```
**********************************************************
   61145 Thu Aug 28 11:51:42 2008
new/usr/src/common/crypto/aes/aes_impl.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
**********************************************************
______unchanged_portion_omitted_
#endif  /* sun4u */
/* EXPORT DELETE END */


/*
 * Initialize key schedules for AES
 *
 * Parameters:
 * cipherKey    User key
 * keyBits      AES key size (128, 192, or 256 bits)
 * keysched     AES key schedule to be initialized, of type aes_key_t.
 *              Allocated by aes_alloc_keysched().
 */
void
aes_init_keysched(const uint8_t *cipherKey, uint_t keyBits, void *keysched)
{
/* EXPORT DELETE START */
        aes_key_t       *newbie = keysched;
        uint_t          keysize, i, j;
        union {
                uint64_t        ka64[4];
                uint32_t        ka32[8];
                } keyarr;

        switch (keyBits) {
        case 128:
                newbie->nr = 10;
                break;

        case 192:
                newbie->nr = 12;
                break;

        case 256:
                newbie->nr = 14;
                break;

        default:
                /* should never get here */
                return;
        }
        keysize = keyBits >> 3;

        /*
         * For _LITTLE_ENDIAN machines (except AMD64), reverse every
         * 4 bytes in the key.  On _BIG_ENDIAN and AMD64, copy the key
         * without reversing bytes.
         * For AMD64, do not byte swap for aes_setupkeys().
         *
         * SPARCv8/v9 uses a key schedule array with 64-bit elements.
         * X86/AMD64  uses a key schedule array with 32-bit elements.
         */
#ifndef AES_BYTE_SWAP
        if (IS_P2ALIGNED(cipherKey, sizeof (uint64_t))) {
                for (i = 0, j = 0; j < keysize; i++, j += 8) {
                        /* LINTED: pointer alignment */
                        keyarr.ka64[i] = *((uint64_t *)&cipherKey[j]);
                }
        } else {
                bcopy(cipherKey, keyarr.ka32, keysize);
        }

#else   /* byte swap */
        for (i = 0, j = 0; j < keysize; i++, j += 4) {
                keyarr.ka32[i] = htonl(*(uint32_t *)(void *)&cipherKey[j]);
```



```
                keyarr.ka32[i] = htonl(*(uint32_t *)&cipherKey[j]);
        }
#endif

        aes_setupkeys(newbie, keyarr.ka32, keyBits);
/* EXPORT DELETE END */
}

/*
 * Encrypt one block using AES.
 * Align if needed and (for x86 32-bit only) byte-swap.
 *
 * Parameters:
 * ks   Key schedule, of type aes_key_t
 * pt   Input block (plain text)
 * ct   Output block (crypto text).  Can overlap with pt
 */
int
aes_encrypt_block(const void *ks, const uint8_t *pt, uint8_t *ct)
{
/* EXPORT DELETE START */
        aes_key_t       *ksch = (aes_key_t *)ks;

#ifndef AES_BYTE_SWAP
        if (IS_P2ALIGNED2(pt, ct, sizeof (uint32_t))) {
                AES_ENCRYPT_IMPL(&ksch->encr_ks.ks32[0], ksch->nr,
                    /* LINTED:  pointer alignment */
                    (uint32_t *)pt, (uint32_t *)ct);
        } else {
#endif
                uint32_t buffer[AES_BLOCK_LEN / sizeof (uint32_t)];

                /* Copy input block into buffer */
#ifndef AES_BYTE_SWAP
                bcopy(pt, &buffer, AES_BLOCK_LEN);

#else   /* byte swap */
                buffer[0] = htonl(*(uint32_t *)(void *)&pt[0]);
                buffer[1] = htonl(*(uint32_t *)(void *)&pt[4]);
                buffer[2] = htonl(*(uint32_t *)(void *)&pt[8]);
                buffer[3] = htonl(*(uint32_t *)(void *)&pt[12]);
                buffer[0] = htonl(*(uint32_t *)&pt[0]);
                buffer[1] = htonl(*(uint32_t *)&pt[4]);
                buffer[2] = htonl(*(uint32_t *)&pt[8]);
                buffer[3] = htonl(*(uint32_t *)&pt[12]);
#endif

                AES_ENCRYPT_IMPL(&ksch->encr_ks.ks32[0], ksch->nr,
                    buffer, buffer);

                /* Copy result from buffer to output block */
#ifndef AES_BYTE_SWAP
                bcopy(&buffer, ct, AES_BLOCK_LEN);
        }

#else   /* byte swap */
                *(uint32_t *)(void *)&ct[0] = htonl(buffer[0]);
                *(uint32_t *)(void *)&ct[4] = htonl(buffer[1]);
                *(uint32_t *)(void *)&ct[8] = htonl(buffer[2]);
                *(uint32_t *)(void *)&ct[12] = htonl(buffer[3]);
                *(uint32_t *)&ct[0] = htonl(buffer[0]);
                *(uint32_t *)&ct[4] = htonl(buffer[1]);
                *(uint32_t *)&ct[8] = htonl(buffer[2]);
                *(uint32_t *)&ct[12] = htonl(buffer[3]);
#endif
/* EXPORT DELETE END */
```

```
        return (CRYPTO_SUCCESS);
}

/*
 * Decrypt one block using AES.
 * Align and byte-swap if needed.
 *
 * Parameters:
 * ks   Key schedule, of type aes_key_t
 * ct   Input block (crypto text)
 * pt   Output block (plain text). Can overlap with pt
 */
int
aes_decrypt_block(const void *ks, const uint8_t *ct, uint8_t *pt)
{
/* EXPORT DELETE START */
        aes_key_t       *ksch = (aes_key_t *)ks;

#ifndef AES_BYTE_SWAP
        if (IS_P2ALIGNED2(ct, pt, sizeof (uint32_t))) {
                AES_DECRYPT_IMPL(&ksch->decr_ks.ks32[0], ksch->nr,
                    /* LINTED:  pointer alignment */
                    (uint32_t *)ct, (uint32_t *)pt);
        } else {
#endif
                uint32_t buffer[AES_BLOCK_LEN / sizeof (uint32_t)];

                /* Copy input block into buffer */
#ifndef AES_BYTE_SWAP
                bcopy(ct, &buffer, AES_BLOCK_LEN);

#else   /* byte swap */
                buffer[0] = htonl(*(uint32_t *)(void *)&ct[0]);
                buffer[1] = htonl(*(uint32_t *)(void *)&ct[4]);
                buffer[2] = htonl(*(uint32_t *)(void *)&ct[8]);
                buffer[3] = htonl(*(uint32_t *)(void *)&ct[12]);
                buffer[0] = htonl(*(uint32_t *)&ct[0]);
                buffer[1] = htonl(*(uint32_t *)&ct[4]);
                buffer[2] = htonl(*(uint32_t *)&ct[8]);
                buffer[3] = htonl(*(uint32_t *)&ct[12]);
#endif

                AES_DECRYPT_IMPL(&ksch->decr_ks.ks32[0], ksch->nr,
                    buffer, buffer);

                /* Copy result from buffer to output block */
#ifndef AES_BYTE_SWAP
                bcopy(&buffer, pt, AES_BLOCK_LEN);
        }

#else   /* byte swap */
        *(uint32_t *)(void *)&pt[0] = htonl(buffer[0]);
        *(uint32_t *)(void *)&pt[4] = htonl(buffer[1]);
        *(uint32_t *)(void *)&pt[8] = htonl(buffer[2]);
        *(uint32_t *)(void *)&pt[12] = htonl(buffer[3]);
        *(uint32_t *)&pt[0] = htonl(buffer[0]);
        *(uint32_t *)&pt[4] = htonl(buffer[1]);
        *(uint32_t *)&pt[8] = htonl(buffer[2]);
        *(uint32_t *)&pt[12] = htonl(buffer[3]);
#endif

/* EXPORT DELETE END */
        return (CRYPTO_SUCCESS);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   26805 Thu Aug 28 11:51:55 2008
new/usr/src/common/crypto/blowfish/blowfish_impl.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
**********************************************************
______unchanged_portion_omitted_

/*
 * Since ROUND() is a macro, make sure that the things inside can be
 * evaluated more than once.  Especially when calling F().
