
	a239e00b8baa0410	4a22b37b58a5b1b0	3f7b1c260c82e54f	029f66c7b4569bf0
$t = 37$:	d6f41e25a8b634d6	d48d97ce438cf4f0	09cca487d39b02a1	39265f358594de27
	25755cb8179dd0b0	a239e00b8baa0410	4a22b37b58a5b1b0	3f7b1c260c82e54f
$t = 38$:	54078334358573b4	d6f41e25a8b634d6	d48d97ce438cf4f0	09cca487d39b02a1
	0e419fb0802b0efc	25755cb8179dd0b0	a239e00b8baa0410	4a22b37b58a5b1b0
$t = 39$:	db24f9a03f4fff6b	54078334358573b4	d6f41e25a8b634d6	d48d97ce438cf4f0
	d30e99b4b394b090	0e419fb0802b0efc	25755cb8179dd0b0	a239e00b8baa0410
$t = 40$:	3604c53a845efc37	db24f9a03f4fff6b	54078334358573b4	d6f41e25a8b634d6
	791b2b4af7338b99	d30e99b4b394b090	0e419fb0802b0efc	25755cb8179dd0b0
$t = 41$:	f41b1c0eee89bdc6	3604c53a845efc37	db24f9a03f4fff6b	54078334358573b4
	e319b77d9e4e87f9	791b2b4af7338b99	d30e99b4b394b090	0e419fb0802b0efc
$t = 42$:	36644ae374632e3a	f41b1c0eee89bdc6	3604c53a845efc37	db24f9a03f4fff6b
	458250878a3972b2	e319b77d9e4e87f9	791b2b4af7338b99	d30e99b4b394b090
$t = 43$:	88806f6ae9fcd65b	36644ae374632e3a	f41b1c0eee89bdc6	3604c53a845efc37

	cfde2e6ea54fa576	458250878a3972b2	e319b77d9e4e87f9	791b2b4af7338b99
$t = 44$:	51dcaa36995c301d	88806f6ae9fcd65b	36644ae374632e3a	f41b1c0eee89bdc6
	e37f778353998050	cfde2e6ea54fa576	458250878a3972b2	e319b77d9e4e87f9
$t = 45$:	ef5e3885a2f238df	51dcaa36995c301d	88806f6ae9fcd65b	36644ae374632e3a
	740e347f24e18fda	e37f778353998050	cfde2e6ea54fa576	458250878a3972b2
$t = 46$:	eb3753f4283f4818	ef5e3885a2f238df	51dcaa36995c301d	88806f6ae9fcd65b
	0ae48cf840bb8be9	740e347f24e18fda	e37f778353998050	cfde2e6ea54fa576
$t = 47$:	a6998d63a5d09e04	eb3753f4283f4818	ef5e3885a2f238df	51dcaa36995c301d
	e21095012ee0b72a	0ae48cf840bb8be9	740e347f24e18fda	e37f778353998050
$t = 48$:	d3698fb64df175b0	a6998d63a5d09e04	eb3753f4283f4818	ef5e3885a2f238df
	c2f0b90ffce80739	e21095012ee0b72a	0ae48cf840bb8be9	740e347f24e18fda
$t = 49$:	317a3b295b991914	d3698fb64df175b0	a6998d63a5d09e04	eb3753f4283f4818
	1cadff2e6cb5aa4d	c2f0b90ffce80739	e21095012ee0b72a	0ae48cf840bb8be9
$t = 50$:	0941da08148ba463	317a3b295b991914	d3698fb64df175b0	a6998d63a5d09e04
	833eb9a4bb5a073e	1cadff2e6cb5aa4d	c2f0b90ffce80739	e21095012ee0b72a
$t = 51$:	494ac238d68c3d0b	0941da08148ba463	317a3b295b991914	d3698fb64df175b0
	80c8fc138e645028	833eb9a4bb5a073e	1cadff2e6cb5aa4d	c2f0b90ffce80739
$t = 52$:	c87e9168db9e97de	494ac238d68c3d0b	0941da08148ba463	317a3b295b991914
	65cf7f6a829aca04	80c8fc138e645028	833eb9a4bb5a073e	1cadff2e6cb5aa4d
$t = 53$:	edb4448879391dbb	c87e9168db9e97de	494ac238d68c3d0b	0941da08148ba463
	7729c85475dd318f	65cf7f6a829aca04	80c8fc138e645028	833eb9a4bb5a073e
$t = 54$:	073775c2456dc7db	edb4448879391dbb	c87e9168db9e97de	494ac238d68c3d0b
	a9cca0b6266b1d77	7729c85475dd318f	65cf7f6a829aca04	80c8fc138e645028
$t = 55$:	54de8857b24afaf7	073775c2456dc7db	edb4448879391dbb	c87e9168db9e97de
	8de51cff2ae4b068	a9cca0b6266b1d77	7729c85475dd318f	65cf7f6a829aca04
$t = 56$:	8a9cdd80f7f09c05	54de8857b24afaf7	073775c2456dc7db	edb4448879391dbb
	a60ba5e9ebaeb96a	8de51cff2ae4b068	a9cca0b6266b1d77	7729c85475dd318f
$t = 57$:	3eeb22a7524d8d7f	8a9cdd80f7f09c05	54de8857b24afaf7	073775c2456dc7db
	e2e6830b139df58f	a60ba5e9ebaeb96a	8de51cff2ae4b068	a9cca0b6266b1d77
$t = 58$:	0ed77c9cde8883d3	3eeb22a7524d8d7f	8a9cdd80f7f09c05	54de8857b24afaf7
	38413a2052387a9e	e2e6830b139df58f	a60ba5e9ebaeb96a	8de51cff2ae4b068
$t = 59$:	e64e4135f9d30dbc	0ed77c9cde8883d3	3eeb22a7524d8d7f	8a9cdd80f7f09c05
	45b640454c75c349	38413a2052387a9e	e2e6830b139df58f	a60ba5e9ebaeb96a
$t = 60$:	1ca93a293d544328	e64e4135f9d30dbc	0ed77c9cde8883d3	3eeb22a7524d8d7f
	efbef83a35c0319e	45b640454c75c349	38413a2052387a9e	e2e6830b139df58f
$t = 61$:	3dc764f89e54043a	1ca93a293d544328	e64e4135f9d30dbc	0ed77c9cde8883d3
	a57784945550cf94	efbef83a35c0319e	45b640454c75c349	38413a2052387a9e
$t = 62$:	56fb5883f1c87a05	3dc764f89e54043a	1ca93a293d544328	e64e4135f9d30dbc
	f5198a41eb80e022	a57784945550cf94	efbef83a35c0319e	45b640454c75c349
$t = 63$:	24a1124262a331c7	56fb5883f1c87a05	3dc764f89e54043a	1ca93a293d544328
	06edacae6e7b54ad	f5198a41eb80e022	a57784945550cf94	efbef83a35c0319e
$t = 64$:	eb85d19201c89694	24a1124262a331c7	56fb5883f1c87a05	3dc764f89e54043a
	9ced24983eec8723	06edacae6e7b54ad	f5198a41eb80e022	a57784945550cf94
$t = 65$:	cc981ab3a59c1db4	eb85d19201c89694	24a1124262a331c7	56fb5883f1c87a05
	eac5516336bc8882	9ced24983eec8723	06edacae6e7b54ad	f5198a41eb80e022
$t = 66$:	ceef5d997e148b44	cc981ab3a59c1db4	eb85d19201c89694	24a1124262a331c7
	617bbf70bb165212	eac5516336bc8882	9ced24983eec8723	06edacae6e7b54ad

$t = 67$:	689edf608a8e3f14	ceef5d997e148b44	cc981ab3a59c1db4	eb85d19201c89694
	3280d88472c100fd	617bbf70bb165212	eac5516336bc8882	9ced24983eec8723
$t = 68$:	1e6e0255ab88079f	689edf608a8e3f14	ceef5d997e148b44	cc981ab3a59c1db4
	f2001138439902b1	3280d88472c100fd	617bbf70bb165212	eac5516336bc8882
$t = 69$:	8c5d3b7fdad66e70	1e6e0255ab88079f	689edf608a8e3f14	ceef5d997e148b44
	90d18ec8b69f0345	f2001138439902b1	3280d88472c100fd	617bbf70bb165212
$t = 70$:	32e5ed8655871e9b	8c5d3b7fdad66e70	1e6e0255ab88079f	689edf608a8e3f14
	51105f6241313777	90d18ec8b69f0345	f2001138439902b1	3280d88472c100fd
$t = 71$:	bcd5061679be7336	32e5ed8655871e9b	8c5d3b7fdad66e70	1e6e0255ab88079f
	454b99f654443ad0	51105f6241313777	90d18ec8b69f0345	f2001138439902b1
$t = 72$:	e7d913b6678e78ef	bcd5061679be7336	32e5ed8655871e9b	8c5d3b7fdad66e70
	1ff613b5aa63776e	454b99f654443ad0	51105f6241313777	90d18ec8b69f0345
$t = 73$:	e6b8cb8dfa3475ab	e7d913b6678e78ef	bcd5061679be7336	32e5ed8655871e9b
	2e75f34303d39bb0	1ff613b5aa63776e	454b99f654443ad0	51105f6241313777
$t = 74$:	fdd4a30e168c4ae5	e6b8cb8dfa3475ab	e7d913b6678e78ef	bcd5061679be7336
	83a35dbe2a64fc26	2e75f34303d39bb0	1ff613b5aa63776e	454b99f654443ad0
$t = 75$:	12aeb6268dfa3e14	fdd4a30e168c4ae5	e6b8cb8dfa3475ab	e7d913b6678e78ef
	f660943b276786f7	83a35dbe2a64fc26	2e75f34303d39bb0	1ff613b5aa63776e
$t = 76$:	055b73814cf102b4	12aeb6268dfa3e14	fdd4a30e168c4ae5	e6b8cb8dfa3475ab
	c4b149710f5d6a71	f660943b276786f7	83a35dbe2a64fc26	2e75f34303d39bb0
$t = 77$:	95d33150de6df44c	055b73814cf102b4	12aeb6268dfa3e14	fdd4a30e168c4ae5
	c7f7bff08ebf0d30	c4b149710f5d6a71	f660943b276786f7	83a35dbe2a64fc26
$t = 78$:	5306143f64497b00	95d33150de6df44c	055b73814cf102b4	12aeb6268dfa3e14
	ca06a219cc701096	c7f7bff08ebf0d30	c4b149710f5d6a71	f660943b276786f7
$t = 79$:	ff44d7e1849dbfb3	5306143f64497b00	95d33150de6df44c	055b73814cf102b4
	1952e0c3a227c0f2	ca06a219cc701096	c7f7bff08ebf0d30	c4b149710f5d6a71

That completes the processing of the first and only message block, $M^{(1)}$. The final hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{cbbb9d5dc1059ed8} + \texttt{ff44d7e1849dbfb3} = \texttt{cb00753f45a35e8b}$$
$$H_1^{(1)} = \texttt{629a292a367cd507} + \texttt{5306143f64497b00} = \texttt{b5a03d699ac65007}$$
$$H_2^{(1)} = \texttt{9159015a3070dd17} + \texttt{95d33150de6df44c} = \texttt{272c32ab0eded163}$$
$$H_3^{(1)} = \texttt{152fecd8f70e5939} + \texttt{055b73814cf102b4} = \texttt{1a8b605a43ff5bed}$$
$$H_4^{(1)} = \texttt{67332667ffc00b31} + \texttt{1952e0c3a227c0f2} = \texttt{8086072ba1e7cc23}$$
$$H_5^{(1)} = \texttt{8eb44a8768581511} + \texttt{ca06a219cc701096} = \texttt{58baeca134c825a7}$$
$$H_6^{(1)} = \texttt{db0c2e0d64f98fa7} + \texttt{c7f7bff08ebf0d30} = \texttt{a303edfdf3b89cd7}$$
$$H_7^{(1)} = \texttt{47b5481dbefa4fa4} + \texttt{c4b149710f5d6a71} = \texttt{0c66918ece57ba15.}$$

The final hash value is truncated to its left-most 384 bits (i.e., $H_0^{(1)}, \ldots, H_5^{(1)}$), resulting in the 384-bit message digest

```
cb00753f45a35e8b b5a03d699ac65007 272c32ab0eded163 1a8b605a43ff5bed
8086072ba1e7cc23 58baeca134c825a7.
```

D.2 SHA-384 Example (Multi-Block Message)

Let the message, *M*, be the 896-bit ($\ell = 896$) ASCII string

"**abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmn**
hijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu".

The message is padded by appending a "1" bit, followed by 1023 "0" bits, and ending with the hex value

0000000000000000 0000000000000380

(the two 64-bit word representation of the length, 24). Thus, the final padded message consists of two blocks ($N = 2$).

For SHA-384, the initial hash value, $H^{(0)}$, is

$H_0^{(0)}$ = cbbb9d5dc1059ed8
$H_1^{(0)}$ = 629a292a367cd507
$H_2^{(0)}$ = 9159015a3070dd17
$H_3^{(0)}$ = 152fecd8f70e5939
$H_4^{(0)}$ = 67332667ffc00b31
$H_5^{(0)}$ = 8eb44a8768581511
$H_6^{(0)}$ = db0c2e0d64f98fa7
$H_7^{(0)}$ = 47b5481dbefa4fa4.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0 = 6162636465666768
W_1 = 6263646566676869
W_2 = 636465666768696a
W_3 = 6465666768696a6b
W_4 = 65666768696a6b6c
W_5 = 666768696a6b6c6d
W_6 = 6768696a6b6c6d6e
W_7 = 68696a6b6c6d6e6f
W_8 = 696a6b6c6d6e6f70
W_9 = 6a6b6c6d6e6f7071
W_{10} = 6b6c6d6e6f707172
W_{11} = 6c6d6e6f70717273
W_{12} = 6d6e6f7071727374
W_{13} = 6e6f707172737475
W_{14} = 8000000000000000
W_{15} = 0000000000000000.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 79" loop described in Sec. 6.3.2, step 4.

	a / ***e***	***b*** / ***f***	***c*** / ***g***	***d*** / ***h***
$t = 0$:	4709949195eda6f0 bd03f70923c6dd61	cbbb9d5dc1059ed8 67332667ffc00b31	629a292a367cd507 8eb44a8768581511	9159015a3070dd17 db0c2e0d64f98fa7
$t = 1$:	78d3f8bc03a38303 ae067f071cd18a36	4709949195eda6f0 bd03f70923c6dd61	cbbb9d5dc1059ed8 67332667ffc00b31	629a292a367cd507 8eb44a8768581511
$t = 2$:	ed59d30beff95306 c180c7a74ed5cf1f	78d3f8bc03a38303 ae067f071cd18a36	4709949195eda6f0 bd03f70923c6dd61	cbbb9d5dc1059ed8 67332667ffc00b31
$t = 3$:	8e7fe2aba3168f2b d92d19667920b327	ed59d30beff95306 c180c7a74ed5cf1f	78d3f8bc03a38303 ae067f071cd18a36	4709949195eda6f0 bd03f70923c6dd61
$t = 4$:	1174f9b374a9263a dd371f2d13661c52	8e7fe2aba3168f2b d92d19667920b327	ed59d30beff95306 c180c7a74ed5cf1f	78d3f8bc03a38303 ae067f071cd18a36
$t = 5$:	27aaafb7fbef806b 21af3c6430a9af9c	1174f9b374a9263a dd371f2d13661c52	8e7fe2aba3168f2b d92d19667920b327	ed59d30beff95306 c180c7a74ed5cf1f
$t = 6$:	b352d03a0bd34d65 69397de9a30e1473	27aaafb7fbef806b 21af3c6430a9af9c	1174f9b374a9263a dd371f2d13661c52	8e7fe2aba3168f2b d92d19667920b327
$t = 7$:	412db7f990563d7c 5062fd5924e2b62e	b352d03a0bd34d65 69397de9a30e1473	27aaafb7fbef806b 21af3c6430a9af9c	1174f9b374a9263a dd371f2d13661c52
$t = 8$:	0f79040546e6edf7 6b6c511b25a6bdbc	412db7f990563d7c 5062fd5924e2b62e	b352d03a0bd34d65 69397de9a30e1473	27aaafb7fbef806b 21af3c6430a9af9c
$t = 9$:	ebf02410f67b8ee7 dac695b91543ae80	0f79040546e6edf7 6b6c511b25a6bdbc	412db7f990563d7c 5062fd5924e2b62e	b352d03a0bd34d65 69397de9a30e1473
$t = 10$:	97aa05d89b8dbe6d 83b8b72646c0b598	ebf02410f67b8ee7 dac695b91543ae80	0f79040546e6edf7 6b6c511b25a6bdbc	412db7f990563d7c 5062fd5924e2b62e
$t = 11$:	23d0a36b692118eb a5f6c5155e221e8c	97aa05d89b8dbe6d 83b8b72646c0b598	ebf02410f67b8ee7 dac695b91543ae80	0f79040546e6edf7 6b6c511b25a6bdbc
$t = 12$:	e1041368d2fca1a2 ae01675bfb003180	23d0a36b692118eb a5f6c5155e221e8c	97aa05d89b8dbe6d 83b8b72646c0b598	ebf02410f67b8ee7 dac695b91543ae80
$t = 13$:	45bd6f69efec540d c35cc50c1cf7ef98	e1041368d2fca1a2 ae01675bfb003180	23d0a36b692118eb a5f6c5155e221e8c	97aa05d89b8dbe6d 83b8b72646c0b598
$t = 14$:	c237fa23abb9bc16 a16c4f134b28923e	45bd6f69efec540d c35cc50c1cf7ef98	e1041368d2fca1a2 ae01675bfb003180	23d0a36b692118eb a5f6c5155e221e8c
$t = 15$:	b4092df1c0f81853 008178e17fa649f2	c237fa23abb9bc16 a16c4f134b28923e	45bd6f69efec540d c35cc50c1cf7ef98	e1041368d2fca1a2 ae01675bfb003180
$t = 16$:	21e5c91d11809c13 a26dfa04ed8c9b63	b4092df1c0f81853 008178e17fa649f2	c237fa23abb9bc16 a16c4f134b28923e	45bd6f69efec540d c35cc50c1cf7ef98
$t = 17$:	2c957137cd4304a5 6be210614b10949b	21e5c91d11809c13 a26dfa04ed8c9b63	b4092df1c0f81853 008178e17fa649f2	c237fa23abb9bc16 a16c4f134b28923e
$t = 18$:	2180e61afe322bc7 76396996200065f7	2c957137cd4304a5 6be210614b10949b	21e5c91d11809c13 a26dfa04ed8c9b63	b4092df1c0f81853 008178e17fa649f2
$t = 19$:	f2911c11c96e5ff5 1bc2160f4f3711dc	2180e61afe322bc7 76396996200065f7	2c957137cd4304a5 6be210614b10949b	21e5c91d11809c13 a26dfa04ed8c9b63
$t = 20$:	5eab10b19a5143a8 98d2b19d201f2bb6	f2911c11c96e5ff5 1bc2160f4f3711dc	2180e61afe322bc7 76396996200065f7	2c957137cd4304a5 6be210614b10949b
$t = 21$:	29c5348d87cd5590	5eab10b19a5143a8	f2911c11c96e5ff5	2180e61afe322bc7