 * Assume the presence of local variables:
 *
 *      uint32_t *P;
 *      uint32_t *S;
 *      uint32_t tmp;
 *
 *
 * And to Microsoft interview survivors out there, perhaps I should do the
 * XOR swap trick, or at least #ifdef (__i386) the tmp = ... = tmp; stuff.
 */

#define F(word) \
        (((S[(word >> 24) & 0xff] + S[256 + ((word >> 16) & 0xff)]) ^ \
                S[512 + ((word >> 8) & 0xff)]) + S[768 + (word & 0xff)])

#define ROUND(left, right, i) \
        (left) ^= P[i]; \
        (right) ^= F((left)); \
        tmp = (left); \
        (left) = (right); \
        (right) = tmp;

/* EXPORT DELETE END */

/*
 * Encrypt a block of data.  Because of addition operations, convert blocks
 * to their big-endian representation, even on Intel boxen.
 */
/* ARGSUSED */
int
blowfish_encrypt_block(const void *cookie, const uint8_t *block,
    uint8_t *out_block)
{
/* EXPORT DELETE START */
        keysched_t *ksch = (keysched_t *)cookie;

        uint32_t left, right, tmp;
        uint32_t *P = ksch->ksch_P;
        uint32_t *S = ksch->ksch_S;
#ifdef _BIG_ENDIAN
        uint32_t *b32;

        if (IS_P2ALIGNED(block, sizeof (uint32_t))) {
                /* LINTED:  pointer alignment */
                b32 = (uint32_t *)block;
                left = b32[0];
                right = b32[1];
        } else
#endif
        {
        /*
         * Read input block and place in left/right in big-endian order.
         */
#ifdef UNALIGNED_POINTERS_PERMITTED
        left = htonl(*(uint32_t *)(void *)&block[0]);
        right = htonl(*(uint32_t *)(void *)&block[4]);
        left = htonl(*(uint32_t *)&block[0]);
        right = htonl(*(uint32_t *)&block[4]);
#else
        left = ((uint32_t)block[0] << 24)
            | ((uint32_t)block[1] << 16)
```



```
            | ((uint32_t)block[2] << 8)
            | (uint32_t)block[3];
        right = ((uint32_t)block[4] << 24)
            | ((uint32_t)block[5] << 16)
            | ((uint32_t)block[6] << 8)
            | (uint32_t)block[7];
#endif  /* UNALIGNED_POINTERS_PERMITTED */
        }

        ROUND(left, right, 0);
        ROUND(left, right, 1);
        ROUND(left, right, 2);
        ROUND(left, right, 3);
        ROUND(left, right, 4);
        ROUND(left, right, 5);
        ROUND(left, right, 6);
        ROUND(left, right, 7);
        ROUND(left, right, 8);
        ROUND(left, right, 9);
        ROUND(left, right, 10);
        ROUND(left, right, 11);
        ROUND(left, right, 12);
        ROUND(left, right, 13);
        ROUND(left, right, 14);
        ROUND(left, right, 15);

        tmp = left;
        left = right;
        right = tmp;
        right ^= P[16];
        left ^= P[17];

#ifdef _BIG_ENDIAN
        if (IS_P2ALIGNED(out_block, sizeof (uint32_t))) {
                /* LINTED:  pointer alignment */
                b32 = (uint32_t *)out_block;
                b32[0] = left;
                b32[1] = right;
        } else
#endif
        {
                /* Put the block back into the user's block with final swap */
#ifdef UNALIGNED_POINTERS_PERMITTED
                *(uint32_t *)(void *)&out_block[0] = htonl(left);
                *(uint32_t *)(void *)&out_block[4] = htonl(right);
                *(uint32_t *)&out_block[0] = htonl(left);
                *(uint32_t *)&out_block[4] = htonl(right);
#else
                out_block[0] = left >> 24;
                out_block[1] = left >> 16;
                out_block[2] = left >> 8;
                out_block[3] = left;
                out_block[4] = right >> 24;
                out_block[5] = right >> 16;
                out_block[6] = right >> 8;
                out_block[7] = right;
#endif  /* UNALIGNED_POINTERS_PERMITTED */
        }
/* EXPORT DELETE END */
        return (CRYPTO_SUCCESS);
}

/*
 * Decrypt a block of data.  Because of addition operations, convert blocks
 * to their big-endian representation, even on Intel boxen.
 * It should look like the blowfish_encrypt_block() operation
```

```
 * except for the order in which the S/P boxes are accessed.
 */
/* ARGSUSED */
int
blowfish_decrypt_block(const void *cookie, const uint8_t *block,
    uint8_t *out_block)
{
/* EXPORT DELETE START */
        keysched_t *ksch = (keysched_t *)cookie;

        uint32_t left, right, tmp;
        uint32_t *P = ksch->ksch_P;
        uint32_t *S = ksch->ksch_S;
#ifdef _BIG_ENDIAN
        uint32_t *b32;

        if (IS_P2ALIGNED(block, sizeof (uint32_t))) {
                /* LINTED:  pointer alignment */
                b32 = (uint32_t *)block;
                left = b32[0];
                right = b32[1];
        } else
#endif
        {
        /*
         * Read input block and place in left/right in big-endian order.
         */
#ifdef UNALIGNED_POINTERS_PERMITTED
        left = htonl(*(uint32_t *)(void *)&block[0]);
        right = htonl(*(uint32_t *)(void *)&block[4]);
        left = htonl(*(uint32_t *)&block[0]);
        right = htonl(*(uint32_t *)&block[4]);
#else
        left = ((uint32_t)block[0] << 24)
            | ((uint32_t)block[1] << 16)
            | ((uint32_t)block[2] << 8)
            | (uint32_t)block[3];
        right = ((uint32_t)block[4] << 24)
            | ((uint32_t)block[5] << 16)
            | ((uint32_t)block[6] << 8)
            | (uint32_t)block[7];
#endif  /* UNALIGNED_POINTERS_PERMITTED */
        }

        ROUND(left, right, 17);
        ROUND(left, right, 16);
        ROUND(left, right, 15);
        ROUND(left, right, 14);
        ROUND(left, right, 13);
        ROUND(left, right, 12);
        ROUND(left, right, 11);
        ROUND(left, right, 10);
        ROUND(left, right, 9);
        ROUND(left, right, 8);
        ROUND(left, right, 7);
        ROUND(left, right, 6);
        ROUND(left, right, 5);
        ROUND(left, right, 4);
        ROUND(left, right, 3);
        ROUND(left, right, 2);

        tmp = left;
        left = right;
        right = tmp;
        right ^= P[1];
        left ^= P[0];
```



```
#ifdef _BIG_ENDIAN
        if (IS_P2ALIGNED(out_block, sizeof (uint32_t))) {
                /* LINTED:  pointer alignment */
                b32 = (uint32_t *)out_block;
                b32[0] = left;
                b32[1] = right;
        } else
#endif
        {
        /* Put the block back into the user's block with final swap */
#ifdef UNALIGNED_POINTERS_PERMITTED
                *(uint32_t *)(void *)&out_block[0] = htonl(left);
                *(uint32_t *)(void *)&out_block[4] = htonl(right);
                *(uint32_t *)&out_block[0] = htonl(left);
                *(uint32_t *)&out_block[4] = htonl(right);
#else
                out_block[0] = left >> 24;
                out_block[1] = left >> 16;
                out_block[2] = left >> 8;
                out_block[3] = left;
                out_block[4] = right >> 24;
                out_block[5] = right >> 16;
                out_block[6] = right >> 8;
                out_block[7] = right;
#endif  /* UNALIGNED_POINTERS_PERMITTED */
        }
/* EXPORT DELETE END */
        return (CRYPTO_SUCCESS);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   47321 Thu Aug 28 11:52:05 2008
new/usr/src/common/crypto/des/des_impl.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
**********************************************************
______unchanged_portion_omitted_
#endif /* !sun4u */

/* EXPORT DELETE END */

int
des3_crunch_block(const void *cookie, const uint8_t block[DES_BLOCK_LEN],
    uint8_t out_block[DES_BLOCK_LEN], boolean_t decrypt)
{
/* EXPORT DELETE START */
	keysched3_t *ksch = (keysched3_t *)cookie;

	/*
	 * The code below, that is always executed on LITTLE_ENDIAN machines,
	 * reverses bytes in the block.  On BIG_ENDIAN, the same code
	 * copies the block without reversing bytes.
	 */
#ifdef _BIG_ENDIAN
	if (IS_P2ALIGNED(block, sizeof (uint64_t)) &&
	    IS_P2ALIGNED(out_block, sizeof (uint64_t))) {
		if (decrypt == B_TRUE)
			/* LINTED */
			*(uint64_t *)out_block = des_crypt_impl(
			    ksch->ksch_decrypt, /* LINTED */
			    *(uint64_t *)block, 3);
		else
			/* LINTED */
			*(uint64_t *)out_block = des_crypt_impl(
			    ksch->ksch_encrypt, /* LINTED */
			    *(uint64_t *)block, 3);
	} else
#endif	/* _BIG_ENDIAN */
	{
		uint64_t tmp;

#ifdef UNALIGNED_POINTERS_PERMITTED
		tmp = htonll(*(uint64_t *)(void *)&block[0]);
		tmp = htonll(*(uint64_t *)&block[0]);
#else
		tmp = (((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
		    ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
		    ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
		    ((uint64_t)block[6] << 8) | (uint64_t)block[7]);
#endif	/* UNALIGNED_POINTERS_PERMITTED */

		if (decrypt == B_TRUE)
			tmp = des_crypt_impl(ksch->ksch_decrypt, tmp, 3);
		else
			tmp = des_crypt_impl(ksch->ksch_encrypt, tmp, 3);

#ifdef UNALIGNED_POINTERS_PERMITTED
		*(uint64_t *)(void *)&out_block[0] = htonll(tmp);
		*(uint64_t *)&out_block[0] = htonll(tmp);
#else
		out_block[0] = tmp >> 56;
		out_block[1] = tmp >> 48;
		out_block[2] = tmp >> 40;
		out_block[3] = tmp >> 32;
		out_block[4] = tmp >> 24;
		out_block[5] = tmp >> 16;
		out_block[6] = tmp >> 8;
		out_block[7] = (uint8_t)tmp;
#endif	/* UNALIGNED_POINTERS_PERMITTED */
	}
/* EXPORT DELETE END */
	return (CRYPTO_SUCCESS);
```



```
}

int
des_crunch_block(const void *cookie, const uint8_t block[DES_BLOCK_LEN],
    uint8_t out_block[DES_BLOCK_LEN], boolean_t decrypt)
{
/* EXPORT DELETE START */
	keysched_t *ksch = (keysched_t *)cookie;

	/*
	 * The code below, that is always executed on LITTLE_ENDIAN machines,
	 * reverses bytes in the block.  On BIG_ENDIAN, the same code
	 * copies the block without reversing bytes.
	 */
#ifdef _BIG_ENDIAN
	if (IS_P2ALIGNED(block, sizeof (uint64_t)) &&
	    IS_P2ALIGNED(out_block, sizeof (uint64_t))) {
		if (decrypt == B_TRUE)
			/* LINTED */
			*(uint64_t *)out_block = des_crypt_impl(
			    ksch->ksch_decrypt, /* LINTED */
			    *(uint64_t *)block, 1);
		else
			/* LINTED */
			*(uint64_t *)out_block = des_crypt_impl(
			    ksch->ksch_encrypt, /* LINTED */
			    *(uint64_t *)block, 1);

	} else
#endif	/* _BIG_ENDIAN */
	{
		uint64_t tmp;

#ifdef UNALIGNED_POINTERS_PERMITTED
		tmp = htonll(*(uint64_t *)(void *)&block[0]);
		tmp = htonll(*(uint64_t *)&block[0]);
#else
		tmp = (((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
		    ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
		    ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
		    ((uint64_t)block[6] << 8) | (uint64_t)block[7]);
#endif	/* UNALIGNED_POINTERS_PERMITTED */


		if (decrypt == B_TRUE)
			tmp = des_crypt_impl(ksch->ksch_decrypt, tmp, 1);
		else
			tmp = des_crypt_impl(ksch->ksch_encrypt, tmp, 1);

#ifdef UNALIGNED_POINTERS_PERMITTED
		*(uint64_t *)(void *)&out_block[0] = htonll(tmp);
		*(uint64_t *)&out_block[0] = htonll(tmp);
#else
		out_block[0] = tmp >> 56;
		out_block[1] = tmp >> 48;
		out_block[2] = tmp >> 40;
		out_block[3] = tmp >> 32;
		out_block[4] = tmp >> 24;
		out_block[5] = tmp >> 16;
		out_block[6] = tmp >> 8;
		out_block[7] = (uint8_t)tmp;
#endif	/* UNALIGNED_POINTERS_PERMITTED */
	}
/* EXPORT DELETE END */
	return (CRYPTO_SUCCESS);
}
```

```
static boolean_t
keycheck(uint8_t *key, uint8_t *corrected_key)
{
/* EXPORT DELETE START */
        uint64_t key_so_far;
        uint_t i;
        /*
         * Table of weak and semi-weak keys.  Fortunately, weak keys are
         * endian-independent, and some semi-weak keys can be paired up in
         * endian-opposite order.  Since keys are stored as uint64_t's,
         * use the ifdef _LITTLE_ENDIAN where appropriate.
         */
        static uint64_t des_weak_keys[] = {
                /* Really weak keys.  Byte-order independent values. */
                0x0101010101010101ULL,
                0x1f1f1f1f0e0e0e0eULL,
                0xe0e0e0e0f1f1f1f1ULL,
                0xfefefefefefefefeULL,

                /* Semi-weak (and a few possibly-weak) keys. */

                /* Byte-order independent semi-weak keys. */
                0x01fe01fe01fe01feULL,  0xfe01fe01fe01fe01ULL,

                /* Byte-order dependent semi-weak keys. */
#ifdef _LITTLE_ENDIAN
                0xf10ef10ee01fe01fULL,  0x0ef10ef11fe01fe0ULL,
                0x01f101f101e001e0ULL,  0xf101f101e001e001ULL,
                0x0efe0efe1ffe1ffeULL,  0xfe0efe0efe1ffe1fULL,
                0x010e010e011f011fULL,  0x0e010e011f011f01ULL,
                0xf1fef1fee0fee0feULL,  0xfef1fef1fee0fee0ULL,
#else   /* Big endian */
                0x1fe01fe00ef10ef1ULL,  0xe01fe01ff10ef10eULL,
                0x01e001e001f101f1ULL,  0xe001e001f101f101ULL,
                0x1ffe1ffe0efe0efeULL,  0xfe1ffe1ffe0efe0eULL,
                0x011f011f010e010eULL,  0x1f011f010e010e01ULL,
                0xe0fee0fef1fef1feULL,  0xfee0fee0fef1fef1ULL,
#endif  /* _LITTLE_ENDIAN */

                /* We'll save the other possibly-weak keys for the future. */
        };

        if (key == NULL)
                return (B_FALSE);

#ifdef UNALIGNED_POINTERS_PERMITTED
        key_so_far = htonll(*(uint64_t *)(void *)&key[0]);
        key_so_far = htonll(*(uint64_t *)&key[0]);
#else
        /*
         * The code below reverses the bytes on LITTLE_ENDIAN machines.