	4324c8caccf7753c	98d2b19d201f2bb6	1bc2160f4f3711dc	76396996200065f7
$t = 22$:	33c6b4a0166b7c9c	29c5348d87cd5590	5eab10b19a5143a8	f2911c11c96e5ff5
	d49cef5bd2dec121	4324c8caccf7753c	98d2b19d201f2bb6	1bc2160f4f3711dc
$t = 23$:	1db4ee606d2a7a96	33c6b4a0166b7c9c	29c5348d87cd5590	5eab10b19a5143a8
	b17d15b397521ab3	d49cef5bd2dec121	4324c8caccf7753c	98d2b19d201f2bb6
$t = 24$:	5cef5b2f00142660	1db4ee606d2a7a96	33c6b4a0166b7c9c	29c5348d87cd5590
	789e540f22e13932	b17d15b397521ab3	d49cef5bd2dec121	4324c8caccf7753c
$t = 25$:	ff74f4a162435903	5cef5b2f00142660	1db4ee606d2a7a96	33c6b4a0166b7c9c
	6c0be33dcc6e7572	789e540f22e13932	b17d15b397521ab3	d49cef5bd2dec121
$t = 26$:	41740b736e9676a9	ff74f4a162435903	5cef5b2f00142660	1db4ee606d2a7a96
	d8e401251592da6c	6c0be33dcc6e7572	789e540f22e13932	b17d15b397521ab3
$t = 27$:	931059fe9279ff1d	41740b736e9676a9	ff74f4a162435903	5cef5b2f00142660
	7f31116887eea596	d8e401251592da6c	6c0be33dcc6e7572	789e540f22e13932
$t = 28$:	356d08d982e2ead4	931059fe9279ff1d	41740b736e9676a9	ff74f4a162435903
	40c28c34b1bbe906	7f31116887eea596	d8e401251592da6c	6c0be33dcc6e7572
$t = 29$:	89dc825e7235c74b	356d08d982e2ead4	931059fe9279ff1d	41740b736e9676a9
	7a499ae05da50bf2	40c28c34b1bbe906	7f31116887eea596	d8e401251592da6c
$t = 30$:	97901f333e662fdc	89dc825e7235c74b	356d08d982e2ead4	931059fe9279ff1d
	4472b2e331ddfab4	7a499ae05da50bf2	40c28c34b1bbe906	7f31116887eea596
$t = 31$:	69c8f40eb38b6022	97901f333e662fdc	89dc825e7235c74b	356d08d982e2ead4
	177589502dd39aa2	4472b2e331ddfab4	7a499ae05da50bf2	40c28c34b1bbe906
$t = 32$:	4920943ffe52b207	69c8f40eb38b6022	97901f333e662fdc	89dc825e7235c74b
	6b813a0d0cdf4991	177589502dd39aa2	4472b2e331ddfab4	7a499ae05da50bf2
$t = 33$:	b4cb0df332d108ab	4920943ffe52b207	69c8f40eb38b6022	97901f333e662fdc
	8fe3d28097f18618	6b813a0d0cdf4991	177589502dd39aa2	4472b2e331ddfab4
$t = 34$:	e7748fbf744a5240	b4cb0df332d108ab	4920943ffe52b207	69c8f40eb38b6022
	0d7ab03208f1d7a5	8fe3d28097f18618	6b813a0d0cdf4991	177589502dd39aa2
$t = 35$:	7416ca18d9e265e0	e7748fbf744a5240	b4cb0df332d108ab	4920943ffe52b207
	11200c2d47c082f8	0d7ab03208f1d7a5	8fe3d28097f18618	6b813a0d0cdf4991
$t = 36$:	75476f5456e82f9c	7416ca18d9e265e0	e7748fbf744a5240	b4cb0df332d108ab
	3024702447f76224	11200c2d47c082f8	0d7ab03208f1d7a5	8fe3d28097f18618
$t = 37$:	f638a568b53a2f8f	75476f5456e82f9c	7416ca18d9e265e0	e7748fbf744a5240
	6217c1c02153302c	3024702447f76224	11200c2d47c082f8	0d7ab03208f1d7a5
$t = 38$:	c418f6f90602c79a	f638a568b53a2f8f	75476f5456e82f9c	7416ca18d9e265e0
	87f0901c227adbb3	6217c1c02153302c	3024702447f76224	11200c2d47c082f8
$t = 39$:	4f1f4f21df3dcf43	c418f6f90602c79a	f638a568b53a2f8f	75476f5456e82f9c
	fb7c63fcddf4a1c2	87f0901c227adbb3	6217c1c02153302c	3024702447f76224
$t = 40$:	13eb82e4b98d0e67	4f1f4f21df3dcf43	c418f6f90602c79a	f638a568b53a2f8f
	fb6c0e54d48d4f2d	fb7c63fcddf4a1c2	87f0901c227adbb3	6217c1c02153302c
$t = 41$:	820e75046567bace	13eb82e4b98d0e67	4f1f4f21df3dcf43	c418f6f90602c79a
	b16a9397472f0123	fb6c0e54d48d4f2d	fb7c63fcddf4a1c2	87f0901c227adbb3
$t = 42$:	741fa5dc290dd02c	820e75046567bace	13eb82e4b98d0e67	4f1f4f21df3dcf43
	ed40c88214823792	b16a9397472f0123	fb6c0e54d48d4f2d	fb7c63fcddf4a1c2
$t = 43$:	a4809bf6da6aa8bd	741fa5dc290dd02c	820e75046567bace	13eb82e4b98d0e67
	bec3d7e88c855194	ed40c88214823792	b16a9397472f0123	fb6c0e54d48d4f2d
$t = 44$:	d70b1aa4c800979c	a4809bf6da6aa8bd	741fa5dc290dd02c	820e75046567bace

	4962f310bdbd54b0	bec3d7e88c855194	ed40c88214823792	b16a9397472f0123
$t = 45$:	9a195492cfdb4745	d70b1aa4c800979c	a4809bf6da6aa8bd	741fa5dc290dd02c
	2c82d09cf05cf687	4962f310bdbd54b0	bec3d7e88c855194	ed40c88214823792
$t = 46$:	b7e68364f07f017e	9a195492cfdb4745	d70b1aa4c800979c	a4809bf6da6aa8bd
	2a1ffb84031b1b6c	2c82d09cf05cf687	4962f310bdbd54b0	bec3d7e88c855194
$t = 47$:	0e574b8e0b35e452	b7e68364f07f017e	9a195492cfdb4745	d70b1aa4c800979c
	29bdab29ee472a23	2a1ffb84031b1b6c	2c82d09cf05cf687	4962f310bdbd54b0
$t = 48$:	c176009cf82fa842	0e574b8e0b35e452	b7e68364f07f017e	9a195492cfdb4745
	cca47fbe31b335f4	29bdab29ee472a23	2a1ffb84031b1b6c	2c82d09cf05cf687
$t = 49$:	5d4f78c7a9bdbed2	c176009cf82fa842	0e574b8e0b35e452	b7e68364f07f017e
	eaf198615e99ffdc	cca47fbe31b335f4	29bdab29ee472a23	2a1ffb84031b1b6c
$t = 50$:	51ab3be828d8d13c	5d4f78c7a9bdbed2	c176009cf82fa842	0e574b8e0b35e452
	bd527cd188fb59ae	eaf198615e99ffdc	cca47fbe31b335f4	29bdab29ee472a23
$t = 51$:	4d639ef80d0f6d3e	51ab3be828d8d13c	5d4f78c7a9bdbed2	c176009cf82fa842
	b2611b90f90d732f	bd527cd188fb59ae	eaf198615e99ffdc	cca47fbe31b335f4
$t = 52$:	bba9c9efe0fbc6c8	4d639ef80d0f6d3e	51ab3be828d8d13c	5d4f78c7a9bdbed2
	fc0579337591a2c9	b2611b90f90d732f	bd527cd188fb59ae	eaf198615e99ffdc
$t = 53$:	3405d7cad2e8a689	bba9c9efe0fbc6c8	4d639ef80d0f6d3e	51ab3be828d8d13c
	0f6649f64ec8e109	fc0579337591a2c9	b2611b90f90d732f	bd527cd188fb59ae
$t = 54$:	ea54d908505798b3	3405d7cad2e8a689	bba9c9efe0fbc6c8	4d639ef80d0f6d3e
	ef48a48999108077	0f6649f64ec8e109	fc0579337591a2c9	b2611b90f90d732f
$t = 55$:	be31d1c0ccc143bc	ea54d908505798b3	3405d7cad2e8a689	bba9c9efe0fbc6c8
	4fc2d4cad0c91afc	ef48a48999108077	0f6649f64ec8e109	fc0579337591a2c9
$t = 56$:	285a76d23f6a0073	be31d1c0ccc143bc	ea54d908505798b3	3405d7cad2e8a689
	a730855599b738a3	4fc2d4cad0c91afc	ef48a48999108077	0f6649f64ec8e109
$t = 57$:	a714ceff14bebc24	285a76d23f6a0073	be31d1c0ccc143bc	ea54d908505798b3
	53c581dae1831d80	a730855599b738a3	4fc2d4cad0c91afc	ef48a48999108077
$t = 58$:	697ca14913a50a26	a714ceff14bebc24	285a76d23f6a0073	be31d1c0ccc143bc
	34d39344354aacd2	53c581dae1831d80	a730855599b738a3	4fc2d4cad0c91afc
$t = 59$:	3a38fa3775d7007c	697ca14913a50a26	a714ceff14bebc24	285a76d23f6a0073
	e26f3a21e9a27691	34d39344354aacd2	53c581dae1831d80	a730855599b738a3
$t = 60$:	44ea14d8e450c844	3a38fa3775d7007c	697ca14913a50a26	a714ceff14bebc24
	5319374fb88dd485	e26f3a21e9a27691	34d39344354aacd2	53c581dae1831d80
$t = 61$:	0928b75c925f91e2	44ea14d8e450c844	3a38fa3775d7007c	697ca14913a50a26
	79f4be3c5a372911	5319374fb88dd485	e26f3a21e9a27691	34d39344354aacd2
$t = 62$:	6db5469fa19c0e27	0928b75c925f91e2	44ea14d8e450c844	3a38fa3775d7007c
	16beec0fec168e79	79f4be3c5a372911	5319374fb88dd485	e26f3a21e9a27691
$t = 63$:	384e3159898a7362	6db5469fa19c0e27	0928b75c925f91e2	44ea14d8e450c844
	55fa3ad1102298a8	16beec0fec168e79	79f4be3c5a372911	5319374fb88dd485
$t = 64$:	483c64d3fdebf828	384e3159898a7362	6db5469fa19c0e27	0928b75c925f91e2
	1a238431921ea75e	55fa3ad1102298a8	16beec0fec168e79	79f4be3c5a372911
$t = 65$:	c9464988a1939bcf	483c64d3fdebf828	384e3159898a7362	6db5469fa19c0e27
	e3f3f08ac90f86cd	1a238431921ea75e	55fa3ad1102298a8	16beec0fec168e79
$t = 66$:	98bc93bca795059c	c9464988a1939bcf	483c64d3fdebf828	384e3159898a7362
	9e04fb49a5fd91de	e3f3f08ac90f86cd	1a238431921ea75e	55fa3ad1102298a8
$t = 67$:	b6fc101ad1d74e20	98bc93bca795059c	c9464988a1939bcf	483c64d3fdebf828
	fd13cd3620f6c1f4	9e04fb49a5fd91de	e3f3f08ac90f86cd	1a238431921ea75e

$t = 68$:	fac26e6e4da4705d	b6fc101ad1d74e20	98bc93bca795059c	c9464988a1939bcf
	0d60228aa6e55b6e	fd13cd3620f6c1f4	9e04fb49a5fd91de	e3f3f08ac90f86cd
$t = 69$:	2a630c58cc27fcaa	fac26e6e4da4705d	b6fc101ad1d74e20	98bc93bca795059c
	a2f7f27a3ec25aba	0d60228aa6e55b6e	fd13cd3620f6c1f4	9e04fb49a5fd91de
$t = 70$:	159a02d4faee11b4	2a630c58cc27fcaa	fac26e6e4da4705d	b6fc101ad1d74e20
	b2860fc55bdedaa6	a2f7f27a3ec25aba	0d60228aa6e55b6e	fd13cd3620f6c1f4
$t = 71$:	9d38bdb9df22b557	159a02d4faee11b4	2a630c58cc27fcaa	fac26e6e4da4705d
	dfc37c68af65f8bc	b2860fc55bdedaa6	a2f7f27a3ec25aba	0d60228aa6e55b6e
$t = 72$:	d42c3a57cfa78513	9d38bdb9df22b557	159a02d4faee11b4	2a630c58cc27fcaa
	bb56dea6a325ba32	dfc37c68af65f8bc	b2860fc55bdedaa6	a2f7f27a3ec25aba
$t = 73$:	abab4b0ca75a17c7	d42c3a57cfa78513	9d38bdb9df22b557	159a02d4faee11b4
	9ac71d1c037a8bbd	bb56dea6a325ba32	dfc37c68af65f8bc	b2860fc55bdedaa6
$t = 74$:	500f7b61186f6c2e	abab4b0ca75a17c7	d42c3a57cfa78513	9d38bdb9df22b557
	8347f5736531b3ec	9ac71d1c037a8bbd	bb56dea6a325ba32	dfc37c68af65f8bc
$t = 75$:	4abe0af6a67db2fe	500f7b61186f6c2e	abab4b0ca75a17c7	d42c3a57cfa78513
	14e986342ddced0f	8347f5736531b3ec	9ac71d1c037a8bbd	bb56dea6a325ba32
$t = 76$:	e1053fc85f9e56be	4abe0af6a67db2fe	500f7b61186f6c2e	abab4b0ca75a17c7
	4779767cc2ec5321	14e986342ddced0f	8347f5736531b3ec	9ac71d1c037a8bbd
$t = 77$:	7001201948fb3d71	e1053fc85f9e56be	4abe0af6a67db2fe	500f7b61186f6c2e
	5cdf6c58fc052572	4779767cc2ec5321	14e986342ddced0f	8347f5736531b3ec
$t = 78$:	88146da76ff6f23a	7001201948fb3d71	e1053fc85f9e56be	4abe0af6a67db2fe
	8901cffe7a74db98	5cdf6c58fc052572	4779767cc2ec5321	14e986342ddced0f
$t = 79$:	5ec3802b9ecfef33	88146da76ff6f23a	7001201948fb3d71	e1053fc85f9e56be
	5f2eead69efb4233	8901cffe7a74db98	5cdf6c58fc052572	4779767cc2ec5321

That completes the processing of the first message block, $M^{(1)}$. The intermediate hash value, $H^{(1)}$, is calculated to be

$$H_0^{(1)} = \texttt{cbbb9d5dc1059ed8} + \texttt{5ec3802b9ecfef33} = \texttt{2a7f1d895fd58e0b}$$
$$H_1^{(1)} = \texttt{629a292a367cd507} + \texttt{88146da76ff6f23a} = \texttt{eaae96d1a673c741}$$
$$H_2^{(1)} = \texttt{9159015a3070dd17} + \texttt{7001201948fb3d71} = \texttt{015a2173796c1a88}$$
$$H_3^{(1)} = \texttt{152fecd8f70e5939} + \texttt{e1053fc85f9e56be} = \texttt{f6352ca156acaff7}$$
$$H_4^{(1)} = \texttt{67332667ffc00b31} + \texttt{5f2eead69efb4233} = \texttt{c662113e9ebb4d64}$$
$$H_5^{(1)} = \texttt{8eb44a8768581511} + \texttt{8901cffe7a74db98} = \texttt{17b61a85e2ccf0a9}$$
$$H_6^{(1)} = \texttt{db0c2e0d64f98fa7} + \texttt{5cdf6c58fc052572} = \texttt{37eb9a6660feb519}$$
$$H_7^{(1)} = \texttt{47b5481dbefa4fa4} + \texttt{4779767cc2ec5321} = \texttt{8f2ebe9a81e6a2c5}.$$

The words of the *second* padded message block, $M^{(2)}$, are then assigned to the words $W_0,\ldots,W_{15}$ of the message schedule:

W_0	= 0000000000000000	W_8	= 0000000000000000
W_1	= 0000000000000000	W_9	= 0000000000000000
W_2	= 0000000000000000	W_{10}	= 0000000000000000
W_3	= 0000000000000000	W_{11}	= 0000000000000000
W_4	= 0000000000000000	W_{12}	= 0000000000000000
W_5	= 0000000000000000	W_{13}	= 0000000000000000
W_6	= 0000000000000000	W_{14}	= 0000000000000000
W_7	= 0000000000000000	W_{15}	= 0000000000000380.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h*** *after* pass *t* of the "for *t* = 0 to 79" loop described in Sec. 6.3.2, step 4.

	a / ***e***	***b*** / ***f***	***c*** / ***g***	***d*** / ***h***
t = 0 :	657a3c2ca9639d40 791f2ad0055fdd62	2a7f1d895fd58e0b c662113e9ebb4d64	eaae96d1a673c741 17b61a85e2ccf0a9	015a2173796c1a88 37eb9a6660feb519
t = 1 :	2a4ad5d9b9fd6d86 dbf2e656b5be3f14	657a3c2ca9639d40 791f2ad0055fdd62	2a7f1d895fd58e0b c662113e9ebb4d64	eaae96d1a673c741 17b61a85e2ccf0a9
t = 2 :	f0aa6758653d1664 6e0466c82f4fd35d	2a4ad5d9b9fd6d86 dbf2e656b5be3f14	657a3c2ca9639d40 791f2ad0055fdd62	2a7f1d895fd58e0b c662113e9ebb4d64
t = 3 :	43a76f011a73d317 1367bd36d15e8b40	f0aa6758653d1664 6e0466c82f4fd35d	2a4ad5d9b9fd6d86 dbf2e656b5be3f14	657a3c2ca9639d40 791f2ad0055fdd62
t = 4 :	d802c2dfd7cc48f6 f73d759b839a2a21	43a76f011a73d317 1367bd36d15e8b40	f0aa6758653d1664 6e0466c82f4fd35d	2a4ad5d9b9fd6d86 dbf2e656b5be3f14
t = 5 :	481208e5e8314602 6b2271a46f14c843	d802c2dfd7cc48f6 f73d759b839a2a21	43a76f011a73d317 1367bd36d15e8b40	f0aa6758653d1664 6e0466c82f4fd35d
t = 6 :	af9f8112df35cf33 257f4a7d524d7b0b	481208e5e8314602 6b2271a46f14c843	d802c2dfd7cc48f6 f73d759b839a2a21	43a76f011a73d317 1367bd36d15e8b40
t = 7 :	6730781342d1131b 81957ad408cec995	af9f8112df35cf33 257f4a7d524d7b0b	481208e5e8314602 6b2271a46f14c843	d802c2dfd7cc48f6 f73d759b839a2a21
t = 8 :	82e64c677356a82e 10b62fdce4ebaa51	6730781342d1131b 81957ad408cec995	af9f8112df35cf33 257f4a7d524d7b0b	481208e5e8314602 6b2271a46f14c843
t = 9 :	203578820a8f27d0 9937b3a0cb9248a1	82e64c677356a82e 10b62fdce4ebaa51	6730781342d1131b 81957ad408cec995	af9f8112df35cf33 257f4a7d524d7b0b
t = 10 :	0bac2a84c29a1e2b 6ad288dab3de0d53	203578820a8f27d0 9937b3a0cb9248a1	82e64c677356a82e 10b62fdce4ebaa51	6730781342d1131b 81957ad408cec995
t = 11 :	dd3ff8a140485c25 3149b728123c465e	0bac2a84c29a1e2b 6ad288dab3de0d53	203578820a8f27d0 9937b3a0cb9248a1	82e64c677356a82e 10b62fdce4ebaa51
t = 12 :	e826239f830c5346 4bb7b199c4ced186	dd3ff8a140485c25 3149b728123c465e	0bac2a84c29a1e2b 6ad288dab3de0d53	203578820a8f27d0 9937b3a0cb9248a1
t = 13 :	32215ce49aae40f8 9a2872c72d790d49	e826239f830c5346 4bb7b199c4ced186	dd3ff8a140485c25 3149b728123c465e	0bac2a84c29a1e2b 6ad288dab3de0d53
t = 14 :	859533bac457f94e 539f225d25ebeb4c	32215ce49aae40f8 9a2872c72d790d49	e826239f830c5346 4bb7b199c4ced186	dd3ff8a140485c25 3149b728123c465e
t = 15 :	a88704d9962849f3	859533bac457f94e	32215ce49aae40f8	e826239f830c5346

	63bf0472ef24f7a5	539f225d25ebeb4c	9a2872c72d790d49	4bb7b199c4ced186
$t = 16$:	3aa5c566a6cfad1c	a88704d9962849f3	859533bac457f94e	32215ce49aae40f8
	ce23f6380ead33c2	63bf0472ef24f7a5	539f225d25ebeb4c	9a2872c72d790d49
$t = 17$:	2e9c483a7c08c9c1	3aa5c566a6cfad1c	a88704d9962849f3	859533bac457f94e
	b033f945f3e6b4a2	ce23f6380ead33c2	63bf0472ef24f7a5	539f225d25ebeb4c
$t = 18$:	5a68585ae0835231	2e9c483a7c08c9c1	3aa5c566a6cfad1c	a88704d9962849f3
	8a0187a9ce93d875	b033f945f3e6b4a2	ce23f6380ead33c2	63bf0472ef24f7a5
$t = 19$:	cf9cd481e6407ced	5a68585ae0835231	2e9c483a7c08c9c1	3aa5c566a6cfad1c
	37a29fa30531bac7	8a0187a9ce93d875	b033f945f3e6b4a2	ce23f6380ead33c2
$t = 20$:	3f463f864f6474d9	cf9cd481e6407ced	5a68585ae0835231	2e9c483a7c08c9c1
	0cf45bb3c07e847d	37a29fa30531bac7	8a0187a9ce93d875	b033f945f3e6b4a2
$t = 21$:	cea26288dff931a5	3f463f864f6474d9	cf9cd481e6407ced	5a68585ae0835231
	34f1b5f46bf48a73	0cf45bb3c07e847d	37a29fa30531bac7	8a0187a9ce93d875
$t = 22$:	89634cd0f4f6c08a	cea26288dff931a5	3f463f864f6474d9	cf9cd481e6407ced
	3a728a543405a8e4	34f1b5f46bf48a73	0cf45bb3c07e847d	37a29fa30531bac7
$t = 23$:	625fa38464e5c880	89634cd0f4f6c08a	cea26288dff931a5	3f463f864f6474d9
	cee1b47a49b2fc42	3a728a543405a8e4	34f1b5f46bf48a73	0cf45bb3c07e847d
$t = 24$:	7dd21453a15a3b92	625fa38464e5c880	89634cd0f4f6c08a	cea26288dff931a5
	9308bfa1be1f800b	cee1b47a49b2fc42	3a728a543405a8e4	34f1b5f46bf48a73
$t = 25$:	3d76277bc8cb0601	7dd21453a15a3b92	625fa38464e5c880	89634cd0f4f6c08a
	480e017f5d1f0b1e	9308bfa1be1f800b	cee1b47a49b2fc42	3a728a543405a8e4
$t = 26$:	c8d904196f5a1f54	3d76277bc8cb0601	7dd21453a15a3b92	625fa38464e5c880
	4bd2f1f6e940c332	480e017f5d1f0b1e	9308bfa1be1f800b	cee1b47a49b2fc42
$t = 27$:	b033139b58b6e423	c8d904196f5a1f54	3d76277bc8cb0601	7dd21453a15a3b92
	f816ec1cbe0adafb	4bd2f1f6e940c332	480e017f5d1f0b1e	9308bfa1be1f800b
$t = 28$:	097768182cb65f57	b033139b58b6e423	c8d904196f5a1f54	3d76277bc8cb0601
	62e3de54dcd8f974	f816ec1cbe0adafb	4bd2f1f6e940c332	480e017f5d1f0b1e
$t = 29$:	3196649ab5f5cc39	097768182cb65f57	b033139b58b6e423	c8d904196f5a1f54
	f6887de116d0bd8f	62e3de54dcd8f974	f816ec1cbe0adafb	4bd2f1f6e940c332
$t = 30$:	f78d3d221d16965f	3196649ab5f5cc39	097768182cb65f57	b033139b58b6e423
	c7e4859c2858ed3c	f6887de116d0bd8f	62e3de54dcd8f974	f816ec1cbe0adafb
$t = 31$:	f58e9876b4984b51	f78d3d221d16965f	3196649ab5f5cc39	097768182cb65f57
	621352b394b8ca02	c7e4859c2858ed3c	f6887de116d0bd8f	62e3de54dcd8f974
$t = 32$:	38fbf0e726e04f78	f58e9876b4984b51	f78d3d221d16965f	3196649ab5f5cc39
	4319856f17a0a430	621352b394b8ca02	c7e4859c2858ed3c	f6887de116d0bd8f
$t = 33$:	f4be0b32a57597a2	38fbf0e726e04f78	f58e9876b4984b51	f78d3d221d16965f
	c6d392a3b4eb0ed8	4319856f17a0a430	621352b394b8ca02	c7e4859c2858ed3c
$t = 34$:	f8a6b3fe2e4f0634	f4be0b32a57597a2	38fbf0e726e04f78	f58e9876b4984b51
	602663c0f34eff33	c6d392a3b4eb0ed8	4319856f17a0a430	621352b394b8ca02
$t = 35$:	9bc3871be8046113	f8a6b3fe2e4f0634	f4be0b32a57597a2	38fbf0e726e04f78
	05542ecd9883c6ba	602663c0f34eff33	c6d392a3b4eb0ed8	4319856f17a0a430
$t = 36$:	f1bd2d46be619585	9bc3871be8046113	f8a6b3fe2e4f0634	f4be0b32a57597a2
	e47b9933bafdc655	05542ecd9883c6ba	602663c0f34eff33	c6d392a3b4eb0ed8
$t = 37$:	24c84b58d119affe	f1bd2d46be619585	9bc3871be8046113	f8a6b3fe2e4f0634
	5ae0b1175beb5d2b	e47b9933bafdc655	05542ecd9883c6ba	602663c0f34eff33
$t = 38$:	ec6d3abc2b291fd3	24c84b58d119affe	f1bd2d46be619585	9bc3871be8046113

	9ecc381d277748a3	5ae0b1175beb5d2b	e47b9933bafdc655	05542ecd9883c6ba
$t = 39$:	e266c1f77d5ee90e	ec6d3abc2b291fd3	24c84b58d119affe	f1bd2d46be619585
	d92f34c110296b32	9ecc381d277748a3	5ae0b1175beb5d2b	e47b9933bafdc655
$t = 40$:	5adbaa463642b570	e266c1f77d5ee90e	ec6d3abc2b291fd3	24c84b58d119affe
	83e8f410f859388e	d92f34c110296b32	9ecc381d277748a3	5ae0b1175beb5d2b
$t = 41$:	50fdb7bb2e499a34	5adbaa463642b570	e266c1f77d5ee90e	ec6d3abc2b291fd3
	257ed8ea645e933a	83e8f410f859388e	d92f34c110296b32	9ecc381d277748a3
$t = 42$:	06514212bb7fa152	50fdb7bb2e499a34	5adbaa463642b570	e266c1f77d5ee90e
	466781db35181abe	257ed8ea645e933a	83e8f410f859388e	d92f34c110296b32
$t = 43$:	673ed5a55ff2b07d	06514212bb7fa152	50fdb7bb2e499a34	5adbaa463642b570
	ba78f3545e7914f0	466781db35181abe	257ed8ea645e933a	83e8f410f859388e
$t = 44$:	125e2e5118393e2b	673ed5a55ff2b07d	06514212bb7fa152	50fdb7bb2e499a34
	4453b23a3e13b090	ba78f3545e7914f0	466781db35181abe	257ed8ea645e933a
$t = 45$:	07ee813df5910cec	125e2e5118393e2b	673ed5a55ff2b07d	06514212bb7fa152
	eae013a0510d23cc	4453b23a3e13b090	ba78f3545e7914f0	466781db35181abe
$t = 46$:	0a0508f0a1d719c3	07ee813df5910cec	125e2e5118393e2b	673ed5a55ff2b07d
	a93815eb58891016	eae013a0510d23cc	4453b23a3e13b090	ba78f3545e7914f0
$t = 47$:	0fc8f3b3efcb1b96	0a0508f0a1d719c3	07ee813df5910cec	125e2e5118393e2b
	a071cc73b966e801	a93815eb58891016	eae013a0510d23cc	4453b23a3e13b090
$t = 48$:	02aa5b28199f304a	0fc8f3b3efcb1b96	0a0508f0a1d719c3	07ee813df5910cec
	a49f1e14f8a2be7a	a071cc73b966e801	a93815eb58891016	eae013a0510d23cc
$t = 49$:	9223e1b34382f104	02aa5b28199f304a	0fc8f3b3efcb1b96	0a0508f0a1d719c3
	bfe2106e512a7331	a49f1e14f8a2be7a	a071cc73b966e801	a93815eb58891016
$t = 50$:	e01a1e47ee8d5656	9223e1b34382f104	02aa5b28199f304a	0fc8f3b3efcb1b96
	592b899b35469a78	bfe2106e512a7331	a49f1e14f8a2be7a	a071cc73b966e801
$t = 51$:	fa7b17aad857c2f4	e01a1e47ee8d5656	9223e1b34382f104	02aa5b28199f304a
	eb6e85e4682c1671	592b899b35469a78	bfe2106e512a7331	a49f1e14f8a2be7a
$t = 52$:	0c523b7a3c84ab77	fa7b17aad857c2f4	e01a1e47ee8d5656	9223e1b34382f104
	b5e80e871ac0c005	eb6e85e4682c1671	592b899b35469a78	bfe2106e512a7331
$t = 53$:	c773d8b69da1fde2	0c523b7a3c84ab77	fa7b17aad857c2f4	e01a1e47ee8d5656
	be2b0602fc6f8f65	b5e80e871ac0c005	eb6e85e4682c1671	592b899b35469a78
$t = 54$:	c6b1bc79a4f23679	c773d8b69da1fde2	0c523b7a3c84ab77	fa7b17aad857c2f4
	c80bdc57f38a05e4	be2b0602fc6f8f65	b5e80e871ac0c005	eb6e85e4682c1671
$t = 55$:	bef9bb0fe467fd60	c6b1bc79a4f23679	c773d8b69da1fde2	0c523b7a3c84ab77
	1dab0bd116e434e5	c80bdc57f38a05e4	be2b0602fc6f8f65	b5e80e871ac0c005
$t = 56$:	8e3db3e380ec7f22	bef9bb0fe467fd60	c6b1bc79a4f23679	c773d8b69da1fde2
	32ef50751734ffee	1dab0bd116e434e5	c80bdc57f38a05e4	be2b0602fc6f8f65
$t = 57$:	1003ec42412c7b7d	8e3db3e380ec7f22	bef9bb0fe467fd60	c6b1bc79a4f23679
	1ec0d46f349fd058	32ef50751734ffee	1dab0bd116e434e5	c80bdc57f38a05e4
$t = 58$:	375facc76291f85e	1003ec42412c7b7d	8e3db3e380ec7f22	bef9bb0fe467fd60
	59c8bc0488f9768b	1ec0d46f349fd058	32ef50751734ffee	1dab0bd116e434e5
$t = 59$:	bd113d92e0354fb9	375facc76291f85e	1003ec42412c7b7d	8e3db3e380ec7f22
	e66c73db3fad397d	59c8bc0488f9768b	1ec0d46f349fd058	32ef50751734ffee
$t = 60$:	2f61d4fd8e36d9d4	bd113d92e0354fb9	375facc76291f85e	1003ec42412c7b7d
	e9f21933e1c02948	e66c73db3fad397d	59c8bc0488f9768b	1ec0d46f349fd058
$t = 61$:	1b1ad88b92701ae2	2f61d4fd8e36d9d4	bd113d92e0354fb9	375facc76291f85e
	6fd0c1719bcac335	e9f21933e1c02948	e66c73db3fad397d	59c8bc0488f9768b

$t = 62$:	93d09fc06a19c5da	1b1ad88b92701ae2	2f61d4fd8e36d9d4	bd113d92e0354fb9
	b765273f571a571e	6fd0c1719bcac335	e9f21933e1c02948	e66c73db3fad397d
$t = 63$:	04bea2ce99cc3bf6	93d09fc06a19c5da	1b1ad88b92701ae2	2f61d4fd8e36d9d4
	6ab0e443c2f63714	b765273f571a571e	6fd0c1719bcac335	e9f21933e1c02948
$t = 64$:	02ebfc0a13492f52	04bea2ce99cc3bf6	93d09fc06a19c5da	1b1ad88b92701ae2
	77300c52e05af415	6ab0e443c2f63714	b765273f571a571e	6fd0c1719bcac335
$t = 65$:	1bf525abce8d6f04	02ebfc0a13492f52	04bea2ce99cc3bf6	93d09fc06a19c5da
	8faf12c33bb371b9	77300c52e05af415	6ab0e443c2f63714	b765273f571a571e
$t = 66$:	b6a36a3431547328	1bf525abce8d6f04	02ebfc0a13492f52	04bea2ce99cc3bf6
	fa8bb40b4e08100f	8faf12c33bb371b9	77300c52e05af415	6ab0e443c2f63714
$t = 67$:	ffdaf83202af0d72	b6a36a3431547328	1bf525abce8d6f04	02ebfc0a13492f52
	8045a82f723a9b4e	fa8bb40b4e08100f	8faf12c33bb371b9	77300c52e05af415
$t = 68$:	12737373d2985232	ffdaf83202af0d72	b6a36a3431547328	1bf525abce8d6f04
	870dbce23bad8988	8045a82f723a9b4e	fa8bb40b4e08100f	8faf12c33bb371b9
$t = 69$:	6189f68162b256b5	12737373d2985232	ffdaf83202af0d72	b6a36a3431547328
	8c059af157146580	870dbce23bad8988	8045a82f723a9b4e	fa8bb40b4e08100f
$t = 70$:	20b0a9a1d21c482d	6189f68162b256b5	12737373d2985232	ffdaf83202af0d72
	f22b874c96785ec8	8c059af157146580	870dbce23bad8988	8045a82f723a9b4e
$t = 71$:	ef6d863c2127b394	20b0a9a1d21c482d	6189f68162b256b5	12737373d2985232
	b7aee28337d69dab	f22b874c96785ec8	8c059af157146580	870dbce23bad8988
$t = 72$:	d3efe8b442689074	ef6d863c2127b394	20b0a9a1d21c482d	6189f68162b256b5
	22491ab9cdecb6b0	b7aee28337d69dab	f22b874c96785ec8	8c059af157146580
$t = 73$:	4694354944a9f487	d3efe8b442689074	ef6d863c2127b394	20b0a9a1d21c482d
	659890a5818d0c50	22491ab9cdecb6b0	b7aee28337d69dab	f22b874c96785ec8
$t = 74$:	b93c2403773dd08c	4694354944a9f487	d3efe8b442689074	ef6d863c2127b394
	88c2c2ac52c4f679	659890a5818d0c50	22491ab9cdecb6b0	b7aee28337d69dab
$t = 75$:	025848e3ab6b69d3	b93c2403773dd08c	4694354944a9f487	d3efe8b442689074
	750da3d4e16a1b64	88c2c2ac52c4f679	659890a5818d0c50	22491ab9cdecb6b0
$t = 76$:	396b53e58d04471b	025848e3ab6b69d3	b93c2403773dd08c	4694354944a9f487
	700486bf252cba75	750da3d4e16a1b64	88c2c2ac52c4f679	659890a5818d0c50
$t = 77$:	51b6f9a3c1ceeb4a	396b53e58d04471b	025848e3ab6b69d3	b93c2403773dd08c
	e6b3850de8ae6230	700486bf252cba75	750da3d4e16a1b64	88c2c2ac52c4f679
$t = 78$:	526a98f5dc595406	51b6f9a3c1ceeb4a	396b53e58d04471b	025848e3ab6b69d3
	4f0dcf74aea76f90	e6b3850de8ae6230	700486bf252cba75	750da3d4e16a1b64
$t = 79$:	deb3eeaa973bb9dd	526a98f5dc595406	51b6f9a3c1ceeb4a	396b53e58d04471b
	3665b5dbb6c2e055	4f0dcf74aea76f90	e6b3850de8ae6230	700486bf252cba75

That completes the processing of the second and final message block, $M^{(2)}$. The final hash value, $H^{(2)}$, is calculated to be

$$H_0^{(2)} = \texttt{2a7f1d895fd58e0b} + \texttt{deb3eeaa973bb9dd} = \texttt{09330c33f71147e8}$$
$$H_1^{(2)} = \texttt{eaae96d1a673c741} + \texttt{526a98f5dc595406} = \texttt{3d192fc782cd1b47}$$
$$H_2^{(2)} = \texttt{015a2173796c1a88} + \texttt{51b6f9a3c1ceeb4a} = \texttt{53111b173b3b05d2}$$
$$H_3^{(2)} = \texttt{f6352ca156acaff7} + \texttt{396b53e58d04471b} = \texttt{2fa08086e3b0f712}$$
$$H_4^{(2)} = \texttt{c662113e9ebb4d64} + \texttt{3665b5dbb6c2e055} = \texttt{fcc7c71a557e2db9}$$

$H_5^{(2)}$ = 17b61a85e2ccf0a9 + 4f0dcf74aea76f90 = 66c3e9fa91746039

$H_6^{(2)}$ = 37eb9a6660feb519 + e6b3850de8ae6230 = 1e9f1f7449ad1749

$H_7^{(2)}$ = 8f2ebe9a81e6a2c5 + 700486bf252cba75 = ff334559a7135d3a.

The final hash value is truncated to its left-most 384 bits (i.e., $H_0^{(1)}, \ldots, H_5^{(1)}$), resulting in the 384-bit message digest

```
09330c33f71147e8 3d192fc782cd1b47 53111b173b3b05d2 2fa08086e3b0f712
fcc7c71a557e2db9 66c3e9fa91746039.
```

D.3 SHA-384 Example (Long Message)

Let the message *M* be the binary-coded form of the ASCII string which consists of 1,000,000 repetitions of the character "**a**". The resulting SHA-384 message digest is

```
9d0e1809716474cb 086e834e310a4a1c ed149e9c00f24852 7972cec5704c2a5b
07b8b3dc38ecc4eb ae97ddd87f3d8985.
```

APPENDIX E: REFERENCES

[180-1] Federal Information Processing Standards (FIPS) Publication 180-1, *Secure Hash Standard (SHS)*, U.S. DoC/NIST, April 17, 1995.

[HAC] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press, Inc., October 1997.

American National Standard
for Financial Services

X9.63–2001

Public Key Cryptography for the Financial Services Industry

Key Agreement and Key Transport Using Elliptic Curve Cryptography

Secretariat:
American Bankers Association

Approved: November 20, 2001

American National Standards Institute

Foreword

Business practice has changed with the introduction of computer-based technologies. The substitution of electronic transactions for their paper-based predecessors has reduced costs and improved efficiency. Trillions of dollars in funds and securities are transferred daily by telephone, wire services, and other electronic communication mechanisms. The high value or sheer volume of such transactions within an open environment exposes the financial community and its customers to potentially severe risks from the accidental or deliberate disclosure, alteration, substitution, or destruction of data. These risks are compounded by interconnected networks, and the increased number and sophistication of malicious adversaries. Electronically communicated data may be secured through the use of symmetrically keyed encryption algorithms (e.g. ANSI X9.52, Triple-DEA) in combination with public-key cryptography-based key management techniques.

This standard, X9.63-2001, *Public Key Cryptography For The Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, defines a suite of mechanisms designed to facilitate the secure establishment of cryptographic data for the keying of symmetrically keyed algorithms (e.g. DEA, TDEA). These mechanisms are based on the elliptic curve analogue of the Diffie-Hellman key agreement mechanism [4]. Because the mechanisms are based on the same fundamental mathematics as the Elliptic Curve Digital Signature Algorithm (ECDSA) (see [7]), additional efficiencies and functionality may be obtained by combining these and other cryptographic techniques.

While the techniques specified in this standard are designed to facilitate key management applications, the standard does not guarantee that a particular implementation is secure. It is the responsibility of the financial institution to put an overall process in place with the necessary controls to ensure that the process is securely implemented. Furthermore, the controls should include the application of appropriate audit tests in order to verify compliance.

The user's attention is called to the possibility that compliance with this standard may require the use of an invention covered by patent rights. By publication of this standard, no position is taken with respect to the validity of potential claims or of any patent rights in connection therewith. The patent holders have, however, filed a statement of willingness to grant a license under these rights on reasonable and nondiscriminatory terms and conditions to applicants desiring to obtain such a license. Details may be obtained from the X9 Secretariat,

Copyright 2001 by American Bankers Association
All rights reserved.
No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without prior written permission of the publisher. Printed in the United States of America

Suggestions for the improvement or revision of this standard are welcome. They should be sent to the X9 Secretariat, American Bankers Association, 1120 Connecticut Avenue, N.W., Washington D.C. 20036.

This standard was processed and approved for submittal to ANSI by the Accredited Standards Committee on Financial Services, X9. Committee approval of the standard does not necessarily imply that all the committee members voted for its approval.

At the time that this standard was approved, the X9 Committee had the following members:

Harold G. Deal, X9 Chairman, BB&T
Vincent DeSantis, X9 Vice Chairman, New York Clearing House
Cynthia L. Fuller, Managing Director
Darlene J. Schubert, Program Manager

The X9 committee had the following members:

Organization	Representative
ACI Worldwide	Jim Shafer
ACI Worldwide	Cindy Rink
American Bankers Association	Stephen Schutze
American Bankers Association	Michael Scully
American Express Company	Mike Jones
American Express Company	Dick Schreiber
American Express Company	Gerry Smith
American Express Company	Barbara Wakefield
BB&T	Harold Deal
Bank One Corporation	Jacqueline Pagan
Bank One Corporation	Kimberly Ray
Bank of America	Mack Hicks
Bank of America	Richard Phillips
Bank of America	Daniel Welch
BancTec, Inc.	Christopher Dowdell
BancTec, Inc	David Hunt
Certicom Corporation	Daniel Brown
Certicom Corporation	Donald Johnson
Citigroup, Inc	Mark Scott
Citigroup, Inc	Daniel Schutzer
Citigroup, Inc	Skip Zehnder
Check Solutions	Jerry Bowman
Check Solutions	Donald Harman
Check Solutions	Ron Schultz
Compaq Computer Corp.	Larry Hines
Compaq Computer Corp.	Gary Lefkowitz
Datum	John Bernardi

Datum ..... Sandra Lambert
Datum ..... Jerry Willett
Diebold, Inc. ..... Bruce Chapa
Diebold, Inc. ..... Judy Edwards
Deluxe Corporation ..... Maury Jansen
Discover Financial Services ..... Pamela Ellington
Discover Financial Services ..... Masood Mirza
Discover Financial Services ..... Patsie Rinchiuso
eFunds Corporation ..... Chuck Bram
eFunds Corporation ..... Richard Fird
eFunds Corporation ..... Forrest Martin
eFunds Corporation ..... Joseph Stein
eFunds Corporation ..... Cory Surges
eFunds Corporation ..... Daniel Rick
Federal Reserve Bank ..... Dexter Holt
Federal Reserve Bank ..... Jeannine DeLano
First Data Corporation ..... Gene Kathol
Food Marketing Institute ..... Ted Mason
Food Marketing Institute ..... Stacy Fitzgerald-Redd
Griffin Consulting ..... Harriette Griffin
Griffin Consulting ..... Phillip H. Griffin
HW and W Inc. ..... Martin Ferris
JP Morgan Chase and Co. ..... Robert Blair
JP Morgan Chase and Co. ..... Richard Yen
KPMG Peat Marwick LLP ..... Al Van Ranst
KPMG Peat Marwick LLP ..... Jeff Stapleton
Mag-Tek, Inc. ..... Terry Bensen
Mag-Tek, Inc. ..... Jeff Duncan
Mag-Tek, Inc. ..... Mimi Hart
Mag-Tek, Inc ..... Carlos Morales
MasterCard International ..... Ron Karlin
MasterCard International ..... Naiyre Foster
Mellon Bank, N.A. ..... Richard Adams
Mellon Bank, N.A. ..... Jennifer Smith
Mellon Bank, N.A. ..... David Taddeo
Merrill Lynch ..... John Dolan
Merrill Lynch ..... Dave Yeger
National Association of Convenience Stores ..... John Hervey
National Association of Convenience Stores ..... Teri Richman
National Association of Convenience Stores ..... Robert Swanson
National Security Agency ..... Greg Bergren
National Security Agency ..... Sheila Brand
NCR Corporation ..... David Norris
NCR Corporation ..... Steve Stevens
New York Clearing House ..... Vincent DeSantis