         * On BIG_ENDIAN, the same code copies without reversing
         * the bytes.
         */
        key_so_far = (((uint64_t)key[0] << 56) | ((uint64_t)key[1] << 48) |
            ((uint64_t)key[2] << 40) | ((uint64_t)key[3] << 32) |
            ((uint64_t)key[4] << 24) | ((uint64_t)key[5] << 16) |
            ((uint64_t)key[6] << 8) | (uint64_t)key[7]);
#endif  /* UNALIGNED_POINTERS_PERMITTED */

        /*
         * Fix parity.
         */
        fix_des_parity(&key_so_far);
```



```
        /* Do weak key check itself. */
        for (i = 0; i < (sizeof (des_weak_keys) / sizeof (uint64_t)); i++)
                if (key_so_far == des_weak_keys[i]) {
                        return (B_FALSE);
                }

        if (corrected_key != NULL) {
#ifdef UNALIGNED_POINTERS_PERMITTED
                *(uint64_t *)(void *)&corrected_key[0] = htonll(key_so_far);
                *(uint64_t *)&corrected_key[0] = htonll(key_so_far);
#else
                /*
                 * The code below reverses the bytes on LITTLE_ENDIAN machines.
                 * On BIG_ENDIAN, the same code copies without reversing
                 * the bytes.
                 */
                corrected_key[0] = key_so_far >> 56;
                corrected_key[1] = key_so_far >> 48;
                corrected_key[2] = key_so_far >> 40;
                corrected_key[3] = key_so_far >> 32;
                corrected_key[4] = key_so_far >> 24;
                corrected_key[5] = key_so_far >> 16;
                corrected_key[6] = key_so_far >> 8;
                corrected_key[7] = (uint8_t)key_so_far;
#endif  /* UNALIGNED_POINTERS_PERMITTED */
        }
/* EXPORT DELETE END */
        return (B_TRUE);
}
```

______***unchanged_portion_omitted_***

```
void
des_parity_fix(uint8_t *key, des_strength_t strength, uint8_t *corrected_key)
{
/* EXPORT DELETE START */
        uint64_t aligned_key[DES3_KEYSIZE / sizeof (uint64_t)];
        uint8_t *paritied_key;
        uint64_t key_so_far;
        int i = 0, offset = 0;

        if (strength == DES)
                bcopy(key, aligned_key, DES_KEYSIZE);
        else
                bcopy(key, aligned_key, DES3_KEYSIZE);

        paritied_key = (uint8_t *)aligned_key;
        while (strength > i) {
                offset = 8 * i;
#ifdef UNALIGNED_POINTERS_PERMITTED
                key_so_far = htonll(*(uint64_t *)(void *)&paritied_key[offset]);
                key_so_far = htonll(*(uint64_t *)&paritied_key[offset]);
#else
                key_so_far = (((uint64_t)paritied_key[offset + 0] << 56) |
                    ((uint64_t)paritied_key[offset + 1] << 48) |
                    ((uint64_t)paritied_key[offset + 2] << 40) |
                    ((uint64_t)paritied_key[offset + 3] << 32) |
                    ((uint64_t)paritied_key[offset + 4] << 24) |
                    ((uint64_t)paritied_key[offset + 5] << 16) |
                    ((uint64_t)paritied_key[offset + 6] << 8) |
                    (uint64_t)paritied_key[offset + 7]);
#endif  /* UNALIGNED_POINTERS_PERMITTED */

                fix_des_parity(&key_so_far);

#ifdef UNALIGNED_POINTERS_PERMITTED
                *(uint64_t *)(void *)&paritied_key[offset] = htonll(key_so_far);
```

```
                *(uint64_t *)&paritied_key[offset] = htonll(key_so_far);
#else
                paritied_key[offset + 0] = key_so_far >> 56;
                paritied_key[offset + 1] = key_so_far >> 48;
                paritied_key[offset + 2] = key_so_far >> 40;
                paritied_key[offset + 3] = key_so_far >> 32;
                paritied_key[offset + 4] = key_so_far >> 24;
                paritied_key[offset + 5] = key_so_far >> 16;
                paritied_key[offset + 6] = key_so_far >> 8;
                paritied_key[offset + 7] = (uint8_t)key_so_far;
#endif  /* UNALIGNED_POINTERS_PERMITTED */

                i++;
        }

        bcopy(paritied_key, corrected_key, DES_KEYSIZE * strength);
/* EXPORT DELETE END */
}


/*
 * Initialize key schedule for DES, DES2, and DES3
 */
void
des_init_keysched(uint8_t *cipherKey, des_strength_t strength, void *ks)
{
/* EXPORT DELETE START */
        uint64_t *encryption_ks;
        uint64_t *decryption_ks;
        uint64_t keysched[48];
        uint64_t key_uint64[3];
        uint64_t tmp;
        uint_t keysize, i, j;

        switch (strength) {
        case DES:
                keysize = DES_KEYSIZE;
                encryption_ks = ((keysched_t *)ks)->ksch_encrypt;
                decryption_ks = ((keysched_t *)ks)->ksch_decrypt;
                break;
        case DES2:
                keysize = DES2_KEYSIZE;
                encryption_ks = ((keysched3_t *)ks)->ksch_encrypt;
                decryption_ks = ((keysched3_t *)ks)->ksch_decrypt;
                break;
        case DES3:
                keysize = DES3_KEYSIZE;
                encryption_ks = ((keysched3_t *)ks)->ksch_encrypt;
                decryption_ks = ((keysched3_t *)ks)->ksch_decrypt;
        }

        /*
         * The code below, that is always executed on LITTLE_ENDIAN machines,
         * reverses every 8 bytes in the key.  On BIG_ENDIAN, the same code
         * copies the key without reversing bytes.
         */
#ifdef _BIG_ENDIAN
        if (IS_P2ALIGNED(cipherKey, sizeof (uint64_t))) {
                for (i = 0, j = 0; j < keysize; i++, j += 8) {
                        /* LINTED: pointer alignment */
                        key_uint64[i] = *((uint64_t *)&cipherKey[j]);
                }
        } else
#endif  /* _BIG_ENDIAN */
        {
                for (i = 0, j = 0; j < keysize; i++, j += 8) {
```



```
#ifdef UNALIGNED_POINTERS_PERMITTED
                        key_uint64[i] =
                            htonll(*(uint64_t *)(void *)&cipherKey[j]);
                        key_uint64[i] = htonll(*(uint64_t *)&cipherKey[j]);
#else
                        key_uint64[i] = (((uint64_t)cipherKey[j] << 56) |
                            ((uint64_t)cipherKey[j + 1] << 48) |
                            ((uint64_t)cipherKey[j + 2] << 40) |
                            ((uint64_t)cipherKey[j + 3] << 32) |
                            ((uint64_t)cipherKey[j + 4] << 24) |
                            ((uint64_t)cipherKey[j + 5] << 16) |
                            ((uint64_t)cipherKey[j + 6] << 8) |
                            (uint64_t)cipherKey[j + 7]);
#endif  /* UNALIGNED_POINTERS_PERMITTED */
                }
        }

        switch (strength) {
        case DES:
                des_ks(keysched, key_uint64[0]);
                break;

        case DES2:
                /* DES2 is just DES3 with the first and third keys the same */
                bcopy(key_uint64, key_uint64 + 2, DES_KEYSIZE);
                /* FALLTHRU */
        case DES3:
                des_ks(keysched, key_uint64[0]);
                des_ks(keysched + 16, key_uint64[1]);
                for (i = 0; i < 8; i++) {
                        tmp = keysched[16+i];
                        keysched[16+i] = keysched[31-i];
                        keysched[31-i] = tmp;
                }
                des_ks(keysched+32, key_uint64[2]);
                keysize = DES3_KEYSIZE;
        }

        /* save the encryption keyschedule */
        bcopy(keysched, encryption_ks, keysize * 16);

        /* reverse the key schedule */
        for (i = 0; i < keysize; i++) {
                tmp = keysched[i];
                keysched[i] = keysched[2 * keysize - 1 - i];
                keysched[2 * keysize -1 -i] = tmp;
        }

        /* save the decryption keyschedule */
        bcopy(keysched, decryption_ks, keysize * 16);
/* EXPORT DELETE END */
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
    8356 Thu Aug 28 11:52:20 2008
new/usr/src/common/crypto/md4/md4.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
**********************************************************
______unchanged_portion_omitted_

/*
 * Encodes input (uint32_t) into output (unsigned char). Assumes len is
 * a multiple of 4.