New York Clearing House ..... John Dunn
PricewaterhouseCoopers ..... Jeff Zimmerman
Silas Technologies..... Andrew Garner
Silas Technologies..... Ray Gatland
SPYRUS..... Karen Randall
SPYRUS..... James Randall
Star Systems, Inc. ..... Elizabeth Lynn
Star Systems, Inc ..... Michael Wade
Sun Microsystems ..... Yvonne Humphery
Sun Microsystems ..... Joel Weise
Unisys Corporation ..... Thomas Hayosh
Unisys Corporation ..... Navnit Shah
VeriFone, Inc..... Brad McGuinness
VeriFone, Inc..... John Sheets
VeriFone, Inc..... Brenda Watlington
Visa International ..... Patricia Greenhalgh
Wells Fargo Bank..... Terry Leahy
Wells Fargo Bank..... Ruven Schwartz

The X9F subcommittee on Data and Information Security had the following members:

Mr. Richard J. Sweeney, Chairman, Inovant

Organization Represented	Representative
ACI Worldwide	Cindy Rink
ACI Worldwide	Jim Shaffer
American Bankers Association	Stephen Schutze
American Bankers Association	Donald Rhodes
American Express Company	Mike Jones
American Express Company	Mark Merkow
American Express Company	Gerry Smith
American Express Company	Dick Schreiber
BancTec, Inc.	Christopher Dowdell
Bank of America	Mack Hick
Bank of America	Richard Phillips
Bank of America	Craig Worstell
Bank One Corporation	Mark Ryding
BB&T	Harold Deal
Caradas	John Gould
Caradas	Tom Johnston
Caradas	Richard Kastner
Certicom Corporation	Daniel Brown
Certicom Corporation	Donald Johnson
Certicom Corporation	Brenda Klein
Certicom Corporation	Sherry Vanstone

Check Solutions..... Harry Hankla
Check Solutions..... Ron Schultz
Check Solutions..... Jerry Bowman
Chrysalis-ITS..... Terry Fletcher
Communications Security Establishment..... Alan Poplove
Communications Security Establishment..... Mike Chawrun
Compaq Computer Corporation..... Larry Hines
Compaq Computer Corporation..... Gary Lefkowitz
Datum, Inc..... Sandra Lambert
Deluxe Corporation..... Maury Jansen
Diebold, Inc..... Bruce Chapa
Diebold, Inc..... Judy Edwards
Digital Signature Trust..... Brandon Brown
Digital Signature Trust..... Trent Henry
Discover Financial Services..... Pamela Ellington
Discover Financial Services..... Masood Mirza
Diversinet Corporation..... Michael Crerar
eFunds Corporation..... Chuck Bram
eFunds Corporation..... Forrest Martin
Entrust Technologies..... Santosh Chokhani
Entrust Technologies..... Miles Smid
Entrust Technologies..... Mike Just
Federal Reserve Bank..... Dexter Holt
First Data Corporation..... Gene Kathol
Food Marketing Institute..... Ted Mason
Food Marketing Instittute..... Stacy Fitzgerald-Redd
Griffin Consulting..... Phillip H. Griffin
Griffin Consulting..... Harriette Griffin
H W and W, Inc..... Martin Ferris
IBM Corporation..... Michael Kelly
IBM Corporation..... Stephen Mike Matyas
Ingenico Canada, Ltd. ..... John Spence
Inovant..... Richard Sweeney
Jones Futurex, Inc..... Ray Bryan
Jones Futurex, Inc..... Steve Junod
JP Morgan Chase & Co..... Robert Blair
JP Morgan Chase & Co..... Richard Yen
KPMG Peat Marwick LLP..... Jeff Stapleton
KPMG Peat Marwick LLP..... Al Van Ranst, Jr.
KPMG Peat Marwick LLP..... Azita Amini
Mag-Tek, Inc..... Mimi Hart
Mag-Tek, Inc..... Terry Benson
MasterCard International..... Ron Karlin
MasterCard International..... William Poletti
Mellon Bank, N.A. ..... David Taddeo

Merrill Lynch ..... Lawrence LaBella
Merril Lynch..... Jennifer Smith
National Association of Convenience Stores ..... John Hervey
National Association of Convenience Stores ..... Robert Swanson
National Security Agency..... Gregory Bergren
National Security Agency..... Sheila Brand
National Security Agency..... John Stevens
nCipher ..... William Franklin
NCR Corporation ..... David Norris
NCR Corporation ..... Adrian Shields
NCR Corporation ..... Steve Stevens
NIST ..... Elaine Barker
NIST ..... Lawrence Bassham III
NIST ..... Morris Dworkin
NIST ..... Annabelle Lee
Pitney Bowes, Inc. ..... Andrei Obrea
Pitney Bowes, Inc. ..... Leon Pintsov
Pitney Bowes, Inc. ..... Matthew Campagna
PricewaterhouseCoopers ..... Jeff Zimmerman
Rainbow Technologies ..... Georgina Schroder
Rainbow Technologies ..... Vic Sundararajan
RSA Securities ..... Russ Housley
RSA Securities ..... Robert Silverman
SPYRUS ..... Karen Randall
SPYRUS ..... James Randall
Star Systems, Inc. ..... Elizabeth Lynn
Star Systems, Inc. ..... Michael Wade
Star Systems, Inc. ..... Carol Fazzone
Sun Microsystems PS ..... Yvonne Humphery
Sun Microsystems PS ..... Joel Weise
TECSEC Incorporated..... Ed Scheidt
TECSEC Incorporation ..... Pud Reaver
TECSEC Incorporated..... Jay Wack
VeriFone ..... John Sheets
Verisign, Inc. ..... Warwick Ford
VISA International ..... Richard Hite
Wells Fargo Bank ..... Terry Leahy
Wells Fargo Bank ..... Gordon Martin
Wells Fargo Bank ..... Ruven Schwartz
Zaxus, Inc. ..... Samuel Epstein
Zefer Boston ..... Michael Versace

The X9F1 Cryptographic Tool Standards and Guidelines group that developed this standard had the following members:

Miles Smid, Chairman, Entrust Technologies
Phillip H. Griffin, Vice Chair, Griffin Consulting
Daniel Brown, Project Editor, Certicom Corporation

Organization	Representative
Certicom Corporation	Daniel Brown
Certicom Corporation	Don Johnson
Certicom Corporation	Alfred Menezes
Certicom Corporation	Scott Vanstone
Certicom Corporation	Simon Blake-Wilson
Chrysalis-ITS	Francois Rousseau
Chrysalis-ITS	Terry Fletcher
Communications Security Establishment of Canada	Mike Chawrun
Communications Security Establishment of Canada	Alan Poplove
Diversinet	Michael Crerar
Entrust	Miles Smid
Entrust	Robert Zuccherato
Federal Reserve Bank of Atlanta	John Hannan
Federal Reserve Bank of Atlanta	Jeff Harris
Griffin Consulting	Phillip H. Griffin
IBM Corporation	Todd Arnold
IBM Corporation	Allen Roginsky
IBM Corporation	Steven Matyas
JP Morgan Chase & Co.	Gene Rao
JP Morgan Chase & Co.	Richard Yen
M. Blake Greenlee Associates, Ltd.	M. Blake Greenlee
Merrill Lynch	Larry LaBella
National Institute of Standards and Technology	Morris Dworkin
National Institute of Standards and Technology	Elaine Barker
National Institute of Standards and Technology	Sharon Keller
National Security Agency	Paul Timmel
National Security Agency	Bob Reiter
Pitney Bowes, Inc.	Leon Pintsov
RSA Security	Russ Housley
RSA Security	Burt Kaliski
RSA Security	Robert Silverman
SPYRUS	Karen Randall
SPYRUS	James Randall
SPYRUS	Peter Yee
TecSec	Ersin Domangue

Contents

Annex K
(informative)
Bibliography

A comprehensive treatment of modern cryptography can be found in [57].

Elliptic curve cryptosystems were first proposed in 1985 independently by Neil Koblitz [48] and Victor Miller [58]. Since then, much research has been undertaken towards improving the efficiency of these systems and evaluating their security. For a summary of this work, consult [54]. A description of a hardware implementation of an elliptic curve cryptosystem can be found in [12]. ECDSA is specified in [7]. For a detailed treatment of the mathematical theory of elliptic curves, see [63]. A less technical approach to the theory can be found in [54].

Three references on the theory of finite fields are the books of McEliece [53], Lidl and Neiderreiter [52], and Jungnickel [44]. Lidl and Neiderreiter's book [52] contains introductory material on polynomial and normal bases. The article [11] discusses methods that efficiently perform arithmetic operations in finite fields of characteristic 2. A hardware implementation of arithmetic in such fields that exploits the properties of optimal normal bases is described in [13].

SHA-1 is specified in [3] and [25].

The SHA-1-based MAC scheme is HMAC, which was introduced in [15].

The asymmetric encryption scheme specified in this Standard was introduced in [18]. Preliminary work by the same authors can be found in [17].

The fundamental concept of asymmetric key agreement was introduced in [23]. The extensions to the traditional Diffie-Hellman primitive specified in this Standard were introduced in [19], [24], and [56]. See also [50]. These key agreement schemes have also been standardized in the algebraic context of the multiplicative group of a finite field [4].

The key transport schemes specified here are based on those in [39]. Also closely related are the entity authentication schemes specified in [38] and [26].

ASN.1 is described in [32]-[37]. BER and DER can be found in [36].

1. ANSI X3.92-1981: *Data Encryption Algorithm*. December 30, 1981.

2. ANSI X9.19-1996: *Financial Institution Retail Message Authentication*. 1996.

3. ANSI X9.30-1993, Part 2: *Public Key Cryptography using Irreversible Algorithms for the Financial Services Industry: The Secure Hash Algorithm 1* (*SHA-1*)(*Revised*). 1993.

4. ANSI X9.42-2001: *Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Algorithm Keys Using Diffie-Hellman*. 2001.

5. ANSI X9.52-1996: *Triple Data Encryption Algorithm Modes of Operation*. July, 1998.

6. ANSI X9.57-1997: *Public Key Cryptography for the Financial Services Industry: Certificate Management*. 1997.

7. ANSI X9.62-1999: *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm* (*ECDSA*). 1998.

8. ANSI X9.70-200x: *Management of Symmetric Keys Using Public Key Algorithms*. Working Draft.

9. ANSI X9.71-1999: *Keyed Hash Message Authentication Code*. 1999.

10. ANSI X9.80-2000: *Prime Number Generation*. 2000.

11. G. Agnew, T. Beth, R. Mullin, and S. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptology*, 6, pages 3-13, 1993.

12. G. Agnew, R. Mullin, and S. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11, pages 804-813, 1993.

13. G. Agnew, R. Mullin, I. Onyszchuk, and S. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3, pages 63-79, 1991.

14. L. Bassham, R. Housley, and W. Polk. *Algorithms and identifiers for the Internet X.509 Public Key Infrastructure*. Internet Engineering Task Force, internet-draft, 2001.

15. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In Advances in Cryptology: Crypto '96, pages 1-15, 1996.

16. M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology: Crypto '93*, pages 232-249, 1993.

17. M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security*, pages 62-73, 1993.

18. M. Bellare and P. Rogaway. Minimizing the use of random oracles in authenticated encryption schemes. In *Proceedings of PKS '97*, 1997.

19. S. Blake-Wilson, D. Johnson, and A.J. Menezes. Key agreement protocols and their security analysis. In *Cryptography and Coding*, 6th IMA Conference, Springer-Verlag, pages 30-45, 1997.

20. M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener. *Minimal key lengths for symmetric ciphers to provide adequate commercial security*. January, 1996.

21. D. Boneh and R.J. Lipton. Algorithms for black-box fields and their application to cryptography. In *Advances in Cryptology: Crypto '96*, pages 283-297, 1996.

22. E. Brickell, D. Gordon, K. McCurley, and D. Wilson. Fast Exponentiation with precomputation. In *Advances in Cryptology: EuroCrypt '92*, pages 200-207, 1993.

23. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*. IT-22(6): 644-654, November 1976.

24. W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes, and Cryptography*. 2, pages 107-125, 1992.

25. FIPS 180-1. *Secure Hash Standard*. Federal Information Processing Standards Publication 180-1, 1995.

26. FIPS 196. *Entity Authentication using Public Key Cryptography*. Federal Information Processing Standards Publication 196, February 18, 1997.

27. M. Fouquet, P. Gaudry, and R. Harley. An extension of Satoh's algorithm and its implementation. *Journal of the Ramanujan Mathematical Society*, 15, pages 281-318, 2000.

28. G. Frey and H.-G. Ruck. A remark concerning m-divisibility and the discrete logarithm problem in the divisor class group of curves. *Mathematics of Computation*, 62, pages 865-874. 1994.

29. R. Gallant, R. Lambert, and S. Vanstone. Improving the parallelized Pollard lambda search on binary anomalous curves. To appear in *Mathematics of Computation*.

30. IEEE 1363-2000. *Standard Specifications for Public-Key Cryptography*. 2000.

31. IEEE P1363A. *Standard for Public-Key Cryptography - Addendum*. July 11, 1997. Working Document.

32. ISO/IEC 8824-1 | ITU-T Recommendation X.680. Information Technology - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.

33. ISO/IEC 8824-2 | ITU-T Recommendation X.681. Information Technology - Abstract Syntax Notation One (ASN.1): Information Object Specification.

34. ISO/IEC 8824-3 | ITU-T Recommendation X.682. Information Technology - Abstract Syntax Notation One (ASN.1): Constraint Specification.

35. ISO/IEC 8824-4 | ITU-T Recommendation X.683. Information Technology - Abstract Syntax Notation One (ASN.1): Parametrization of ASN.1 Specifications.

36. ISO/IEC 8825-1 | ITU-T Recommendation X.690. Information Technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).

37. ISO/IEC 8825-2 | ITU-T Recommendation X.691. Information Technology - ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER).

38. ISO/IEC 9798-3. *Information technology – Security techniques – Entity authentication – Part 3: Mechanisms using asymmetric signature techniques*. April 1, 1997. Review document.

39. ISO/IEC 11770-3. *Information technology - Security techniques - Key management - Part 3: Mechanisms using asymmetric signature techniques*. March 22, 1996.

40. ISO/IEC 15946-1. *Cryptographic techniques based on elliptic curves – Part 1: Gen*eral. 2001. Working draft.

41. ISO/IEC 15946-2. *Cryptographic techniques based on elliptic curves – Part 2: Signatures*. 2000. Working draft.

42. ISO/IEC 15946-3. *Cryptographic techniques based on elliptic curves – Part 3: Key establishment*. 2001. Working draft.

43. D. Johnson. *Diffie-Hellman Key Agreement Small Subgroup Attack, a Contribution to X9F1 by Certicom*. July 16, 1996.

44. D. Jungnickel. *Finite Fields: Structure and Arithmetics*. B.I.Wissenschaftsverlag, Mannheim, 1993.

45. B. Kaliski. MQV vulnerability. Posting to ANSI X9F1 and IEEE P1363 newsgroups. 1998.

46. D. Knuth. *The Art of Computer Programming*. Volume 1, Addison-Wesley, Reading, Massachusetts, 1973.

47. D. Knuth, *The Art of Computer Programming*. Volume 2, 2nd edition, 1981.

48. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48, pages 203-209, 1987.

49. N. Koblitz. *A Course in Number Theory and Cryptography*, Springer-Verlag, 2nd edition, 1994.

50. L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone. An efficient protocol for authenticated key agreement. Technical report CORR 98-05, Department of Combinatorics & Optimization, University of Waterloo, March, 1998.

51. A. Lenstra and E. Verheul. Selecting cryptographic key sizes. *Proceedings of PKC 2000*, Springer-Verlag, pages 446-465, 2000.

52. R. Lidl and H. Neiderreiter. *Finite Fields*. Cambridge University Press, 1987.

53. R.J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

54. A.J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

55. A.J. Menezes, T. Okamoto, and S.A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39, pages 1639-1646, 1993.

56. A.J. Menezes, M. Qu, and S.A. Vanstone. Some new key agreement protocols providing implicit authentication. Workshop record. *2nd Workshop on Selected Areas in Cryptography* (*SAC '95*), Ottawa, Canada, May 18-19, 1995.

57. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

58. V. Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptology: Crypto '85*, pages 417-426, 1985.

59. A. Odlyzko. The Future of Integer Factorization. *CryptoBytes*, volume 1, number 2, pages 5-12, summer 1995.

60. J. Pollard. Monte Carlo methods for index computation mod *p*. *Mathematics of Computation*, 32, pages 918-924, 1978.

61. T. Satoh and K. Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Commentarii Mathematici Universitatis Pauli*, 47, pages 81-92, 1998.

62. R. Schoof. Elliptic curves over finite fields and the computation of square roots mod *p*. *Mathematics of Computation*, 44, pages 483-494, 1987.

63. J. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag, New York, 1985.

64. N. Smart. The discrete logarithm problem on elliptic curves of trace one. *Journal of Cryptology*, 12, pages 193-196, 1999.

65. P.C. van Oorschot and M. Wiener. Parallel collision search with applications to hash functions and discrete logarithms. *2nd ACM Conference on Computer and Communications Security*, pages 210-218, ACM Press. 1994.

66. M. Wiener and R. Zuccherato. Fast attacks on elliptic curve cryptosystems. To appear in *Fifth Annual Workshop on Selected Areas in Cryptography – SAC '98*, Lecture Notes in Computer Science, Springer-Verlag.

FIPS PUB 186-2
(+Change Notice)

FEDERAL INFORMATION
PROCESSING STANDARDS PUBLICATION

2000 January 27

U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology

DIGITAL SIGNATURE STANDARD (DSS)

CATEGORY: COMPUTER SECURITY

U.S. DEPARTMENT OF COMMERCE, William M. Daley, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY,
Raymond G. Kammer, Director

Foreword

The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST) is the official series of publications relating to standards and guidelines adopted and promulgated under the provisions of Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106), and the Computer Security Act of 1987 (Public Law 100-235). These mandates have given the Secretary of Commerce and NIST important responsibilities for improving the utilization and management of computer and related telecommunications systems in the Federal Government. The NIST, through its Information Technology Laboratory, provides leadership, technical guidance, and coordination of Government efforts in the development of standards and guidelines in these areas.

Comments concerning Federal Information Processing Standards Publications are welcomed and should be addressed to the Director, Information Technology Laboratory, National Institute of Standards and Technology, 100 Bureau Dr. Stop 8900, Gaithersburg, MD 20899-8900.

William Mehuron, Director
Information Technology Laboratory

Abstract

This standard specifies a suite of algorithms which can be used to generate a digital signature. Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity of the signatory. In addition, the recipient of signed data can use a digital signature in proving to a third party that the signature was in fact generated by the signatory. This is known as nonrepudiation since the signatory cannot, at a later time, repudiate the signature.

Key words: ADP security, computer security, digital signatures, public-key cryptography, Federal Information Processing Standards.

Federal Information
Processing Standards Publication 186-2

2000 January 27

Announcing the

DIGITAL SIGNATURE STANDARD (DSS)

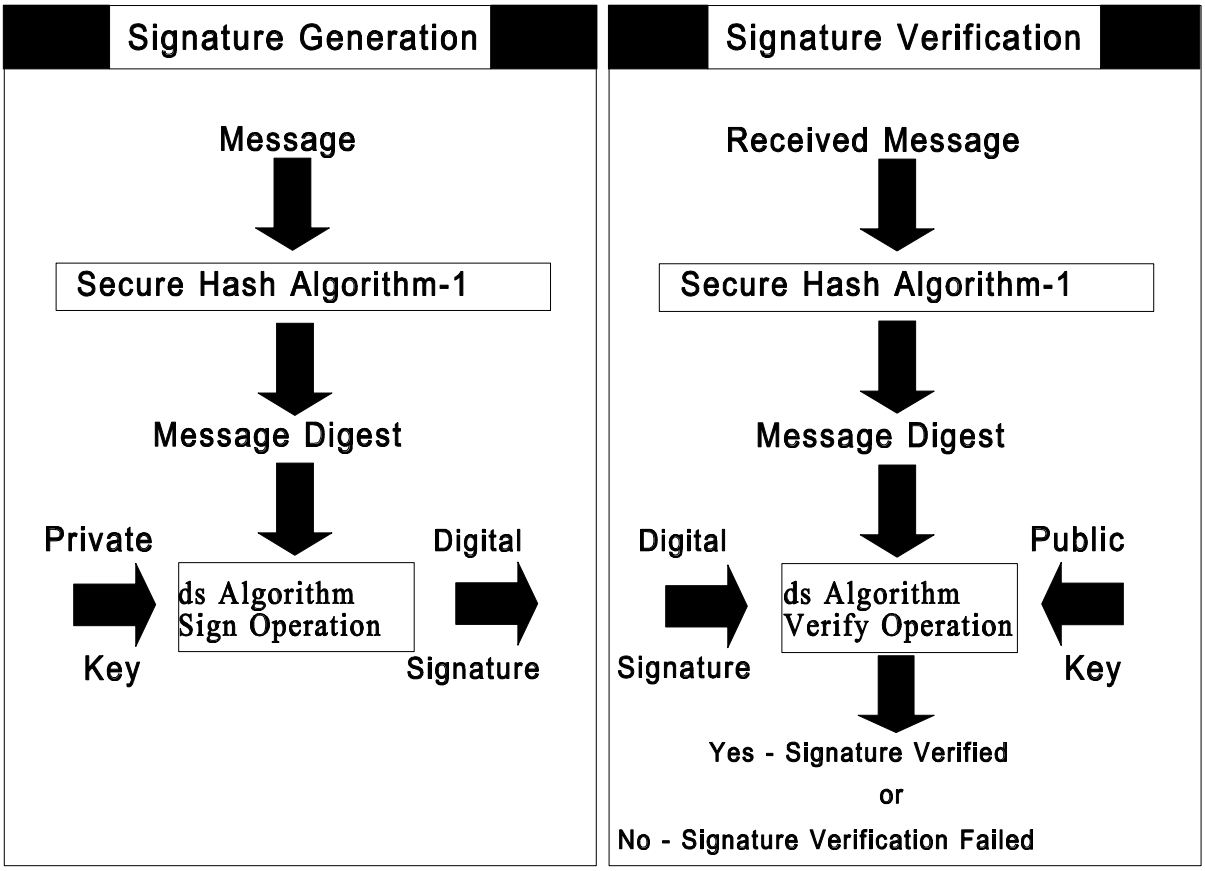
Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106), and the Computer Security Act of 1987 (Public Law 100-235).