 */
static void
Encode(unsigned char *output, uint32_t *input, unsigned int len)
{
        unsigned int i, j;

        for (i = 0, j = 0; j < len; i++, j += 4) {
#if defined(_LITTLE_ENDIAN) && defined(UNALIGNED_POINTERS_PERMITTED)
                *(uint32_t *)(void *)&output[j] = input[i];
                *(uint32_t *)&output[j] = input[i];
#else
                /* endian-independent code */
                output[j] = (unsigned char)(input[i] & 0xff);
                output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
                output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
                output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
#endif  /* _LITTLE_ENDIAN && UNALIGNED_POINTERS_PERMITTED */
        }
}

/*
 * Decodes input (unsigned char) into output (uint32_t). Assumes len is
 * a multiple of 4.
 */
static void
Decode(uint32_t *output, unsigned char *input, unsigned int len)
{
        unsigned int i, j;

        for (i = 0, j = 0; j < len; i++, j += 4) {
#if defined(_LITTLE_ENDIAN) && defined(UNALIGNED_POINTERS_PERMITTED)
                output[i] = *(uint32_t *)(void *)&input[j];
                output[i] = *(uint32_t *)&input[j];
#else
                /* endian-independent code */
                output[i] = ((uint32_t)input[j]) |
                    (((uint32_t)input[j+1]) << 8) |
                    (((uint32_t)input[j+2]) << 16) |
                    (((uint32_t)input[j+3]) << 24);
#endif  /* _LITTLE_ENDIAN && UNALIGNED_POINTERS_PERMITTED */
        }

}
______unchanged_portion_omitted_
```

****
25256 Thu Aug 28 11:52:36 2008
new/usr/src/common/crypto/modes/ccm.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
****
______unchanged_portion_omitted_

```
/*
 * This will decrypt the cipher text.  However, the plaintext won't be
 * returned to the caller.  It will be returned when decrypt_final() is
 * called if the MAC matches
 */
/* ARGSUSED */
int
ccm_mode_decrypt_contiguous_blocks(ccm_ctx_t *ctx, char *data, size_t length,
    crypto_data_t *out, size_t block_size,
    int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
    void (*copy_block)(uint8_t *, uint8_t *),
    void (*xor_block)(uint8_t *, uint8_t *))
{
        size_t remainder = length;
        size_t need;
        uint8_t *datap = (uint8_t *)data;
        uint8_t *blockp;
        uint8_t *cbp;
        uint64_t counter;
        size_t pt_len, total_decrypted_len, mac_len, pm_len, pd_len;
        uint8_t *resultp;
#ifdef _LITTLE_ENDIAN
        uint8_t *p;
#endif  /* _LITTLE_ENDIAN */


        pm_len = ctx->ccm_processed_mac_len;

        if (pm_len > 0) {
                uint8_t *tmp;
                /*
                 * all ciphertext has been processed, just waiting for
                 * part of the value of the mac
                 */
                if ((pm_len + length) > ctx->ccm_mac_len) {
                        return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
                }
                tmp = (uint8_t *)ctx->ccm_mac_input_buf;

                bcopy(datap, tmp + pm_len, length);

                ctx->ccm_processed_mac_len += length;
                return (CRYPTO_SUCCESS);
        }

        /*
         * If we decrypt the given data, what total amount of data would
         * have been decrypted?
         */
        pd_len = ctx->ccm_processed_data_len;
        total_decrypted_len = pd_len + length + ctx->ccm_remainder_len;

        if (total_decrypted_len >
            (ctx->ccm_data_len + ctx->ccm_mac_len)) {
                return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
        }

        pt_len = ctx->ccm_data_len;

        if (total_decrypted_len > pt_len) {
                /*
                 * part of the input will be the MAC, need to isolate that
                 * to be dealt with later.  The left-over data in
                 * ccm_remainder_len from last time will not be part of the
```



```
                 * MAC.  Otherwise, it would have already been taken out
                 * when this call is made last time.
                 */
                size_t pt_part = pt_len - pd_len - ctx->ccm_remainder_len;

                mac_len = length - pt_part;

                ctx->ccm_processed_mac_len = mac_len;
                bcopy(data + pt_part, ctx->ccm_mac_input_buf, mac_len);

                if (pt_part + ctx->ccm_remainder_len < block_size) {
                        /*
                         * since this is last of the ciphertext, will
                         * just decrypt with it here
                         */
                        bcopy(datap, &((uint8_t *)ctx->ccm_remainder)
                            [ctx->ccm_remainder_len], pt_part);
                        ctx->ccm_remainder_len += pt_part;
                        ccm_decrypt_incomplete_block(ctx, encrypt_block);
                        ctx->ccm_remainder_len = 0;
                        ctx->ccm_processed_data_len += pt_part;
                        return (CRYPTO_SUCCESS);
                } else {
                        /* let rest of the code handle this */
                        length = pt_part;
                }
        } else if (length + ctx->ccm_remainder_len < block_size) {
                        /* accumulate bytes here and return */
                bcopy(datap,
                    (uint8_t *)ctx->ccm_remainder + ctx->ccm_remainder_len,
                    length);
                ctx->ccm_remainder_len += length;
                ctx->ccm_copy_to = datap;
                return (CRYPTO_SUCCESS);
        }

        do {
                /* Unprocessed data from last call. */
                if (ctx->ccm_remainder_len > 0) {
                        need = block_size - ctx->ccm_remainder_len;

                        if (need > remainder)
                                return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);

                        bcopy(datap, &((uint8_t *)ctx->ccm_remainder)
                            [ctx->ccm_remainder_len], need);

                        blockp = (uint8_t *)ctx->ccm_remainder;
                } else {
                        blockp = datap;
                }

                /* Calculate the counter mode, ccm_cb is the counter block */
                cbp = (uint8_t *)ctx->ccm_tmp;
                encrypt_block(ctx->ccm_keysched, (uint8_t *)ctx->ccm_cb, cbp);

                /*
                 * Increment counter.