Name of Standard: Digital Signature Standard (DSS).

Category of Standard: Computer Security, Cryptography.

Explanation: This Standard specifies algorithms appropriate for applications requiring a digital, rather than written, signature. A digital signature is represented in a computer as a string of binary digits. A digital signature is computed using a set of rules and a set of parameters such that the identity of the signatory and integrity of the data can be verified. An algorithm provides the capability to generate and verify signatures. Signature generation makes use of a private key to generate a digital signature. Signature verification makes use of a public key which corresponds to, but is not the same as, the private key. Each user possesses a private and public key pair. Public keys are assumed to be known to the public in general. Private keys are never shared. Anyone can verify the signature of a user by employing that user's public key. Signature generation can be performed only by the possessor of the user's private key.

A hash function is used in the signature generation process to obtain a condensed version of data, called a message digest (see Figure 1). The message digest is then input to the digital signature (ds) algorithm to generate the digital signature. The digital signature is sent to the intended verifier along with the signed data (often called the message). The verifier of the message and signature verifies the signature by using the sender's public key. The same hash function must also be used in the verification process. The hash function is specified in a separate standard, the Secure Hash Standard (SHS), FIPS 180-1. FIPS approved ds algorithms must be implemented with the SHS. Similar procedures may be used to generate and verify signatures for stored as well as transmitted data.



Approving Authority: Secretary of Commerce.

Maintenance Agency: U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL).

Applicability: This standard is applicable to all Federal departments and agencies for the protection of sensitive unclassified information that is not subject to section 2315 of Title 10, United States Code, or section 3502(2) of Title 44, United States Code. This standard shall be used in designing and implementing public-key based signature systems that Federal departments and agencies operate or which are operated for them under contract. Adoption and use of this standard is available to private and commercial organizations.

Applications: A digital signature (ds) algorithm authenticates the integrity of the signed data and the identity of the signatory. A ds algorithm may also be used in proving to a third party that data was actually signed by the generator of the signature. A ds algorithm is intended for use in electronic mail, electronic funds transfer, electronic data interchange, software distribution, data storage, and

other applications that require data integrity assurance and data origin authentication. The techniques specified in ANSI X9.31 and ANSI X9.62 may be used in addition to the Digital Signature Algorithm (DSA) specified herein. (NIST editorial note: either DSA, RSA [ANSI X9.31], or ECDSA [ANSI X9.62] may be used; all three do not have to be implemented.)

Implementations: A ds algorithm may be implemented in software firmware, hardware or any combination thereof. NIST has developed a validation program to test implementations for conformance to DSA. Currently, conformance tests for ANSI X9.31 and ANSI X9.62 have not been developed. These tests will be developed and made available in the future. Information about the planned validation program can be obtained from the National Institute of Standards and Technology, Information Technology Laboratory, Attn: DSS Validation, 100 Bureau Drive Stop 8930, Gaithersburg, MD 20899-8930.

Agencies are advised that separate keys should be used for signature and confidentiality purposes when using the X9.31 standard. This is because the RSA algorithm can be used for both data encryption and digital signature purposes.

Export Control: Certain cryptographic devices and technical data regarding them are subject to Federal export controls. Applicable Federal government export controls are specified in Title 15, Code of Federal Regulations (CFR) Part 740.17; Title 15, CFR Part 742; and Title 15, CFR Part 774, Category 5, Part 2.

Patents: The algorithms in this standard may be covered by U.S. or foreign patents.

Implementation Schedule: This standard becomes effective July 27, 2000. A transition period from July 27, 2000 until July 27, 2001 is provided to enable all agencies to develop plans for the acquisition of equipment which implements the digital signature techniques adopted by FIPS 186-2. During the transition period, agencies may continue to use their existing digital signature systems and to acquire additional equipment that may be needed to interoperate with these legacy digital signature systems. Agencies without legacy digital signature systems should plan for the acquisition and use of equipment implementing the digital signature techniques that are adopted by FIPS 186-2. After the transition period, only equipment that implements FIPS 186-2 endorsed techniques should be acquired.

Specifications: Federal Information Processing Standard (FIPS) 186-2 Digital Signature Standard (affixed). Also see an important change notice at the end of this document.

Cross Index:

a. FIPS PUB 46-3, Data Encryption Standard.

b. FIPS PUB 73, Guidelines for Security of Computer Applications.

c. FIPS PUB 140-1, Security Requirements for Cryptographic Modules.

d. FIPS PUB 171, Key Management Using ANSI X9.17.

e. FIPS PUB 180-1, Secure Hash Standard.

f. ANSI X9.31-1998, Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA).

g. ANSI X9.62-1998, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).

Qualifications: The security of a digital signature system is dependent on maintaining the secrecy of users' private keys. Users must therefore guard against the unauthorized acquisition of their private keys. While it is the intent of this standard to specify general security requirements for generating digital signatures, conformance to this standard does not assure that a particular implementation is secure. The responsible authority in each agency or department shall assure that an overall implementation provides an acceptable level of security. This standard will be reviewed every five years in order to assess its adequacy.

Waiver Procedure: Under certain exceptional circumstances, the heads of Federal agencies, or their delegates, may approve waivers to Federal Information Processing Standards (FIPS). The head of such agency may redelegate such authority only to a senior official designated pursuant to section 3506(b) of Title 44, United States Code. Waiver shall be granted only when:

a. Compliance with a standard would adversely affect the accomplishment of the mission of an operator of a Federal computer system; or

b. Cause a major adverse financial impact on the operator which is not offset by Government wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision which explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decisions, 100 Bureau Drive Stop 8970, Gaithersburg, MD 20899-8970.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Governmental Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under 5 U.S.C. Sec. 552(b), shall be part of the procurement documentation and retained by the agency.

Where to Obtain Copies of the Standard: Copies of this publication are for sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. When ordering, refer to Federal Information Processing Standards Publication 186-2 (FIPSPUB186-2), and identify the title. When microfiche is desired, this should be specified. Prices are published by NTIS in current catalogs and other issuances. Payment may be made by check, money order, deposit account or charged to a credit card accepted by NTIS.

Federal Information
Processing Standards Publication 186-2

2000 January 27

Specifications for the

DIGITAL SIGNATURE STANDARD (DSS)

1. INTRODUCTION

This publication prescribes three algorithms suitable for digital signature (ds) generation and verification. The first algorithm, the Digital Signature Algorithm (DSA), is described in sections 4 - 6 and appendices 1 - 5. The second algorithm, the RSA ds algorithm, is discussed in section 7 and the third algorithm, the ECDSA algorithm, is discussed in section 8 and recommended elliptic curves in appendix 6. An important change notice has been appended to this document.

2. GENERAL

When a message is received, the recipient may desire to verify that the message has not been altered in transit. Furthermore, the recipient may wish to be certain of the originator's identity. Both of these services can be provided by a ds algorithm. A digital signature is an electronic analogue of a written signature in that the digital signature can be used in proving to the recipient or a third party that the message was, in fact, signed by the originator. Digital signatures may also be generated for stored data and programs so that the integrity of the data and programs may be verified at any later time.

This publication prescribes two algorithms suitable for digital signature generation and verification.

3. USE OF A DIGITAL SIGNATURE (ds) ALGORITHM

A ds algorithm is used by a *signatory* to generate a digital signature on data and by a *verifier* to verify the authenticity of the signature. Each signatory has a public and private key. The private key is used in the signature generation process and the public key is used in the signature verification process. For both signature generation and verification, the data which is referred to as a message,

M, is reduced by means of the Secure Hash Algorithm (SHA-1) specified in FIPS 180-1. An adversary, who does not know the private key of the signatory, cannot generate the correct signature of the signatory. In other words, signatures cannot be forged. However, by using the signatory's public key, anyone can verify a correctly signed message. A means of associating public and private key pairs to the corresponding users is required. That is, there must be a binding of a user's identity and the user's public key. This binding may be certified by a mutually trusted party. For example, a certifying authority could sign credentials containing a user's public key and identity to form a certificate. Systems for certifying credentials and distributing certificates are beyond the scope of this standard. NIST intends to publish separate document(s) on certifying credentials and distributing certificates.

4. DSA PARAMETERS

The DSA makes use of the following parameters:

1. p = a prime modulus, where $2^{L-1} < p < 2^{L}$ for $512 \leq L \leq 1024$ and L a multiple of 64

2. q = a prime divisor of p - 1, where $2^{159} < q < 2^{160}$

3. $g = h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < p - 1$ such that $h^{(p-1)/q} \bmod p > 1$ (g has order q mod p)

4. x = a randomly or pseudorandomly generated integer with $0 < x < q$

5. $y = g^{x} \bmod p$

6. k = a randomly or pseudorandomly generated integer with $0 < k < q$

The integers p, q, and g can be public and can be common to a group of users. A user's private and public keys are x and y, respectively. They are normally fixed for a period of time. Parameters x and k are used for signature generation only, and must be kept secret. Parameter k must be regenerated for each signature.

Parameters p and q shall be generated as specified in Appendix 2, or using other FIPS approved security methods. Parameters x and k shall be generated as specified in Appendix 3, or using other FIPS approved security methods.

5. DSA SIGNATURE GENERATION

The signature of a message M is the pair of numbers r and s computed according to the equations below:

$r = (g^k \bmod p) \bmod q$ and
$s = (k^{-1}(\text{SHA-1}(M) + xr)) \bmod q$.

In the above, k^{-1} is the multiplicative inverse of k, mod q; i.e., $(k^{-1} k) \bmod q = 1$ and $0 < k^{-1} < q$. The value of SHA-1(M) is a 160-bit string output by the Secure Hash Algorithm specified in FIPS 180-1. For use in computing s, this string must be converted to an integer. The conversion rule is given in Appendix 2.2.

As an option, one may wish to check if r = 0 or s = 0. If either r = 0 or s = 0, a new value of k should be generated and the signature should be recalculated (it is extremely unlikely that r = 0 or s = 0 if signatures are generated properly).

The signature is transmitted along with the message to the verifier.

6. DSA SIGNATURE VERIFICATION

Prior to verifying the signature in a signed message, p, q and g plus the sender's public key and identity are made available to the verifier in an authenticated manner.

Let M′, r′, and s′ be the received versions of M, r, and s, respectively, and let y be the public key of the signatory. To verify the signature, the verifier first checks to see that $0 < r' < q$ and $0 < s' < q$; if either condition is violated the signature shall be rejected. If these two conditions are satisfied, the verifier computes

$w = (s')^{-1} \bmod q$

$u1 = ((\text{SHA-1}(M'))w) \bmod q$

$u2 = ((r')w) \bmod q$

$v = (((g)^{u1} (y)^{u2}) \bmod p) \bmod q$.

If v = r′, then the signature is verified and the verifier can have high confidence that the received message was sent by the party holding the secret key x corresponding to y. For a proof that v = r′ when M′ = M, r′ = r, and s′ = s, see Appendix 1.

If v does not equal r′, then the message may have been modified, the message may have been incorrectly signed by the signatory, or the message may have been signed by an impostor. The message should be considered invalid.

7. RSA DIGITAL SIGNATURE ALGORITHM

The RSA ds algorithm is a FIPS approved cryptographic algorithm for digital signature generation and verification. This is described in ANSI X9.31.

8. ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA)

The ECDSA ds algorithm is a FIPS approved cryptographic algorithm for digital signature generation and verification. ECDSA is the elliptic curve analogue of the DSA. ECDSA is described in ANSI X9.62. The recommended elliptic curves for Federal Government use are included in Appendix 6.

APPENDIX 1. A PROOF THAT v = r′ IN THE DSA

This appendix is for informational purposes only and is not required to meet the standard.

The purpose of this appendix is to show that in the DSA, if M′ = M, r′ = r and s′ = s in the signature verification then v = r′. We need the following easy result.

LEMMA. Let p and q be primes so that q divides p - 1, h a positive integer less than p, and $g = h^{(p-1)/q} \bmod p$. Then $g^q \bmod p = 1$, and if m mod q = n mod q, then $g^m \bmod p = g^n \bmod p$.

Proof: We have

$$g^q \bmod p = (h^{(p-1)/q} \bmod p)^q \bmod p$$

$$= h^{(p-1)} \bmod p$$

$$= 1$$

by Fermat's Little Theorem. Now let m mod q = n mod q, i.e., m = n + kq for some integer k. Then

$$g^m \bmod p = g^{n+kq} \bmod p$$

$$= (g^n g^{kq}) \bmod p$$

$$= ((g^n \bmod p)(g^q \bmod p)^k) \bmod p$$

$$= g^n \bmod p$$

since $g^q \bmod p = 1$. ■

We are now ready to prove the main result.

THEOREM. If M′ = M, r′ = r, and s′ = s in the signature verification, then v = r′.

Proof: We have

$$w = (s')^{-1} \bmod q = s^{-1} \bmod q$$

$$u1 = ((\text{SHA-1}(M'))w) \bmod q = ((\text{SHA-1}(M))w) \bmod q$$

$$u2 = ((r')w) \bmod q = (rw) \bmod q.$$

Now $y = g^x \bmod p$, so that by the lemma,

$$v = ((g^{u1}\, y^{u2}) \bmod p) \bmod q$$

$$= ((g\text{SHA-1}^{(M)w}\, y^{rw}) \bmod p) \bmod q$$

$$= ((g\text{SHA-1}^{(M)w}\, g^{xrw}) \bmod p) \bmod q$$

$$= ((g^{(}\text{SHA-1}^{(M)+xr)w}) \bmod p) \bmod q.$$

Also

$$s = (k^{-1}(\text{SHA-1}(M) + xr)) \bmod q.$$

Hence

$$w = (k(\text{SHA-1}(M) + xr)^{-1}) \bmod q$$

$$(\text{SHA-1}(M) + xr)w \bmod q = k \bmod q.$$

Thus by the lemma,

$$v = (g^k \bmod p) \bmod q$$

$$= r$$

$$= r'. \blacksquare$$

APPENDIX 2. GENERATION OF PRIMES FOR THE DSA

This appendix includes algorithms for generating the primes p and q used in the DSA. These algorithms require a random number generator (see Appendix 3), and an efficient modular exponentiation algorithm. Generation of p and q shall be performed as specified in this appendix, or using other FIPS approved security methods.

2.1. A PROBABILISTIC PRIMALITY TEST

In order to generate the primes p and q, a primality test is required.

There are several fast probabilistic algorithms available. The following algorithm is a simplified version of a procedure due to M.O. Rabin, based in part on ideas of Gary L. Miller. [See Knuth, The Art of Computer Programming, Vol. 2, Addison-Wesley, 1981, Algorithm P, page 379.] If this algorithm is iterated n times, it will produce a false prime with probability no greater than $1/4^n$. Therefore, $n \geq 50$ will give an acceptable probability of error. To test whether an integer is prime:

Step 1. Set $i = 1$ and $n \geq 50$.

Step 2. Set w = the integer to be tested, $w = 1 + 2^a m$, where m is odd and 2^a is the largest power of 2 dividing w - 1.

Step 3. Generate a random integer b in the range $1 < b < w$.

Step 4. Set $j = 0$ and $z = b^m \bmod w$.

Step 5. If $j = 0$ and $z = 1$, or if $z = w - 1$, go to step 9.

Step 6. If $j > 0$ and $z = 1$, go to step 8.

Step 7. $j = j + 1$. If $j < a$, set $z = z^2 \bmod w$ and go to step 5.

Step 8. w is not prime. Stop.

Step 9. If $i < n$, set $i = i + 1$ and go to step 3. Otherwise, w is probably prime.

2.2. GENERATION OF PRIMES

The DSA requires two primes, p and q, satisfying the following three conditions:

a. $2^{159} < q < 2^{160}$

b. $2^{L-1} < p < 2^L$ for a specified L, where $L = 512 + 64j$ for some $0 \leq j \leq 8$

c. q divides p - 1.

This prime generation scheme starts by using the SHA-1 and a user supplied SEED to construct a prime, q, in the range $2^{159} < q < 2^{160}$. Once this is accomplished, the same SEED value is used to construct an X in the range $2^{L-1} < X < 2^{L}$. The prime, p, is then formed by rounding X to a number congruent to 1 mod 2q as described below.

An integer x in the range $0 \le x < 2^{g}$ may be converted to a g-long sequence of bits by using its binary expansion as shown below:

$x = x_1*2^{g-1} + x_2*2^{g-2} + ... + x_{g-1}*2 + x_g \rightarrow \{ x_1,...,x_g \}$.

Conversely, a g-long sequence of bits $\{ x_1,...,x_g \}$ is converted to an integer by the rule

$\{ x_1,...,x_g \} \rightarrow x_1*2^{g-1} + x_2*2^{g-2} + ... + x_{g-1}*2 + x_g$.

Note that the first bit of a sequence corresponds to the most significant bit of the corresponding integer and the last bit to the least significant bit.

Let $L - 1 = n*160 + b$, where both b and n are integers and $0 \le b < 160$.

Step 1. Choose an arbitrary sequence of at least 160 bits and call it SEED. Let g be the length of SEED in bits.

Step 2. Compute

$$U = \text{SHA-1}[\text{SEED}] \text{ XOR SHA-1}[(\text{SEED}+1) \bmod 2^{g}].$$

Step 3. Form q from U by setting the most significant bit (the 2^{159} bit) and the least significant bit to 1. In terms of boolean operations, q = U OR 2^{159} OR 1. Note that $2^{159} < q < 2^{160}$.

Step 4. Use a robust primality testing algorithm to test whether q is prime[1].

Step 5. If q is not prime, go to step 1.

Step 6. Let counter = 0 and offset = 2.

Step 7. For k = 0,...,n let

$$V_k = \text{SHA-1}[(\text{SEED} + \text{offset} + k) \bmod 2^{g}].$$

[1]A robust primality test is one where the probability of a non-prime number passing the test is at most 2^{-80}.

Step 8. Let W be the integer

$$W = V_0 + V_1*2^{160} + ... + V_{n-1}*2^{(n-1)*160} + (V_n \bmod 2^b) * 2^{n*160}$$

and let $X = W + 2^{L-1}$. Note that $0 \le W < 2^{L-1}$ and hence $2^{L-1} \le X < 2^L$.

Step 9. Let $c = X \bmod 2q$ and set $p = X - (c - 1)$. Note that p is congruent to 1 mod 2q.

Step 10. If $p < 2^{L-1}$, then go to step 13.

Step 11. Perform a robust primality test on p.

Step 12. If p passes the test performed in step 11, go to step 15.

Step 13. Let counter = counter + 1 and offset = offset + n + 1.

Step 14. If counter $\ge 2^{12} = 4096$ go to step 1, otherwise (i.e. if counter < 4096) go to step 7.

Step 15. Save the value of SEED and the value of counter for use in certifying the proper generation of p and q.

APPENDIX 3. RANDOM NUMBER GENERATION FOR THE DSA

Any implementation of the DSA requires the ability to generate random or pseudorandom integers. Such numbers are used to derive a user's private key, x, and a user's per message secret number, k. These randomly or pseudorandomly generated integers are selected to be between 0 and the 160-bit prime q (as specified in the standard). They shall be generated by the techniques given in this appendix, or using other FIPS approved security methods.

One FIPS approved pseudorandom integer generator is supplied in Appendix C of ANSI X9.17, "Financial Institution Key Management (Wholesale)."

Other pseudorandom integer generators are given in this appendix. These permit generation of pseudorandom values of x and k for use in the DSA. The algorithm in section 3.1 may be used to generate values for x. An algorithm for k and r is given in section 3.2. The latter algorithm allows most of the signature computation to be precomputed without knowledge of the message to be signed.

The algorithms employ a one-way function G(t,c), where t is 160 bits, c is b bits ($160 \leq b \leq 512$) and G(t,c) is 160 bits. One way to construct G is via the Secure Hash Algorithm (SHA-1), as defined in the Secure Hash Standard (SHS). The 160-bit message digest output of the SHA-1 algorithm when message M is input is denoted by SHA-1(M). A second method for constructing G is to use the Data Encryption Standard (DES). The construction of G by these techniques is discussed in sections 3.3 and 3.4 of this appendix.

In the algorithms in sections 3.1 and 3.2, a secret b-bit seed-key is used. The algorithm in section 3.1 optionally allows the use of a user provided input. If G is constructed via the SHA-1 as defined in section 3.3, then b is between 160 and 512. If DES is used to construct G as defined in section 3.4, then b is equal to 160.

3.1. ALGORITHM FOR COMPUTING m VALUES OF x

Let x be the signer's private key. The following may be used to generate m values of x:

Step 1. Choose a new, secret value for the seed-key, XKEY.

Step 2. In hexadecimal notation let

t = 67452301 EFCDAB89 98BADCFE 10325476 C3D2E1F0.

This is the initial value for $H_0 \| H_1 \| H_2 \| H_3 \| H_4$ in the SHS.

Step 3. For j = 0 to m - 1 do

a. $XSEED_j$ = optional user input.
b. XVAL = (XKEY + $XSEED_j$) mod 2^b.

c. x_j = G(t,XVAL) mod q.

d. XKEY = (1 + XKEY + x_j) mod 2^b.

3.2. ALGORITHM FOR PRECOMPUTING ONE OR MORE k AND r VALUES

This algorithm can be used to precompute k, k^{-1}, and r for m messages at a time. Note that implementation of the DSA with precomputation may be covered by U.S. and foreign patents.

Algorithm:

Step 1. Choose a secret initial value for the seed-key, KKEY.

Step 2. In hexadecimal notation let

t = EFCDAB89 98BADCFE 10325476 C3D2E1F0 67452301.

This is a cyclic shift of the initial value for $H_0 \| H_1 \| H_2 \| H_3 \| H_4$ in the SHS.

Step 3. For j = 0 to m - 1 do

a. k = G(t,KKEY) mod q.

b. Compute $k_j^{-1} = k^{-1}$ mod q.

c. Compute $r_j = (g^k \bmod p) \bmod q$.

d. KKEY = (1 + KKEY + k) mod 2^b.

Step 4. Suppose M_0 , ... , M_{m-1} are the next m messages. For j = 0 to m - 1 do

a. Let h = SHA-1(M_j).

b. Let $s_j = (k_j^{-1}(h + xr_j)) \bmod q$.

c. The signature for M_j is (r_j,s_j).

Step 5. Let t = h.

Step 6. Go to step 3.

Step 3 permits precomputation of the quantities needed to sign the next m messages. Step 4 can begin whenever the first of these m messages is ready. The execution of step 4 can be suspended whenever the next of the m messages is not ready. As soon as steps 4 and 5 have completed, step 3 can be executed, and the results saved until the first member of the next group of m messages is ready.

In addition to space for KKEY, two arrays of length m are needed to store r_0 , ... r_{m-1} and k_0^{-1}, ... , k_{m-1}^{-1} when they are computed in step 3. Storage for s_0 , ... , s_{m-1} is only needed if the signatures for a group of messages are stored; otherwise s_j in step 4 can be replaced by s and a single space allocated.

3.3. CONSTRUCTING THE FUNCTION G FROM THE SHA-1

G(t,c) may be constructed using steps (a) - (e) in section 7 of the Specifications for the Secure Hash Standard. Before executing these steps, $\{H_j\}$ and M_1 must be initialized as follows:

i. Initialize the $\{H_j\}$ by dividing the 160 bit value t into five 32-bit segments as follows:

$t = t_0 \ || \ t_1 \ || \ t_2 \ || \ t_3 \ || \ t_4$

Then $H_j = t_j$ for j = 0 through 4.

ii. There will be only one message block, M_1, which is initialized as follows:

$M_1 = c \ || \ 0^{512-b}$

(The first b bits of M_1 contain c, and the remaining (512-b) bits are set to zero).

Then steps (a) through (e) of section 7 are executed, and G(t,c) is the 160 bit string represented by the five words:

$H_0 \ || \ H_1 \ || \ H_2 \ || \ H_3 \ || \ H_4$

at the end of step (e).

3.4. CONSTRUCTING THE FUNCTION G FROM THE DES

Let a XOR b denote the bitwise exclusive-or of bit strings a and b. Suppose a1, a2, b1, b2 are 32-bit strings. Let b1' be the 24 least significant bits of b1. Let K = b1' || b2 and A = a1 || a2. Define

$DES_{b1,b2}(a1,a2) = DES_K(A)$

In the above, $DES_K(A)$ represents ordinary DES encryption of the 64-bit block A using the 56-bit

key K. Now suppose t and c are each 160 bits. To compute G(t,c):

Step 1. Write

$t = t_1 \| t_2 \| t_3 \| t_4 \| t_5$

$c = c_1 \| c_2 \| c_3 \| c_4 \| c_5$

In the above, each t_i and c_i is 32 bits.

Step 2. For i = 1 to 5 do

$x_i = t_i$ XOR c_i

Step 3. For i = 1 to 5 do

$b1 = c_{((i+3) \bmod 5) + 1}$

$b2 = c_{((i+2) \bmod 5) + 1}$

$a1 = x_i$

$a2 = x_{(i \bmod 5) + 1}$ XOR $x_{((i+3) \bmod 5) + 1}$

$y_{i,1} \| y_{i,2} = DES_{b1,b2}(a1,a2)$ ($y_{i,1}$, $y_{i,2}$ = 32 bits)

Step 4. For i = 1 to 5 do

$z_i = y_{i,1}$ XOR $y_{((i+1) \bmod 5)+1,2}$ XOR $y_{((i+2) \bmod 5)+1,1}$

Step 5. Let

$G(t,c) = z_1 \| z_2 \| z_3 \| z_4 \| z_5$

APPENDIX 4. GENERATION OF OTHER QUANTITIES FOR THE DSA

This appendix is for informational purposes only and is not required to meet the standard.

The algorithms given in this appendix may be used to generate the quantities g, k^{-1}, and s^{-1} used in the DSA.

To generate g:

Step 1. Generate p and q as specified in Appendix 2.

Step 2. Let e = (p - 1)/q.

Step 3. Set h = any integer, where $1 < h < p - 1$ and h differs from any value previously tried.

Step 4. Set $g = h^e \bmod p$.

Step 5. If g = 1, go to step 3.

To compute the multiplicative inverse $n^{-1} \bmod q$ for n with $0 < n < q$, where $0 < n^{-1} < q$:

Step 1. Set i = q, h = n, v = 0, and d = 1.

Step 2. Let t = i DIV h, where DIV is defined as integer division.

Step 3. Set x = h.

Step 4. Set h = i - tx.

Step 5. Set i = x.

Step 6. Set x = d.

Step 7. Set d = v - tx.

Step 8. Set v = x.

Step 9. If h > 0, go to step 2.

Step 10. Let $n^{-1} = v \bmod q$.

Note that in step 10, v may be negative. The v mod q operation should yield a value between 1 and q - 1 inclusive.

APPENDIX 5. EXAMPLE OF THE DSA

This appendix is for informational purposes only and is not required to meet the standard.
Let L = 512 (size of p). The values in this example are expressed in hexadecimal notation. The p and q given here were generated by the prime generation standard described in appendix 2 using the 160-bit SEED:

```
d5014e4b 60ef2ba8 b6211b40 62ba3224 e0427dd3
```

With this SEED, the algorithm found p and q when the counter was at 105. x was generated by the algorithm described in appendix 3, section 3.1, using the SHA-1 to construct G (as in appendix 3, section 3.3) and a 160-bit XKEY:

XKEY =

```
bd029bbe 7f51960b cf9edb2b 61f06f0f eb5a38b6
```

t =

```
67452301 EFCDAB89 98BADCFE 10325476 C3D2E1F0
```

x = G(t,XKEY) mod q

k was generated by the algorithm described in appendix 3, section 3.2, using the SHA-1 to construct G (as in appendix 3, section 3.3) and a 160-bit KKEY:

KKEY =

```
687a66d9 0648f993 867e121f 4ddf9ddb 01205584
```

t =

```
EFCDAB89 98BADCFE 10325476 C3D2E1F0 67452301
```

k = G(t,KKEY) mod q

Finally:

h = 2

p =

```
8df2a494 492276aa 3d25759b b06869cb eac0d83a fb8d0cf7
cbb8324f 0d7882e5 d0762fc5 b7210eaf c2e9adac 32ab7aac
```

```
        49693dfb f83724c2 ec0736ee 31c80291
```

q =

```
        c773218c 737ec8ee 993b4f2d ed30f48e dace915f
```

g =

```
        626d0278 39ea0a13 413163a5 5b4cb500 299d5522 956cefcb
        3bff10f3 99ce2c2e 71cb9de5 fa24babf 58e5b795 21925c9c
        c42e9f6f 464b088c c572af53 e6d78802
```

x =

```
        2070b322 3dba372f de1c0ffc 7b2e3b49 8b260614
```

k =

```
        358dad57 1462710f 50e254cf 1a376b2b deaadfbf
```

k^{-1} =

```
        0d516729 8202e49b 4116ac10 4fc3f415 ae52f917
```

M = ASCII form of "abc" (See FIPS PUB 180-1, Appendix A)

(SHA-1)(M) =

```
        a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d
```

y =

```
        19131871 d75b1612 a819f29d 78d1b0d7 346f7aa7 7bb62a85
        9bfd6c56 75da9d21 2d3a36ef 1672ef66 0b8c7c25 5cc0ec74
        858fba33 f44c0669 9630a76b 030ee333
```

r =

```
        8bac1ab6 6410435c b7181f95 b16ab97c 92b341c0
```

s =

```
        41e2345f 1f56df24 58f426d1 55b4ba2d b6dcd8c8
```

w =

```
        9df4ece5 826be95f ed406d41 b43edc0b 1c18841b
```

u1 =

```
bf655bd0 46f0b35e c791b004 804afcbb 8ef7d69d
```

u2 =

```
821a9263 12e97ade abcc8d08 2b527897 8a2df4b0
```

$g^{u1} \bmod p$ =

```
51b1bf86 7888e5f3 af6fb476 9dd016bc fe667a65 aafc2753
9063bd3d 2b138b4c e02cc0c0 2ec62bb6 7306c63e 4db95bbf
6f96662a 1987a21b e4ec1071 010b6069
```

$y^{u2} \bmod p$ =

```
8b510071 2957e950 50d6b8fd 376a668e 4b0d633c 1e46e665
5c611a72 e2b28483 be52c74d 4b30de61 a668966e dc307a67
c19441f4 22bf3c34 08aeba1f 0a4dbec7
```

v =

```
8bac1ab6 6410435c b7181f95 b16ab97c 92b341c0
```

APPENDIX 6. RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE
July 1999

This collection of elliptic curves is recommended for Federal government use and contains choices of private key length and underlying fields.

1. Parameter Choices

1.1 Choice of Key Lengths

The principal parameters for elliptic curve cryptography are the elliptic curve E and a designated point G on E called the *base point.* The base point has order r, a large prime. The number of points on the curve is $n = fr$ for some integer f (the *cofactor*) not divisible by r. For efficiency reasons, it is desirable to take the cofactor to be as small as possible.

All of the curves given below have cofactors 1, 2, or 4. As a result, the private and public keys are approximately the same length. Each length is chosen to correspond to the cryptovariable length of a common symmetric cryptologic. In each case, the private key length is, at least, approximately twice the symmetric cryptovariable length.

1.2 Choice of Underlying Fields

For each cryptovariable length, there are given two kinds of fields.

- A *prime field* is the field $GF(p)$ which contains a prime number p of elements. The elements of this field are the integers modulo p, and the

field arithmetic is implemented in terms of the arithmetic of integers modulo p.

- A *binary field* is the field $GF(2^m)$ which contains 2^m elements for some m (called the *degree* of the field). The elements of this field are the bit strings of length m, and the field arithmetic is implemented in terms of operations on the bits.

The following table gives the sizes of the various underlying fields. By $||p||$ is meant the length of the binary expansion of the integer p.

Symmetric CV Length	Example Algorithm	Prime Field	Binary Field
80	SKIPJACK	$\|\|p\|\| = 192$	$m = 163$
112	Triple-DES	$\|\|p\|\| = 224$	$m = 233$
128	AES Small	$\|\|p\|\| = 256$	$m = 283$
192	AES Medium	$\|\|p\|\| = 384$	$m = 409$
256	AES Large	$\|\|p\|\| = 521$	$m = 571$

1.3 Choice of Basis

To describe the arithmetic of a binary field, it is first necessary to specify how a bit string is to be interpreted. This is referred to as choosing a *basis* for the field. There are two common types of bases: a *polynomial basis* and a *normal basis*.

- A polynomial basis is specified by an irreducible polynomial modulo 2, called the *field polynomial*. The bit string $(a_{m-1} \ldots a_2\ a_1\ a_0)$ is taken to represent the polynomial

  $$a_{m-1}t^{m-1} + \ldots + a_2 t^2 + a_1 t + a_0$$

  over $GF(2)$. The field arithmetic is implemented as polynomial arithmetic modulo $p(t)$, where $p(t)$ is the field polynomial.
- A normal basis is specified by an element q of a particular kind. The bit string $(a_0\ a_1\ a_2\ \ldots a_{m-1})$ is taken to represent the element

  $$a_0 q + a_1 q^2 + a_2 q^{2^2} + a_{m-1} q^{2^{m-1}}.$$

  Normal basis field arithmetic is not easy to describe or efficient to implement in general, but is for a special class called *Type T low-complexity* normal bases. For a given field degree m, the choice of T specifies the basis and the field arithmetic (see Appendix 6.2).

There are many polynomial bases and normal bases from which to choose. The following procedures are commonly used to select a basis representation.

- *Polynomial Basis*: If an irreducible *trinomial* $t^m + t^k + 1$ exists over $GF(2)$, then the field polynomial $p(t)$ is chosen to be the irreducible trinomial with the lowest-degree middle term t^k. If no irreducible trinomial exists, then one selects instead a *pentanomial* $t^m + t^a + t^b + t^c + 1$. The particular pentanomial chosen has the following properties: the second term t^a has the lowest degree m; the third term t^b has the lowest degree among all irreducible pentanomials of degree m and second term t^a; and the fourth term t^c has the lowest degree among all irreducible pentanomials of degree m, second term t^a, and third term t^b.

- *Normal Basis*: Choose the Type T low-complexity normal basis with the smallest *T*.

For each binary field, the parameters are given for the above basis representations.

1.4 Choice of Curves

Two kinds of curves are given:

- *Pseudo-random* curves are those whose coefficients are generated from the output of a seeded cryptographic hash. If the seed value is given along with the coefficients, it can be verified easily that the coefficients were indeed generated by that method.
- *Special curves* whose coefficients and underlying field have been selected to optimize the efficiency of the elliptic curve operations.

For each size, the following curves are given:

→ A pseudo-random curve over $GF(p)$.

→ A pseudo-random curve over $GF(2^m)$.

→ A special curve over $GF(2^m)$ called a *Koblitz curve* or *anomalous binary curve.*

The pseudo-random curves are generated via the SHA-1 based method given in the ANSI X9.62 and IEEE P1363 standards. (The generation and verification processes are given in Appendices 6-4 through 6-7.)

1.5 Choice of Base Points

Any point of order r can serve as the base point. Each curve is supplied with a sample base point $G = (G_x , G_y)$. Users may want to generate their own base points to ensure cryptographic separation of networks.

2. Curves over Prime Fields

For each prime p, a pseudo-random curve

$$E : \; y^2 \equiv x^3 - 3x + b \; (mod \; p)$$

of prime order r is listed1. (Thus, for these curves, the cofactor is always $f = 1$.) The following parameters are given:

- The prime modulus p
- The order r
- the 160-bit input seed s to SHA-1 based algorithm
- The output c of the SHA-1 based algorithm
- The coefficient b (satisfying $b^2 \; c \equiv -27 \; (mod \; p)$)
- The base point x coordinate G_x
- The base point y coordinate G_y

The integers p and r are given in decimal form; bit strings and field elements are given in hex.

1 The selection $a \equiv -3$ for the coefficient of x was made for reasons of efficiency; see IEEE P1363.

Curve P-192

p = 6277101735386680763835789423207666416083908700390324961279

r = 6277101735386680763835789423176059013767194773182842284081

s = 3045ae6f c8422f64 ed579528 d38120ea e12196d5

c = 3099d2bb bfcb2538 542dcd5f b078b6ef 5f3d6fe2 c745de65

b = 64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1

G_x = 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012

G_y = 07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811

Curve P-224

p = 26959946667150639794667015087019630673557916\
260026308143510066298881

r = 26959946667150639794667015087019625940457807\
7144243917216827223680 61

s = bd713447 99d5c7fc dc45b59f a3b9ab8f 6a948bc5

c = 5b056c7e 11dd68f4\
0469ee7f 3c7a7d74 f7d12111 6506d031 218291fb

b = b4050a85 0c04b3ab\
f5413256 5044b0b7 d7bfd8ba 270b3943 2355ffb4

G_x = b70e0cbd 6bb4bf7f\
321390b9 4a03c1d3 56c21122 343280d6 115c1d21

G_y = bd376388 b5f723fb\
4c22dfe6 cd4375a0 5a074764 44d58199 85007e34

Curve P-256

p = 115792089210356248762697446949407573530086143415290314195533631308867097853951

r = 115792089210356248762697446949407573529996955224135760342422259061068512044369

s = c49d3608 86e70493 6a6678e1 139d26b7 819f7e90

c = 7efba166 2985be94 03cb055c 75d4f7e0 ce8d84a9 c5114abc af317768 0104fa0d

b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b

G_x = 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296

G_y = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5

Curve P-384

p = 39402006196394479212279040100143613805079739\
27046544666794829340424572177149687032904726\
608825893800186160697311231 9

r = 39402006196394479212279040100143613805079739\
27046544666794690527962765939911326356939895\
630815229491355443365394 2643

s = a335926a a319a27a 1d00896a 6773a482 7acdac73

c = 79d1e655 f868f02f\
ff48dcde e14151dd b80643c1 406d0ca1 0dfe6fc5\
2009540a 495e8042 ea5f744f 6e184667 cc722483

b = b3312fa7 e23ee7e4\
988e056b e3f82d19 181d9c6e fe814112 0314088f\
5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

G_x = aa87ca22 be8b0537\
8eb1c71e f320ad74 6e1d3b62 8ba79b98 59f741e0\
82542a38 5502f25d bf55296c 3a545e38 72760ab7

G_y = 3617de4a 96262c6f\
5d9e98bf 9292dc29 f8f41dbd 289a147c e9da3113\
b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f

Curve P-521

p = 6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559\
640661454554977296311391480858037121987999716643812574028291115057151

r = 6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057\
746333217197532963996371363321113864768612440380340372808892707005449

s = d09e8800 291cb853 96cc6717 393284aa a0da64ba

c = 0b4 8bfa5f42 0a349495 39d2bdfc 264eeeeb 077688e4 4fbf0ad8 f6d0edb3 7bd6b533 28100051 8e19f1b9 ffbe0fe9 ed8a3c22 00b8f875 e523868c 70c1e5bf 55bad637

b = 051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b 99b315f3 b8b48991 8ef109e1 56193951 ec7e937b 1652c0bd 3bb1bf07 3573df88 3d2c34f1 ef451fd4 6b503f00

G_x = c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139 053fb521 f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127 a2ffa8de 3348b3c1 856a429b f97e7e31 c2e5bd66

G_y = 118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449 579b4468 17afbd17 273e662c 97ee7299 5ef42640 c550b901

3fad0761 353c7086 a272c240 88be9476 9fd16650

3. Curves over Binary Fields

For each field degree *m*, a pseudo-random curve is given, along with a Koblitz curve. The pseudo-random curve has the form

$$E: y^2 + xy = x^3 + x^2 + b,$$

and the Koblitz curve has the form

$$E_a: y^2 + xy = x^3 + ax^2 + 1$$

where $a = 0$ or 1.

For each pseudorandom curve, the cofactor is $f = 2$. The cofactor of each Koblitz curve is $f = 2$ if $a = 1$ and $f = 4$ if $a = 0$.

The coefficients of the pseudo-random curves, and the coordinates of the base points of both kinds of curves, are given in terms of both the polynomial and normal basis representations discussed in 1.3.

For each *m*, the following parameters are given:

Field Representation:

- The normal basis type *T*
- The field polynomial (a trinomial or pentanomial)

Koblitz Curve:

- The coefficient *a*
- The base point order *r*
- The base point *x* coordinate G_x
- The base point *y* coordinate G_y

Pseudo-random curve:

- The base point order *r*

Pseudo-random curve (Polynomial Basis representation):

- The coefficient *b*
- The base point *x* coordinate G_x
- The base point *y* coordinate G_y

Pseudo-random curve (Normal Basis representation):

- The 160-bit input seed *s* to the SHA-1 based algorithm
- The coefficient *b* (i.e., the output of the SHA-1 based algorithm)
- The base point *x* coordinate G_x
- The base point *y* coordinate G_y

Integers (such as *T*, *m*, and *r*) are given in decimal form; bit strings and field elements are given in hex.