                 * Counter bits are confined to the bottom 64 bits
                 */
#ifdef _LITTLE_ENDIAN
                counter = ntohll(ctx->ccm_cb[1] & ctx->ccm_counter_mask);
                counter = htonll(counter + 1);
#else
                counter = ctx->ccm_cb[1] & ctx->ccm_counter_mask;
                counter++;
```

```
#endif  /* _LITTLE_ENDIAN */
                counter &= ctx->ccm_counter_mask;
                ctx->ccm_cb[1] =
                    (ctx->ccm_cb[1] & ~(ctx->ccm_counter_mask)) | counter;

                /* XOR with the ciphertext */
                xor_block(blockp, cbp);

                /* Copy the plaintext to the "holding buffer" */
                resultp = (uint8_t *)ctx->ccm_pt_buf +
                    ctx->ccm_processed_data_len;
                copy_block(cbp, resultp);

                ctx->ccm_processed_data_len += block_size;

                ctx->ccm_lastp = blockp;

                /* Update pointer to next block of data to be processed. */
                if (ctx->ccm_remainder_len != 0) {
                        datap += need;
                        ctx->ccm_remainder_len = 0;
                } else {
                        datap += block_size;
                }

                remainder = (size_t)&data[length] - (size_t)datap;

                /* Incomplete last block */
                if (remainder > 0 && remainder < block_size) {
                        bcopy(datap, ctx->ccm_remainder, remainder);
                        ctx->ccm_remainder_len = remainder;
                        ctx->ccm_copy_to = datap;
                        if (ctx->ccm_processed_mac_len > 0) {
                                /*
                                 * not expecting anymore ciphertext, just
                                 * compute plaintext for the remaining input
                                 */
                                ccm_decrypt_incomplete_block(ctx,
                                    encrypt_block);
                                ctx->ccm_processed_data_len += remainder;
                                ctx->ccm_remainder_len = 0;
                        }
                        goto out;
                }
                ctx->ccm_copy_to = NULL;

        } while (remainder > 0);

out:
        return (CRYPTO_SUCCESS);
}
```

______***unchanged_portion_omitted_***

```
************************************************************
    6134 Thu Aug 28 11:52:49 2008
new/usr/src/common/crypto/modes/ctr.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
************************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _KERNEL
#include <strings.h>
#include <limits.h>
#include <assert.h>
#include <security/cryptoki.h>
#endif

#include <sys/types.h>
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>

#ifdef _LITTLE_ENDIAN
#include <sys/byteorder.h>
#endif

/*
 * Encrypt and decrypt multiple blocks of data in counter mode.
 */
int
ctr_mode_contiguous_blocks(ctr_ctx_t *ctx, char *data, size_t length,
    crypto_data_t *out, size_t block_size,
    int (*cipher)(const void *ks, const uint8_t *pt, uint8_t *ct),
    void (*xor_block)(uint8_t *, uint8_t *))
{
        size_t remainder = length;
        size_t need;
        uint8_t *datap = (uint8_t *)data;
        uint8_t *blockp;
        uint8_t *lastp;
        void *iov_or_mp;
        offset_t offset;
        uint8_t *out_data_1;
        uint8_t *out_data_2;
        size_t out_data_1_len;
        uint64_t counter;
```



```
#ifdef _LITTLE_ENDIAN
        uint8_t *p;
#endif

        if (length + ctx->ctr_remainder_len < block_size) {
                /* accumulate bytes here and return */
                bcopy(datap,
                    (uint8_t *)ctx->ctr_remainder + ctx->ctr_remainder_len,
                    length);
                ctx->ctr_remainder_len += length;
                ctx->ctr_copy_to = datap;
                return (CRYPTO_SUCCESS);
        }

        lastp = (uint8_t *)ctx->ctr_cb;
        if (out != NULL)
                crypto_init_ptrs(out, &iov_or_mp, &offset);

        do {
                /* Unprocessed data from last call. */
                if (ctx->ctr_remainder_len > 0) {
                        need = block_size - ctx->ctr_remainder_len;

                        if (need > remainder)
                                return (CRYPTO_DATA_LEN_RANGE);

                        bcopy(datap, &((uint8_t *)ctx->ctr_remainder)
                            [ctx->ctr_remainder_len], need);

                        blockp = (uint8_t *)ctx->ctr_remainder;
                } else {
                        blockp = datap;
                }

                /* ctr_cb is the counter block */
                cipher(ctx->ctr_keysched, (uint8_t *)ctx->ctr_cb,
                    (uint8_t *)ctx->ctr_tmp);

                lastp = (uint8_t *)ctx->ctr_tmp;

                /*
                 * Increment counter. Counter bits are confined
                 * to the bottom 64 bits of the counter block.
                 */
#ifdef _LITTLE_ENDIAN
                counter = ntohll(ctx->ctr_cb[1] & ctx->ctr_counter_mask);
                counter = htonll(counter + 1);
#else
                counter = ctx->ctr_cb[1] & ctx->ctr_counter_mask;
                counter++;
#endif  /* _LITTLE_ENDIAN */
                counter &= ctx->ctr_counter_mask;
                ctx->ctr_cb[1] =
                    (ctx->ctr_cb[1] & ~(ctx->ctr_counter_mask)) | counter;

                /*
                 * XOR the previous cipher block or IV with the
                 * current clear block.
                 */
                xor_block(blockp, lastp);

                if (out == NULL) {
                        if (ctx->ctr_remainder_len > 0) {
                                bcopy(lastp, ctx->ctr_copy_to,
                                    ctx->ctr_remainder_len);
                                bcopy(lastp + ctx->ctr_remainder_len, datap,
```

```
                                    need);
                        }
                } else {
                        crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
                            &out_data_1_len, &out_data_2, block_size);

                        /* copy block to where it belongs */
                        bcopy(lastp, out_data_1, out_data_1_len);
                        if (out_data_2 != NULL) {
                                bcopy(lastp + out_data_1_len, out_data_2,
                                    block_size - out_data_1_len);
                        }
                        /* update offset */
                        out->cd_offset += block_size;
                }

                /* Update pointer to next block of data to be processed. */
                if (ctx->ctr_remainder_len != 0) {
                        datap += need;
                        ctx->ctr_remainder_len = 0;
                } else {
                        datap += block_size;
                }

                remainder = (size_t)&data[length] - (size_t)datap;

                /* Incomplete last block. */
                if (remainder > 0 && remainder < block_size) {
                        bcopy(datap, ctx->ctr_remainder, remainder);
                        ctx->ctr_remainder_len = remainder;
                        ctx->ctr_copy_to = datap;
                        goto out;
                }
                ctx->ctr_copy_to = NULL;

        } while (remainder > 0);

out:
        return (CRYPTO_SUCCESS);
}
```

______***unchanged_portion_omitted_***

****
31598 Thu Aug 28 11:52:59 2008
new/usr/src/common/crypto/sha1/sha1.c
6717509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
****
______*unchanged_portion_omitted_*

```
#if !defined(__amd64)

typedef uint32_t sha1word;

/*
 * sparc optimization:
 *
 * on the sparc, we can load big endian 32-bit data easily.  note that
 * special care must be taken to ensure the address is 32-bit aligned.
 * in the interest of speed, we don't check to make sure, since
 * careful programming can guarantee this for us.
 */

#if     defined(_BIG_ENDIAN)
#define LOAD_BIG_32(addr)       (*(uint32_t *)(addr))

#elif   defined(HAVE_HTONL)
#define LOAD_BIG_32(addr) htonl(*((uint32_t *)(addr)))

#else
/* little endian -- will work on big endian, but slowly */
#define LOAD_BIG_32(addr)       \
        (((addr)[0] << 24) | ((addr)[1] << 16) | ((addr)[2] << 8) | (addr)[3])
#endif  /* _BIG_ENDIAN */

/*
 * SHA1Transform()
 */
#if     defined(W_ARRAY)
#define W(n) w[n]
#else   /* !defined(W_ARRAY) */
#define W(n) w_ ## n
#endif  /* !defined(W_ARRAY) */


#if     defined(__sparc)

/*
 * sparc register window optimization:
 *
 * 'a', 'b', 'c', 'd', and 'e' are passed into SHA1Transform
 * explicitly since it increases the number of registers available to
 * the compiler.  under this scheme, these variables can be held in
 * %i0 - %i4, which leaves more local and out registers available.