Degree 163 Binary Field

$T = 4$

$p(t) = t^{163} + t^7 + t^6 + t^3 + 1$

Curve K-163

$a = 1$

$r = 5846006549323611672814741753598448348329118574063$

Polynomial Basis:

$G_x =$ 2 fe13c053 7bbc11ac aa07d793 de4e6d5e 5c94eee8

$G_y =$ 2 89070fb0 5d38ff58 321f2e80 0536d538 ccdaa3d9

Normal Basis:

$G_x =$ 0 5679b353 caa46825 fea2d371 3ba450da 0c2a4541

$G_y =$ 2 35b7c671 00506899 06bac3d9 dec76a83 5591edb2

Curve B-163

$r = 5846006549323611672814742442876390689256843201587$

Polynomial Basis:

$b =$ 2 0a601907 b8c953ca 1481eb10 512f7874 4a3205fd

$G_x =$ 3 f0eba162 86a2d57e a0991168 d4994637 e8343e36

$G_y =$ 0 d51fbc6c 71a0094f a2cdd545 b11c5c0c 797324f1

Normal Basis:

$s =$ 85e25bfe 5c86226c db12016f 7553f9d0 e693a268

$b =$ 6 645f3cac f1638e13 9c6cd13e f61734fb c9e3d9fb

$G_x =$ 0 311103c1 7167564a ce77ccb0 9c681f88 6ba54ee8

G_y = 3 33ac13c6 447f2e67 613bf700 9daf98c8 7bb50c7f

Degree 233 Binary Field

$T = 2$

$p(t) = t^{233} + t^{74} + 1$

Curve K-233

$a = 0$

r =3450873173395281893717377931138512760570940988862252126328087024741343

Polynomial Basis:

G_x = 172 32ba853a 7e731af1 29f22ff4 149563a4 19c26bf5 0a4c9d6e efad6126

G_y = 1db 537dece8 19b7f70f 555a67c4 27a8cd9b f18aeb9b 56e0c110 56fae6a3

Normal Basis:

G_x = 0fd e76d9dcd 26e643ac 26f1aa90 1aa12978 4b71fc07 22b2d056 14d650b3

G_y = 064 3e317633 155c9e04 47ba8020 a3c43177 450ee036 d6335014 34cac978

Curve B-233

r = 690174634679056378743475586227702555583981273734501355379383634485463

Polynomial Basis:

b = 066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad

G_x = 0fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36 f8f8eb73 71fd558b

G_y = 100 6a08a419 03350678 e58528be bf8a0bef f867a7ca 36716f7e 01f81052

Normal Basis:

s = 74d59ff0 7f6b413d 0ea14b34 4b20a2db 049b50c3

b = 1a0 03e0962d 4f9a8e40 7c904a95 38163adb 82521260 0c7752ad 52233279

G_x = 18b 863524b3 cdfefb94 f2784e0b 116faac5 4404bc91 62a363ba b84a14c5

G_y = 049 25df77bd 8b8ff1a5 ff519417 822bfedf 2bbd7526 44292c98 c7af6e02

Degree 283 Binary Field

$T = 6$

$p(t) = t^{283} + t^{12} + t^7 + t^5 + 1$

Curve K-283

$a = 0$

$r = 3885337784451458141838923813647037813284811733793061\backslash$

$3242958749975298158297044226038 73$

Polynomial Basis:

$G_x =$ 503213f 78ca4488 3f1a3b81 62f188e5 53cd265f 23c1567a 16876913 b0c2ac24 58492836

$G_y =$ 1ccda38 0f1c9e31 8d90f95d 07e5426f e87e45c0 e8184698 e4596236 4e341161 77dd2259

Normal Basis:

$G_x =$ 3ab9593 f8db09fc 188f1d7c 4ac9fcc3 e57fcd3b db15024b 212c7022 9de5fcd9 2eb0ea60

$G_y =$ 2118c47 55e7345c d8f603ef 93b98b10 6fe8854f feb9a3b3 04634cc8 3a0e759f 0c2686b1

Curve B-283

r = 77706755689029162836778476272940756265696259243769048\
8910919652677004427778737869287 1

Polynomial Basis:

b = 27b680a c8b8596d a5a4af8a 19a0303f
ca97fd76 45309fa2 a581485a f6263e31 3b79a2f5

G_x = 5f93925 8db7dd90 e1934f8c 70b0dfec
2eed25b8 557eac9c 80e2e198 f8cdbecd 86b12053

G_y = 3676854 fe24141c b98fe6d4 b20d02b4
516ff702 350eddb0 826779c8 13f0df45 be8112f4

Normal Basis:

s = 77e2b073 70eb0f83 2a6dd5b6 2dfc88cd 06bb84be

b = 157261b 894739fb 5a13503f 55f0b3f1
0c560116 66331022 01138cc1 80c0206b dafbc951

G_x = 749468e 464ee468 634b21f7 f61cb700
701817e6 bc36a236 4cb8906e 940948ea a463c35d

G_y = 62968bd 3b489ac5 c9b859da 68475c31 5bafcdc4
ccd0dc90 5b70f624 46f49c05 2f49c08c

Degree 409 Binary Field

$T = 4$

$p(t) = t^{409} + t^{87} + 1$

Curve K-409

$a = 0$

$r = 3305279843951242994759576540163855199142023414821406096423243950228807112892491910506732584577774580140963\66590617731358671$

Polynomial Basis:

$G_x =$ 060f05f 658f49c1 ad3ab189 0f718421 0efd0987 e307c84c 27accfb8 f9f67cc2 c460189e b5aaaa62 ee222eb1 b35540cf e9023746

$G_y =$ 1e36905 0b7c4e42 acba1dac bf04299c 3460782f 918ea427 e6325165 e9ea10e3 da5f6c42 e9c55215 aa9ca27a 5863ec48 d8e0286b

Normal Basis:

$G_x =$ 1b559c7 cba2422e 3affe133 43e808b5 5e012d72 6ca0b7e6 a63aeafb c1e3a98e 10ca0fcf 98350c3b 7f89a975 4a8e1dc0 713cec4a

$G_y =$ 16d8c42 052f07e7 713e7490 eff318ba 1abd6fef 8a5433c8 94b24f5c 817aeb79 852496fb ee803a47 bc8a2038 78ebf1c4 99afd7d6

Curve B-409

r = 661055968790248598951915308032771039828404682964281219284648798304157774827374805208143723762179110965979867288366567526771

Polynomial Basis:

b = 021a5c2 c8ee9feb 5c4b9a75 3b7b476b 7fd6422e f1f3dd67 4761fa99 d6ac27c8 a9a197b2 72822f6c d57a55aa 4f50ae31 7b13545f

G_x = 15d4860 d088ddb3 496b0c60 64756260 441cde4a f1771d4d b01ffe5b 34e59703 dc255a86 8a118051 5603aeab 60794e54 bb7996a7

G_y = 061b1cf ab6be5f3 2bbfa783 24ed106a 7636b9c5 a7bd198d 0158aa4f 5488d08f 38514f1f df4b4f40 d2181b36 81c364ba 0273c706

Normal Basis:

s = 4099b5a4 57f9d69f 79213d09 4c4bcd4d 4262210b

b = 124d065 1c3d3772 f7f5a1fe 6e715559 e2129bdf a04d52f7 b6ac7c53 2cf0ed06 f610072d 88ad2fdc c50c6fde 72843670 f8b3742a

G_x = 0ceacbc 9f475767 d8e69f3b 5dfab398 13685262 bcacf22b 84c7b6dd 981899e7 318c96f0 761f77c6 02c016ce d7c548de 830d708f

G_y = 199d64b a8f089c6 db0e0b61 e80bb959 34afd0ca f2e8be76 d1c5e9af fc7476df

49142691 ad303902 88aa09bc c59c1573 aa3c009a

Degree 571 Binary Field

$T = 10$

$p(t) = t^{571} + t^{10} + t^5 + t^2 + 1$

Curve K-571

$a = 0$

$r =$ 1932268761508629172347675945465993672149463664853217\
4\
99328617625725759571144780212268133978522706711834706\
\
71280082535146127367497406661731192968242161709250355\
\
5733685276673

Polynomial Basis:

$G_x =$ 26eb7a8 59923fbc 82189631
f8103fe4 ac9ca297 0012d5d4 60248048 01841ca4
43709584 93b205e6 47da304d b4ceb08c bbd1ba39
494776fb 988b4717 4dca88c7 e2945283 a01c8972

$G_y =$ 349dc80 7f4fbf37 4f4aeade
3bca9531 4dd58cec 9f307a54 ffc61efc 006d8a2c
9d4979c0 ac44aea7 4fbebbb9 f772aedc b620b01a
7ba7af1b 320430c8 591984f6 01cd4c14 3ef1c7a3

Normal Basis:

$G_x =$ 04bb2db a418d0db 107adae0
03427e5d 7cc139ac b465e593 4f0bea2a b2f3622b
c29b3d5b 9aa7a1fd fd5d8be6 6057c100 8e71e484
bcd98f22 bf847642 37673674 29ef2ec5 bc3ebcf7

G_y = 44cbb57 de20788d 2c952d7b
56cf39bd 3e89b189 84bd124e 751ceff4 369dd8da
c6a59e6e 745df44d 8220ce22 aa2c852c fcbbef49
ebaa98bd 2483e331 80e04286 feaa2530 50caff60

Curve B-571

r = 3864537523017258344695351890931987344298927329706434 9\
9865723525145151914228956042453614399938941577308313 3\
8811219269444862468724628168130702345282883033324113 9\
3191105285703

Polynomial Basis:

b = 2f40e7e 2221f295 de297117 b7f3d62f 5c6a97ff cb8ceff1 cd6ba8ce 4a9a18ad 84ffabbd 8efa5933 2be7ad67 56a66e29 4afd185a 78ff12aa 520e4de7 39baca0c 7ffeff7f 2955727a

G_x = 303001d 34b85629 6c16c0d4 0d3cd775 0a93d1d2 955fa80a a5f40fc8 db7b2abd bde53950 f4c0d293 cdd711a3 5b67fb14 99ae6003 8614f139 4abfa3b4 c850d927 e1e7769c 8eec2d19

G_y = 37bf273 42da639b 6dccfffe b73d69d7 8c6c27a6 009cbbca 1980f853 3921e8a6 84423e43 bab08a57 6291af8f 461bb2a8 b3531d2f 0485c19b 16e2f151 6e23dd3c 1a4827af 1b8ac15b

Normal Basis:

s = 2aa058f7 3a0e33ab 486b0f61 0410c53a 7f132310

b = 3762d0d 47116006 179da356 88eeaccf 591a5cde a7500011 8d9608c5 9132d434 26101a1d fb377411 5f586623 f75f0000 1ce61198 3c1275fa 31f5bc9f 4be1a0f4 67f01ca8 85c74777

G_x = 0735e03 5def5925 cc33173e

G_y = b2a8ce77 67522b46 6d278b65 0a291612 7dfea9d2
d361089f 0a7a0247 a184e1c7 0d417866 e0fe0feb
0ff8f2f3 f9176418 f97d117e 624e2015 df1662a8
04a3642 0572616c df7e606f
ccadaecf c3b76dab 0eb1248d d03fbdfc 9cd3242c
4726be57 9855e812 de7ec5c5 00b4576a 24628048
b6a72d88 0062eed0 dd34b109 6d3acbb6 b01a4a97

APPENDIX 6.1: IMPLEMENTATION OF MODULAR ARITHMETIC

The prime moduli in the above examples are of a special type (called *generalized Mersenne numbers*) for which modular multiplication can be carried out more efficiently than in general. This appendix provides the rules for implementing this faster arithmetic, for each of the prime moduli appearing in the examples.

The usual way to multiply two integers (mod m) is to take the integer product and reduce it (mod m). One therefore has the following problem: given an integer A less than m^2, compute

$$B := A \bmod m.$$

In general, one must obtain B as the remainder of an integer division. If m is a generalized Mersenne number, however, then B can be expressed as a sum or difference (mod m) of a small number of terms. To compute this expression, one can evaluate the integer sum or difference and reduce the result modulo m. The latter reduction can be accomplished by adding or subtracting a few copies of m.

The prime moduli p for each of the five example curves is a generalized Mersenne number.

Curve P-192:

The modulus for this curve is $p = 2^{192} - 2^{64} - 1$. Every integer A less than p^2 can be written

$$A = A_5 \cdot 2^{320} + A_4 \cdot 2^{256} + A_3 \cdot 2^{192} + A_2 \cdot 2^{128} + A_1 \cdot 2^{64} + A_0,$$

where each A_i is a 64-bit integer. The expression for B is

$$B := T + S_1 + S_2 + S_3 \bmod p;$$

where the 192-bit terms are given by

$$T = A_2 \cdot 2^{128} + A_1 \cdot 2^{64} + A_0$$

$$S_1 = \quad A_3 \cdot 2^{64} + A_3$$

$$S_2 = A_4 \cdot 2^{128} + A_4 \cdot 2^{64}$$

$$S_3 = A_5 \cdot 2^{128} + A_5 \cdot 2^{64} + A_5.$$

Curve P-224:

The modulus for this curve is $p = 2^{224} - 2^{96} + 1$. Every integer A less than p^2 can be written

$$A = A_{13} \cdot 2^{416} + A_{12} \cdot 2^{384} + A_{11} \cdot 2^{352} + A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} + A_7 \cdot 2^{224} + A_6 \cdot 2^{192} + A_5 \cdot 2^{160} + A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0,$$

where each A_i is a 32-bit integer. As a concatenation of 32-bit words, this can be denoted by

$$A = (A_{13} \,||\, A_{12} \,||\, \ldots \,||\, A_0).$$

The expression for B is

$$B := T + S_1 + S_2 - D_1 - D_2 \bmod p,$$

where the 224-bit terms are given by

$$T = (A_6 \,||\, A_5 \,||\, A_4 \,||\, A_3 \,||\, A_2 \,||\, A_1 \,||\, A_0)$$

$$S_1 = (A_{10} \,||\, A_9 \,||\, A_8 \,||\, A_7 \,||\, 0 \,||\, 0 \,||\, 0)$$

$$S_2 = (0 \,||\, A_{13} \,||\, A_{12} \,||\, A_{11} \,||\, 0 \,||\, 0 \,||\, 0)$$

$$D_1 = (A_{13} \,||\, A_{12} \,||\, A_{11} \,||\, A_{10} \,||\, A_9 \,||\, A_8 \,||\, A_7)$$

$$D_2 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{13} \,||\, A_{12} \,||\, A_{11}).$$

Curve P-256:

The modulus for this curve is $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. Every integer A less than p^2 can be written

$$A = A_{15} \cdot 2^{480} + A_{14} \cdot 2^{448} + A_{13} \cdot 2^{416} + A_{12} \cdot 2^{384} + A_{11} \cdot 2^{352} +$$
$$A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} + A_7 \cdot 2^{224} + A_6 \cdot 2^{192} + A_5 \cdot 2^{160} +$$
$$A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0,$$

where each A_i is a 32-bit integer. As a concatenation of 32-bit words, this can be denoted by

$$A = (A_{15} \,||\, A_{14} \,||\, \cdots \,||\, A_0).$$

The expression for B is

$$B := T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4 \bmod p,$$

where the 256-bit terms are given by

$$T = (A_7 \,||\, A_6 \,||\, A_5 \,||\, A_4 \,||\, A_3 \,||\, A_2 \,||\, A_1 \,||\, A_0)$$
$$S_1 = (A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{11} \,||\, 0 \,||\, 0 \,||\, 0)$$
$$S_2 = (0 \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, 0 \,||\, 0 \,||\, 0)$$
$$S_3 = (A_{15} \,||\, A_{14} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{10} \,||\, A_9 \,||\, A_8)$$
$$S_4 = (A_8 \,||\, A_{13} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{11} \,||\, A_{10} \,||\, A_9)$$
$$D_1 = (A_{10} \,||\, A_8 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{13} \,||\, A_{12} \,||\, A_{11})$$
$$D_2 = (A_{11} \,||\, A_9 \,||\, 0 \,||\, 0 \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12})$$
$$D_3 = (A_{12} \,||\, 0 \,||\, A_{10} \,||\, A_9 \,||\, A_8 \,||\, A_{15} \,||\, A_{14} \,||\, A_{13})$$
$$D_4 = (A_{13} \,||\, 0 \,||\, A_{11} \,||\, A_{10} \,||\, A_9 \,||\, 0 \,||\, A_{15} \,||\, A_{14}).$$

Curve P-384:

The modulus for this curve is $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$. Every integer A less than p^2 can be written

$$A = A_{23} \cdot 2^{736} + A_{22} \cdot 2^{704} + A_{21} \cdot 2^{672} + A_{20} \cdot 2^{640} + A_{19} \cdot 2^{608} +$$
$$A_{18} \cdot 2^{576} + A_{17} \cdot 2^{544} + A_{16} \cdot 2^{512} + A_{15} \cdot 2^{480} + A_{14} \cdot 2^{448} + A_{13} \cdot 2^{416} + A_{12}$$
$$\cdot 2^{384} + A_{11} \cdot 2^{352} + A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} + A_7 \cdot 2^{224} +$$
$$A_6 \cdot 2^{192} + A_5 \cdot 2^{160} + A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0,$$

where each A_i is a 32-bit integer. As a concatenation of 32-bit words, this can be denoted by

$$A = (A_{23} \,||\, A_{22} \,||\, \cdots \,||\, A_0).$$

The expression for B is

$$B := T + 2S_1 + S_2 + S_3 + S_4 + S_5 + S_6 - D_1 - D_2 - D_3 \bmod p,$$

where the 384-bit terms are given by

$$T = (A_{11} \,||\, A_{10} \,||\, A_9 \,||\, A_8 \,||\, A_7 \,||\, A_6 \,||\, A_5 \,||\, A_4 \,||\, A_3 \,||\, A_2 \,||\, A_1 \,||\, A_0)$$
$$S_1 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0)$$
$$S_2 = (A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, A_{20} \,||\, A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12})$$
$$S_3 = (A_{20} \,||\, A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{23} \,||\, A_{22} \,||\, A_{21})$$
$$S_4 = (A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{20} \,||\, 0 \,||\, A_{23} \,||\, 0)$$
$$S_5 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, A_{20} \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0)$$
$$S_6 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{22} \,||\, A_{21} \,||\, 0 \,||\, 0 \,||\, A_{20})$$
$$D_1 = (A_{22} \,||\, A_{21} \,||\, A_{20} \,||\, A_{19} \,||\, A_{18} \,||\, A_{17} \,||\, A_{16} \,||\, A_{15} \,||\, A_{14} \,||\, A_{13} \,||\, A_{12} \,||\, A_{23})$$
$$D_2 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{22} \,||\, A_2 \,||\, A_{20} \,||\, 0)$$
$$D_3 = (0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, 0 \,||\, A_{23} \,||\, A_{23} \,||\, 0 \,||\, 0 \,||\, 0).$$

Curve P-521:

The modulus for this curve is $p = 2^{521} - 1$. Every integer A less than p^2 can be written

$$A = A_1 \cdot 2^{521} + A_0,$$

The expression for B is

$$B := A_0 + A_1 \bmod p$$

APPENDIX 6.2: NORMAL BASES

The elements of $GF(2^m)$ are expressed in terms of the type T normal *basis* 2 B for $GF(2^m)$, for some T. Each element has a unique representation as a bit string

$$(a_0 \, a_1 \ldots \, a_{m-1})$$

The arithmetic operations are performed as follows.

Addition: addition of two elements is implemented by bitwise addition modulo 2. Thus, for example,

$$(1100111) + (1010010) = (0110101).$$

Squaring: if

$$a = (a_0 \, a_1 \ldots \, a_{m-1})$$

then

$$a^2 = (a_{m-1} \, a_0 \, a_1 \ldots \, a_{m-2})$$

Multiplication: to perform multiplication, one first constructs a function $F(\underline{u},\underline{v})$ on inputs

$$\underline{u} = (u_0 \, u_1 \ldots \, u_{m-1}) \qquad \text{and} \qquad \underline{v} = (v_0 \, v_1 \ldots \, v_{m-1})$$

as follows.

1. Set $p \neg Tm + 1$
2. Let u be an integer having order T modulo p

2 It is assumed in this section that m is odd and T is even, since this is the only case considered in this standard.

3. Compute the sequence $F(1); F(2),\ldots,F(p\text{-}1)$ as follows:

   3.1 Set $w \neg 1$

   3.2 For j from 0 to T-1 do

   Set $n \neg w$

   For i from 0 to m-1 do

   Set $F(n) \neg i$

   Set $n \neg 2n \bmod p$

   Set $w \neg uw \bmod p$

4. Output the formula

$$F(\underline{u}, \underline{v}) := \sum_{k=1}^{p-2} u_{F(k+1)} v_{F(p-k)}.$$

This computation need only be performed once per basis.

Given the function F for B, one computes the product

$$(c_0 c_1 \ldots c_{m-1}) = (a_0 a_1 \ldots a_{m-1}) \times (b_0 b_1 \ldots b_{m-1})$$

as follows.

1. Set $(u_0 u_1 \ldots u_{m-1}) \neg (a_0 a_1 \ldots a_{m-1})$
2. Set $(v_0 v_1 \ldots v_{m-1}) \neg (b_0 b_1 \ldots b_{m-1})$
3. For k from 0 to m - 1 do

   3.1 Compute

   $$c_k := F(\underline{u}, \underline{v})$$

   3.2 Set $u \neg$ **LeftShift** (u) and $v \neg$ **LeftShift** (v), where **LeftShift** denotes the circular left shift operation.

4. Output $c := (c_0 c_1 \ldots c_{m-1})$

EXAMPLE. For the type 4 normal basis for $GF(2^7)$, one has $p = 29$ and $u = 12$ or 17. Thus the values of F are given by

$F(1) = 0$	$F(8) = 3$	$F(15) = 6$	$F(22) = 5$
$F(2) = 1$	$F(9) = 3$	$F(16) = 4$	$F(23) = 6$
$F(3) = 5$	$F(10) = 2$	$F(17) = 0$	$F(24) = 1$
$F(4) = 2$	$F(11) = 4$	$F(18) = 4$	$F(25) = 2$
$F(5) = 1$	$F(12) = 0$	$F(19) = 2$	$F(26) = 5$
$F(6) = 6$	$F(13) = 4$	$F(20) = 3$	$F(27) = 1$
$F(7) = 5$	$F(14) = 6$	$F(21) = 3$	$F(28) = 0$

Therefore

$$F(\underline{u}; \underline{v}) = u_0 v_1 + u_1 (v_0 + v_2 + v_5 + v_6) + u_2 (v_1 + v_3 + v_4 + v_5) + u_3 (v_2 + v_5) + u_4 (v_2 + v_6) + u_5 (v_1 + v_2 + v_3 + v_6) + u_6 (v_1 + v_4 + v_5 + v_6).$$

Thus, if

$$a = (1\ 0\ 1\ 0\ 1\ 1\ 1) \text{ and } b = (1\ 1\ 0\ 0\ 0\ 0\ 1),$$

then

$$c_0 = F((1\ 0\ 1\ 0\ 1\ 1\ 1), (1\ 1\ 0\ 0\ 0\ 0\ 1)) = 1,$$

$$c_1 = F((0\ 1\ 0\ 1\ 1\ 1\ 1), (1\ 0\ 0\ 0\ 0\ 1\ 1)) = 0,$$

$$\vdots$$

$$c_6 = F((1\ 1\ 0\ 1\ 0\ 1\ 1); (1\ 1\ 1\ 0\ 0\ 0\ 0)) = 1,$$

so that $c = ab = (1\ 0\ 1\ 1\ 0\ 0\ 1)$.