 *
 * purpose: sha1 transformation -- updates the digest based on 'block'
 *   input: uint32_t    : bytes  1 -  4 of the digest
 *          uint32_t    : bytes  5 -  8 of the digest
 *          uint32_t    : bytes  9 - 12 of the digest
 *          uint32_t    : bytes 12 - 16 of the digest
 *          uint32_t    : bytes 16 - 20 of the digest
 *          SHA1_CTX *  : the context to update
 *          uint8_t [64]: the block to use to update the digest
 *  output: void
 */

void
SHA1Transform(uint32_t a, uint32_t b, uint32_t c, uint32_t d, uint32_t e,
    SHA1_CTX *ctx, const uint8_t blk[64])
{
        /*
         * sparc optimization:
         *
         * while it is somewhat counter-intuitive, on sparc, it is
         * more efficient to place all the constants used in this
         * function in an array and load the values out of the array
         * than to manually load the constants.  this is because
         * setting a register to a 32-bit value takes two ops in most
         * cases: a 'sethi' and an 'or', but loading a 32-bit value
         * from memory only takes one 'ld' (or 'lduw' on v9).  while
         * this increases memory usage, the compiler can find enough
         * other things to do while waiting to keep the pipeline does
         * not stall.  additionally, it is likely that many of these
         * constants are cached so that later accesses do not even go
         * out to the bus.
         *
         * this array is declared 'static' to keep the compiler from
         * having to bcopy() this array onto the stack frame of
         * SHA1Transform() each time it is called -- which is
         * unacceptably expensive.
         *
         * the 'const' is to ensure that callers are good citizens and
         * do not try to munge the array.  since these routines are
         * going to be called from inside multithreaded kernelland,
         * this is a good safety check. -- 'sha1_consts' will end up in
         * .rodata.
         *
         * unfortunately, loading from an array in this manner hurts
         * performance under Intel.  So, there is a macro,
         * SHA1_CONST(), used in SHA1Transform(), that either expands to
         * a reference to this array, or to the actual constant,
         * depending on what platform this code is compiled for.
         */

        static const uint32_t sha1_consts[] = {
                SHA1_CONST_0, SHA1_CONST_1, SHA1_CONST_2, SHA1_CONST_3
        };

        /*
         * general optimization:
         *
         * use individual integers instead of using an array.  this is a
         * win, although the amount it wins by seems to vary quite a bit.
         */

        uint32_t        w_0, w_1, w_2,  w_3,  w_4,  w_5,  w_6,  w_7;
        uint32_t        w_8, w_9, w_10, w_11, w_12, w_13, w_14, w_15;

        /*
         * sparc optimization:
         *
         * if 'block' is already aligned on a 4-byte boundary, use
         * LOAD_BIG_32() directly.  otherwise, bcopy() into a
         * buffer that *is* aligned on a 4-byte boundary and then do
         * the LOAD_BIG_32() on that buffer.  benchmarks have shown
         * that using the bcopy() is better than loading the bytes
         * individually and doing the endian-swap by hand.
         *
         * even though it's quite tempting to assign to do:
         *
         * blk = bcopy(ctx->buf_un.buf32, blk, sizeof (ctx->buf_un.buf32));
         *
         * and only have one set of LOAD_BIG_32()'s, the compiler
         * *does not* like that, so please resist the urge.
         */

        if ((uintptr_t)blk & 0x3) {             /* not 4-byte aligned? */
                bcopy(blk, ctx->buf_un.buf32,  sizeof (ctx->buf_un.buf32));
                w_15 = LOAD_BIG_32(ctx->buf_un.buf32 + 15);
```

```
                w_14 = LOAD_BIG_32(ctx->buf_un.buf32 + 14);
                w_13 = LOAD_BIG_32(ctx->buf_un.buf32 + 13);
                w_12 = LOAD_BIG_32(ctx->buf_un.buf32 + 12);
                w_11 = LOAD_BIG_32(ctx->buf_un.buf32 + 11);
                w_10 = LOAD_BIG_32(ctx->buf_un.buf32 + 10);
                w_9  = LOAD_BIG_32(ctx->buf_un.buf32 +  9);
                w_8  = LOAD_BIG_32(ctx->buf_un.buf32 +  8);
                w_7  = LOAD_BIG_32(ctx->buf_un.buf32 +  7);
                w_6  = LOAD_BIG_32(ctx->buf_un.buf32 +  6);
                w_5  = LOAD_BIG_32(ctx->buf_un.buf32 +  5);
                w_4  = LOAD_BIG_32(ctx->buf_un.buf32 +  4);
                w_3  = LOAD_BIG_32(ctx->buf_un.buf32 +  3);
                w_2  = LOAD_BIG_32(ctx->buf_un.buf32 +  2);
                w_1  = LOAD_BIG_32(ctx->buf_un.buf32 +  1);
                w_0  = LOAD_BIG_32(ctx->buf_un.buf32 +  0);
        } else {
                /*LINTED*/
                w_15 = LOAD_BIG_32(blk + 60);
                /*LINTED*/
                w_14 = LOAD_BIG_32(blk + 56);
                /*LINTED*/
                w_13 = LOAD_BIG_32(blk + 52);
                /*LINTED*/
                w_12 = LOAD_BIG_32(blk + 48);
                /*LINTED*/
                w_11 = LOAD_BIG_32(blk + 44);
                /*LINTED*/
                w_10 = LOAD_BIG_32(blk + 40);
                /*LINTED*/
                w_9  = LOAD_BIG_32(blk + 36);
                /*LINTED*/
                w_8  = LOAD_BIG_32(blk + 32);
                /*LINTED*/
                w_7  = LOAD_BIG_32(blk + 28);
                /*LINTED*/
                w_6  = LOAD_BIG_32(blk + 24);
                /*LINTED*/
                w_5  = LOAD_BIG_32(blk + 20);
                /*LINTED*/
                w_4  = LOAD_BIG_32(blk + 16);
                /*LINTED*/
                w_3  = LOAD_BIG_32(blk + 12);
                /*LINTED*/
                w_2  = LOAD_BIG_32(blk +  8);
                /*LINTED*/
                w_1  = LOAD_BIG_32(blk +  4);
                /*LINTED*/
                w_0  = LOAD_BIG_32(blk +  0);
        }
#else   /* !defined(__sparc) */

void /* CSTYLED */
SHA1Transform(SHA1_CTX *ctx, const uint8_t blk[64])
{
        /* CSTYLED */
        sha1word a = ctx->state[0];
        sha1word b = ctx->state[1];
        sha1word c = ctx->state[2];
        sha1word d = ctx->state[3];
        sha1word e = ctx->state[4];

#if     defined(W_ARRAY)