APPENDIX 6.3: SCALAR MULTIPLICATION ON KOBLITZ CURVES

This appendix describes a particularly efficient method of computing the scalar multiple nP on the Koblitz curve E_a over $GF(2^m)$.

The operation t is defined by

$$t\,(x, y) = (x^2, y^2)$$

When the normal basis representation is used, then the operation t is implemented by performing right circular shifts on the bit strings representing x and y.

Given m and a, define the following parameters:

- C is some integer greater than 5.
- $m := (-1)^{1-a}$
- For $i = 0$ and $i = 1$, define the sequence $s_i(m)$ by

$$s_i(0) = 0, \qquad s_i(1) = 1 - i,$$

$$s_i(m) = m \cdot s_i(m - 1) - 2\, s_i(m - 2) + (-1)^i$$

- Define the sequence $V(m)$

$$V(0) = 2, \qquad V(1) = m$$

$$V(m) = m \cdot v(m - 1) - 2V(m - 2).$$

For the example curves, the quantities $s_i(m)$ and $V(m)$ are as follows.

Curve K-163:

$s_0(163) = 25793864391107316504195 37$

$s_1(163) = -755360064476226375461594$

$V(163) = -48454666325394107768043 17$

Curve K-233:

$s_0(233) = -27859711741434429761757834964435883$

$s_1(233) = -44192136247082304936052160908934886$

$V(233) = -13738154601110823539498729965136 6779$

Curve K-283:

$s_0(283) = -665981532109049041108795536001591469280025$

$s_1(283) = 11558600549091367751922810725916099139459 68$

$V(283) = 77772448708728309992877919709628239775699 17$

Curve K-409:

$s_0(409) = -183075104560023821378103171987564613785905424875568 6\backslash$
9338419259

$s_1(409) = -889304852613830409719665324184421267962656610099660 6\backslash$
444816790

$V(409)=104572887373156259274476853870483207376387969576875 7\backslash$
5791173829

Curve K-571:

$s_0(571)$ = -37373194468764636924293858924761155671472939645961\3

10241234064202352419167299832613 05

$s_1(571)$ = -31918577064464160995838145959489596741319689121485 64\

65861056511758982848515832612248752

$V(571)$=-14838092698169141389961914029705149036454257418049 3\

93623291233953420851682897311145984 3

The following algorithm computes the scalar multiple nP on the Koblitz curve E_a over $GF(2^m)$. The average number of elliptic additions and subtractions is at most ~ 1 + (m/3), and is at most ~ m/3 with probability at least 1 - 2^{5-C}.

For i = 0 to 1 do

$n¢ \leftarrow \lfloor n / 2^{a-C + (m-9) / 2} \rfloor$

$g¢ \leftarrow s_i(m) \cdot n¢$

$h¢ \leftarrow \lfloor g¢ / 2^m \rfloor$

$j¢ \leftarrow V(m) \cdot h¢$

$l¢ \leftarrow \text{Round}((g¢ + j¢) / 2^{(m+5) / 2})$

$l_i \leftarrow l¢ / 2^C$

$f_i \leftarrow \text{Round}(l_i)$

$h_i \leftarrow l_i - f_i$

$h_i \leftarrow 0$

$h \leftarrow 2\ h_0 + m\ h_1$

If h ³ 1

then

if h_o - 3 $m h_1$ < -1

then set $h_1 \leftarrow m$

else set $h_0 \leftarrow 1$

else

if h_0 + 4 m h_1 ³ 2

then set $h_1 \leftarrow m$

If h < -1

then

if h_0 - 3 m $h_1 \geq$ 1

then set $h_1 \leftarrow$ - m

else set $h_0 \leftarrow$ -1

else

if h_0 + 4 m h_1 < -2

then set $h_1 \leftarrow$ - m

$q_0 \leftarrow f_0 + h_0$

$q_1 \leftarrow f_1 + h_1$

$r_0 \leftarrow n$ - (s_0 + m s_1) q_0 - 2s_1 q_1

$r_1 \leftarrow s_1$ q_0 - s_0 q_1

Set $Q \leftarrow O$

$P_0 \leftarrow P$

While r_0 ¹ 0 or r_1 ¹ 0

If r_0 odd then

set $u \leftarrow 2$ - (r_0 - 2 r_1 mod 4)

set $r_0 \leftarrow r_0$ - u

if $u = 1$ then set $Q \leftarrow Q + P_0$

if $u = -1$ then set $Q \leftarrow Q - P_0$

Set $P_0 \leftarrow tP_0$

Set $(r_0 , r_1) \leftarrow (r_1 + mr_0 /2, - r_0 /2)$

Endwhile

Output Q

APPENDIX 6.4: GENERATION OF PSEUDO-RANDOM CURVES (PRIME CASE)

Let l be the bit length of p, and define

$$v = \lfloor (l - 1)/160 \rfloor$$

$$w = l - 160v - 1$$

1. Choose an arbitrary 160-bit string s.
2. Compute h:= SHA-1(s).
3. Let h_0 be the bit string obtained by taking the w rightmost bits of h.
4. Let z be the integer whose binary expansion is given by the 160-bit string s.
5. For i from 1 to v do:
   5.1 Define the 160-bit string s_i to be binary expansion of the integer $(z + i) \bmod (2^{160})$.
   5.2 Compute h_i :=SHA-1(s_i).
6. Let h be the bit string obtained by the concatenation of h_0 , h_1, . . . , h_v as follows:

$$h = h_0 \,||\, h_1 \,||\, \ldots \,||\, h_v$$

7. Let c be the integer whose binary expansion is given by the bit string h.
8. If $c = 0$ or $4c + 27 \equiv 0 \pmod{p}$, then go to Step 1.
9. Choose integers $a, b \in GF(p)$ such that

$$c\, b^2 \equiv a^3 \pmod{p}.$$

(The simplest choice is $a = c$ and $b = c$. However, one may want to choose differently for performance reasons.)

10. Check that the elliptic curve E over $GF(p)$ given by $y^2 = x^3 + ax + b$ has suitable order. If not, go to Step 1.

APPENDIX 6.5: VERIFICATION OF CURVE PSEUDO-RANDOMNESS (PRIME CASE)

Given the 160-bit seed value *s*, one can verify that the coefficient *b* was obtained from *s* via the cryptographic hash function SHA-1 as follows.

Let *l* be the bit length of *p*, and define

$$v = \lfloor (l - 1)/160 \rfloor$$

$$w = l - 160v - 1$$

1. Compute h := SHA-1(s).
2. Let h_0 be the bit string obtained by taking the w rightmost bits of h.
3. Let z be the integer whose binary expansion is given by the 160-bit string s.
4. For i from 1 to v do
   4.1 Define the 160-bit string s_i to be binary expansion of the integer $(z + i) \bmod (2^{160})$.
   4.2 Compute h_i :=SHA-1(s_i).
5. Let h be the bit string obtained by the concatenation of h_0 , h_1, . . . , h_v as follows:

$$h = h_0 \,||\, h_1 \,||\, \ldots \,||\, h_v.$$

6. Let c be the integer whose binary expansion is given by the bit string h.
7. Verify that $b^2 c \equiv -27 \pmod{p}$.

APPENDIX 6.6: GENERATION OF PSEUDO-RANDOM CURVES (BINARY CASE)

Let:

$$v = \lfloor (m - 1) / B \rfloor$$

$$w = m - Bv$$

1. Choose an arbitrary 160-bit string s.

2, Compute h := SHA-1(s)

3. Let h_0 be the bit string obtained by taking the w rightmost bits of h.

4. Let z be the integer whose binary expansion is given by the 160-bit string s.

5. For i from 1 to v do:

   5.1 Define the 160-bit string s_i to be binary expansion of the integer $(z + i) \bmod (2^{160})$.

   5.2 Compute h_i :=SHA-1(s_i).

6. Let h be the bit string obtained by the concatenation of h_0 , h_1, . . . , h_v as follows:

$$h = h_0 \,||\, h_1 \,||\, \ldots \,||\, h_v.$$

7. Let b be the element of $GF(2^m)$ which binary expansion is given by the bit string h.

8. Choose an element a of $GF(2^m)$.

9. Check that the elliptic curve E over $GF(2^m)$ given by $y^2 + xy = x^3 + ax^2 + b$ has suitable order. If not, go to Step 1.

APPENDIX 6.7: VERIFICATION OF CURVE PSEUDO-RANDOMNESS (BINARY CASE)

Given the 160-bit seed value *s*, one can verify that the coefficient *b* was obtained from *s* via the cryptographic hash function SHA-1 as follows.

Define

$$v = \lfloor (m - 1) /160 \rfloor$$

$$w = m - 160v$$

1. Compute h := SHA-1(s)
2. Let h_0 be the bit string obtained by taking the w rightmost bits of h.
3. Let z be the integer whose binary expansion is given by the 160-bit string s.
4. For i from 1 to v do

    4.1 Define the 160-bit string s_i to be binary expansion of the integer ($z + i$) mod (2^{160})

    4.2 Compute h_i :=SHA-1(s_i).
5. Let h be the bit string obtained by the concatenation of h_0 , h_1, . . . , h_v as follows:

$$h = h_0 \,||\, h_1 \,||\, \ldots \,||\, h_v.$$

6. Let c be the element of $GF(2^m)$ which is represented by the bit string h.
7. Verify that $c = b$.

APPENDIX 6.8: POLYNOMIAL BASIS TO NORMAL BASIS CONVERSION

Suppose that a an element of the field $GF(2^m)$. Denote by **p** the bit string representing a with respect to a given polynomial basis. It is desired to compute **n**, the bit string representing a with respect to a given normal basis. This is done via the matrix computation

$$\mathbf{p}\,\Gamma = \mathbf{n}$$

Where Γ is an m-by-m matrix with entries in $GF(2)$. The matrix Γ, which depends only on the bases, can be computed easily given its second-to-last row. The second-to-last row for each conversion is given in the table below.

Degree 163:

3 e173bfaf 3a86434d 883a2918 a489ddbd 69fe84e1

Degree 233:

0be 19b89595 28bbc490
038f4bc4 da8bdfc1 ca36bb05 853fd0ed 0ae200ce

Degree 283:

3347f17 521fdabc 62ec1551 acf156fb
0bceb855 f174d4c1 7807511c 9f745382 add53bc3

Degree 409:

0eb00f2 ea95fd6c 64024e7f
0b68b81f 5ff8a467 acc2b4c3 b9372843 6265c7ff

a06d896c ae3a7e31 e295ec30 3eb9f769 de78bef5

Degree 571:

7940ffa ef996513 4d59dcbf
e5bf239b e4fe4b41 05959c5d 4d942ffd 46ea35f3
e3cdb0e1 04a2aa01 cef30a3a 49478011 196bfb43
c55091b6 1174d7c0 8d0cdd61 3bf6748a bad972a4

Given the second-to-last row **r** of Γ, the rest of the matrix is computed as follows. Let b be the element of $GF(2^m)$ whose representation with respect to the normal basis is **r**. Then the rows of Γ, from top to bottom, are the bit strings representing the elements

$$b^{m-1}, b^{m-2}, \ldots, b^2, b, 1$$

with respect to the normal basis. (Note that the element 1 is represented by the all-1 bit string.)

Alternatively, the matrix is the inverse of the matrix described in Appendix 6.9.

More details of these computations can be found in Annex A.7 of the IEEE P1363 standard.

APPENDIX 6.9: NORMAL BASIS TO POLYNOMIAL BASIS CONVERSION

Suppose that a an element of the field $GF(2^m)$. Denote by **n** the bit string representing a with respect to a given normal basis. It is desired to compute **p**, the bit string representing a with respect to a given polynomial basis. This is done via the matrix computation

$$\mathbf{n}\,\Gamma = \mathbf{p}$$

where Γ is an m-by-m matrix with entries in $GF(2)$. The matrix Γ, which depends only on the bases, can be computed easily given its top row. The top row for each conversion is given in the table below.

Degree 163:

7 15169c10 9c612e39 0d347c74 8342bcd3 b02a0bef

Degree 233:

149 9e398ac5 d79e3685
59b35ca4 9bb7305d a6c0390b cf9e2300 253203c9

Degree 283:

31e0ed7 91c3282d c5624a72 0818049d
053e8c7a b8663792 bc1d792e ba9867fc 7b317a99

Degree 409:

0dfa06b e206aa97 b7a41fff
b9b0c55f 8f048062 fbe8381b 4248adf9 2912ccc8
e3f91a24 e1cfb395 0532b988 971c2304 2e85708d

Degree 571:

452186b bf5840a0 bcf8c9f0
2a54efa0 4e813b43 c3d41496 06c4d27b 487bf107
393c8907 f79d9778 beb35ee8 7467d328 8274caeb
da6ce05a eb4ca5cf 3c3044bd 4372232f 2c1a27c4

Given the top row **r** of Γ, the rest of the matrix is computed as follows. Let b be the element of $GF(2^m)$ whose representation with respect to the polynomial basis is **r**. Then the rows of Γ, from top to bottom, are the bit strings representing the elements

$$b, b^2, b^{2^2}, \ldots, b^{2^{m-1}}$$

with respect to the polynomial basis.

Alternatively, the matrix is the inverse of the matrix described in Appendix 6.8.

More details of these computations can be found in Annex A.7 of the IEEE P1363 standard.

FIPS 186-2, DIGITAL SIGNATURE STANDARD
CHANGE NOTICE 1

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Gaithersburg, MD 20899

DATE OF CHANGE: 2001 October 5

Federal Information Processing Standard (FIPS) 186-2, Digital Signature Standard, specifies the Digital Signature Algorithm (DSA) that may be used in the generation and verification of digital signatures for sensitive, unclassified applications. FIPS 186-2 also allows the use of the digital signature techniques specified in American National Standards Institute (ANSI) X9.31 (Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)) and ANSI X9.62 (Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)). The standard also specifies a transition period for the use of existing (legacy) digital signature systems. Reversible public key algorithms, such as the RSA or Rabin-Williams algorithms, are often used in these legacy systems.

FIPS 186-2 is used in conjunction with the hash function specified in FIPS 180-1, Secure Hash Standard (SHS), and includes specifications for the size of the prime modulus *p*, and algorithms for the generation of a user's private key, *x*, and a user's per message secret number, *k*.

This change notice provides changes for the continued use of DSA as specified in FIPS 186-2 about the size of the prime modulus p, modifications for the random number generation techniques specified in Appendix 3 of FIPS 186-2, and provides instructions for the use of these techniques when used in contexts other than the generation of DSA keys. This change notice also provides guidance for the use of the reversible public key algorithms within legacy systems.

Questions regarding this change notice may be directed to FIPS186@nist.gov or to Elaine Barker (ebarker@nist.gov, 301-975-2911).

The Size of the Prime Modulus

Section 4 of FIPS 186-2 specifies that the prime modulus *p* of DSA is defined for the range of prime integers $2^{L-1} < p < 2^{L}$, where $512 \le L \le 1024$ and *L* is a multiple of 64. This change notice specifies that L should assume only the value 1024 for DSA as specified in FIPS 186-2, i.e., the prime modulus *p* should be defined in the range $2^{1023} < p < 2^{1024}$.

The RSA and Rabin-Williams algorithms used within legacy systems are defined with a modulus *n* and prime factors *p* and *q* of *n*. This change notice specifies that *n* should be at least 1024 bits

in length, and p and q should be approximately half the size of n in bits.

Random Number Generation

FIPS 186-2 includes algorithms for the generation of a user's private key, x, and a user's per message secret number, k. These values must be generated randomly or pseudorandomly and must have values between 0 and the 160-bit prime q (as specified in the standard). Techniques for generating x and k are provided in Appendix 3 of the standard.

Recently, an unpublished attack on DSA3 was found that relies on the non-uniformity of the pseudorandom number generators (PRNGs) specified in Appendix 3 of the standard. The attack has a workfactor of 2^{64} and requires 2^{22} known signatures. This attack can be defended against by either limiting the number of signatures created using a specific key pair to no more than 2 million signatures while using the PRNGs specified in FIPS 186-2, or by modifying the PRNGs.

If the PRNGs currently defined in FIPS 186-2 are used, the user should be provided with clear guidance about the limitation to the number of signatures that should be created.

Alternatively, the following modifications of the PRNGs may be used in lieu of those PRNGs specified in FIPS 186-2. These modifications reduce the non-uniformity of the PRNGs and do not affect interoperability.

The two algorithms described below use a one-way function G(t,c), where t is 160 bits, c is b bits and G(t,c) is 160 bits. Two methods for constructing G are defined in FIPS 186-2: using SHA-1 as defined in FIPS 180-1, and using the Data Encryption Standard (DES) as defined in FIPS 46-3. If G is constructed using SHA-1, b is between 160 and 512 bits ($160 \leq b \leq 512$); if G is constructed using DES, b is equal to 160 bits.

1. Revised Algorithm for Computing *m* values of *x* (Appendix 3.1 of FIPS 186-2)

Let x be the signer's private key. The following may be used to generate m values of x:

Step 1. Choose a new, secret value for the seed-key, $XKEY$.

Step 2. In hexadecimal notation let

$$t = 67452301\ EFCDAB89\ 98BADCFE\ 10325476\ C3D2E1F0.$$

This is the initial value for $H_0 \| H_1 \| H_2 \| H_3 \| H_4$ in the SHS [FIPS 180-1].

---

3 The attack was discovered by Dr. Daniel Bliechenbacher of Lucent Technologies, Bell Labs, Murray Hill, NJ. See a February 25, 2001 press article at http://www.lucent.com/press/0201/010205.bla.html.

Step 3. For j = 0 to m - 1 do

3.1 $XSEED_j$ = optional user input

3.2 For i = 0 to 1 do

a. $XVAL = (XKEY + XSEED_j) \bmod 2^b$

b. $w_i = G(t, XVAL)$.

c. $XKEY = (1 + XKEY + w_i) \bmod 2^b$.

3.3 $x_j = (w_0 \| w_1) \bmod q$

2. Revised Algorithm for Precomputing one or More *k* and *r* Values (Appendix 3.2 of FIPS 186-2)

This algorithm can be used to precompute k, k^{-1}, and r for m messages at a time. Note that implementation of the DSA with precomputation may be covered by U.S. and foreign patents.

Step 1. Choose a secret initial value for the seed-key, $KKEY$.

Step 2. In hexadecimal notation let

t = EFCDAB89 98BADCFE 10325476 C3D2E1F0 67452301.

This is a cyclic shift of the initial value for $H_0 \| H_1 \| H_2 \| H_3 \| H_4$ in the SHS.

Step 3. For j = 0 to m - 1 do

3.1 For i = 0 to 1 do

a. $w_i = G(t, KKEY)$

b. $KKEY = (1 + KKEY + w_i) \bmod 2^b$

3.2 $k = (w_0 \| w_1) \bmod q$

3.3 Compute $k_j^{-1} = k^{-1} \bmod q$

3.4 Compute $r_j = (g^k \bmod p) \bmod q$

Step 4. Suppose M_0 , ... , M_{m-1} are the next m messages. For j = 0 to m - 1 do

a. Let h = SHA-1(M_j).

b. Let $s_j = (k_j^{-1}(h + xr_j)) \bmod q$

c. The signature for M_j is (r_j, s_j).

Step 5. Let $t = h$

Step 6. Go to step 3.

Step 3 permits pre-computation of the quantities needed to sign the next m messages. Step 4 can begin whenever the first of these m messages is ready. The execution of step 4 can be suspended whenever the next of the m messages is not ready. As soon as steps 4 and 5 have completed, step 3 can be executed, and the results saved until the first member of the next group of m messages is ready.

In addition to space for *KKEY* , two arrays of length m are needed to store r_0 , ... r_{m-1} and k_0^{-1}, ... , k_{m-1}^{-1} when they are computed in step 3. Storage for s_0 , ... , s_{m-1} is only needed if the signatures for a group of messages are stored; otherwise s_j in step 4 can be replaced by s, and a single space allocated.

General Purpose Random Number Generation

Several of the FIPS require the use of an Approved (i.e., FIPS-approved or NIST recommended) random number generator (RNG). The RNG specified as algorithm 1 above or the algorithm specified in Appendix 3.1 of FIPS 186-2 may be used in addition to any other Approved RNG. However, when the RNG is used for the generation of random numbers other than for DSA keys, the "mod q" term should be omitted. This will result in the following changes to the specification:

FIPS 186-2, Appendix 3.1, Step 3 c:

Change "$x_j = G(t, XVAL) \bmod q$" to "$x_j = G(t, XVAL)$".

Algorithm 1 of this change notice, Step 3, substep 3.2:

Change "$x_j = (w_0 \| w_1) \bmod q$" to "$x_j = (w_0 \| w_1)$".