        sha1word        w[16];
#else   /* !defined(W_ARRAY) */
        sha1word        w_0, w_1, w_2,  w_3,  w_4,  w_5,  w_6,  w_7;
        sha1word        w_8, w_9, w_10, w_11, w_12, w_13, w_14, w_15;
```



```
#endif  /* !defined(W_ARRAY) */

        W(0)  = LOAD_BIG_32((void *)(blk +  0));
        W(1)  = LOAD_BIG_32((void *)(blk +  4));
        W(2)  = LOAD_BIG_32((void *)(blk +  8));
        W(3)  = LOAD_BIG_32((void *)(blk + 12));
        W(4)  = LOAD_BIG_32((void *)(blk + 16));
        W(5)  = LOAD_BIG_32((void *)(blk + 20));
        W(6)  = LOAD_BIG_32((void *)(blk + 24));
        W(7)  = LOAD_BIG_32((void *)(blk + 28));
        W(8)  = LOAD_BIG_32((void *)(blk + 32));
        W(9)  = LOAD_BIG_32((void *)(blk + 36));
        W(10) = LOAD_BIG_32((void *)(blk + 40));
        W(11) = LOAD_BIG_32((void *)(blk + 44));
        W(12) = LOAD_BIG_32((void *)(blk + 48));
        W(13) = LOAD_BIG_32((void *)(blk + 52));
        W(14) = LOAD_BIG_32((void *)(blk + 56));
        W(15) = LOAD_BIG_32((void *)(blk + 60));
        W(0)  = LOAD_BIG_32(blk +  0);
        W(1)  = LOAD_BIG_32(blk +  4);
        W(2)  = LOAD_BIG_32(blk +  8);
        W(3)  = LOAD_BIG_32(blk + 12);
        W(4)  = LOAD_BIG_32(blk + 16);
        W(5)  = LOAD_BIG_32(blk + 20);
        W(6)  = LOAD_BIG_32(blk + 24);
        W(7)  = LOAD_BIG_32(blk + 28);
        W(8)  = LOAD_BIG_32(blk + 32);
        W(9)  = LOAD_BIG_32(blk + 36);
        W(10) = LOAD_BIG_32(blk + 40);
        W(11) = LOAD_BIG_32(blk + 44);
        W(12) = LOAD_BIG_32(blk + 48);
        W(13) = LOAD_BIG_32(blk + 52);
        W(14) = LOAD_BIG_32(blk + 56);
        W(15) = LOAD_BIG_32(blk + 60);

#endif  /* !defined(__sparc) */

        /*
         * general optimization:
         *
         * even though this approach is described in the standard as
         * being slower algorithmically, it is 30-40% faster than the
         * "faster" version under SPARC, because this version has more
         * of the constraints specified at compile-time and uses fewer
         * variables (and therefore has better register utilization)
         * than its "speedier" brother.  (i've tried both, trust me)
         *
         * for either method given in the spec, there is an "assignment"
         * phase where the following takes place:
         *
         *      tmp = (main_computation);
         *      e = d; d = c; c = rotate_left(b, 30); b = a; a = tmp;
         *
         * we can make the algorithm go faster by not doing this work,
         * but just pretending that `d' is now `e', etc. this works
         * really well and obviates the need for a temporary variable.
         * however, we still explicitly perform the rotate action,
         * since it is cheaper on SPARC to do it once than to have to
         * do it over and over again.
         */

        /* round 1 */
        e = ROTATE_LEFT(a, 5) + F(b, c, d) + e + W(0) + SHA1_CONST(0); /* 0 */
        b = ROTATE_LEFT(b, 30);

        d = ROTATE_LEFT(e, 5) + F(a, b, c) + d + W(1) + SHA1_CONST(0); /* 1 */
```

```
        a = ROTATE_LEFT(a, 30);

        c = ROTATE_LEFT(d, 5) + F(e, a, b) + c + W(2) + SHA1_CONST(0); /* 2 */
        e = ROTATE_LEFT(e, 30);

        b = ROTATE_LEFT(c, 5) + F(d, e, a) + b + W(3) + SHA1_CONST(0); /* 3 */
        d = ROTATE_LEFT(d, 30);

        a = ROTATE_LEFT(b, 5) + F(c, d, e) + a + W(4) + SHA1_CONST(0); /* 4 */
        c = ROTATE_LEFT(c, 30);

        e = ROTATE_LEFT(a, 5) + F(b, c, d) + e + W(5) + SHA1_CONST(0); /* 5 */
        b = ROTATE_LEFT(b, 30);

        d = ROTATE_LEFT(e, 5) + F(a, b, c) + d + W(6) + SHA1_CONST(0); /* 6 */
        a = ROTATE_LEFT(a, 30);

        c = ROTATE_LEFT(d, 5) + F(e, a, b) + c + W(7) + SHA1_CONST(0); /* 7 */
        e = ROTATE_LEFT(e, 30);

        b = ROTATE_LEFT(c, 5) + F(d, e, a) + b + W(8) + SHA1_CONST(0); /* 8 */
        d = ROTATE_LEFT(d, 30);

        a = ROTATE_LEFT(b, 5) + F(c, d, e) + a + W(9) + SHA1_CONST(0); /* 9 */
        c = ROTATE_LEFT(c, 30);

        e = ROTATE_LEFT(a, 5) + F(b, c, d) + e + W(10) + SHA1_CONST(0); /* 10 */
        b = ROTATE_LEFT(b, 30);

        d = ROTATE_LEFT(e, 5) + F(a, b, c) + d + W(11) + SHA1_CONST(0); /* 11 */
        a = ROTATE_LEFT(a, 30);

        c = ROTATE_LEFT(d, 5) + F(e, a, b) + c + W(12) + SHA1_CONST(0); /* 12 */
        e = ROTATE_LEFT(e, 30);

        b = ROTATE_LEFT(c, 5) + F(d, e, a) + b + W(13) + SHA1_CONST(0); /* 13 */
        d = ROTATE_LEFT(d, 30);

        a = ROTATE_LEFT(b, 5) + F(c, d, e) + a + W(14) + SHA1_CONST(0); /* 14 */
        c = ROTATE_LEFT(c, 30);

        e = ROTATE_LEFT(a, 5) + F(b, c, d) + e + W(15) + SHA1_CONST(0); /* 15 */
        b = ROTATE_LEFT(b, 30);

        W(0) = ROTATE_LEFT((W(13) ^ W(8) ^ W(2) ^ W(0)), 1);            /* 16 */
        d = ROTATE_LEFT(e, 5) + F(a, b, c) + d + W(0) + SHA1_CONST(0);
        a = ROTATE_LEFT(a, 30);

        W(1) = ROTATE_LEFT((W(14) ^ W(9) ^ W(3) ^ W(1)), 1);            /* 17 */
        c = ROTATE_LEFT(d, 5) + F(e, a, b) + c + W(1) + SHA1_CONST(0);
        e = ROTATE_LEFT(e, 30);

        W(2) = ROTATE_LEFT((W(15) ^ W(10) ^ W(4) ^ W(2)), 1);   /* 18 */
        b = ROTATE_LEFT(c, 5) + F(d, e, a) + b + W(2) + SHA1_CONST(0);
        d = ROTATE_LEFT(d, 30);

        W(3) = ROTATE_LEFT((W(0) ^ W(11) ^ W(5) ^ W(3)), 1);            /* 19 */
        a = ROTATE_LEFT(b, 5) + F(c, d, e) + a + W(3) + SHA1_CONST(0);
        c = ROTATE_LEFT(c, 30);

        /* round 2 */
        W(4) = ROTATE_LEFT((W(1) ^ W(12) ^ W(6) ^ W(4)), 1);            /* 20 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(4) + SHA1_CONST(1);
        b = ROTATE_LEFT(b, 30);

        W(5) = ROTATE_LEFT((W(2) ^ W(13) ^ W(7) ^ W(5)), 1);            /* 21 */
```



```
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(5) + SHA1_CONST(1);
        a = ROTATE_LEFT(a, 30);

        W(6) = ROTATE_LEFT((W(3) ^ W(14) ^ W(8) ^ W(6)), 1);            /* 22 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(6) + SHA1_CONST(1);
        e = ROTATE_LEFT(e, 30);

        W(7) = ROTATE_LEFT((W(4) ^ W(15) ^ W(9) ^ W(7)), 1);            /* 23 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(7) + SHA1_CONST(1);
        d = ROTATE_LEFT(d, 30);

        W(8) = ROTATE_LEFT((W(5) ^ W(0) ^ W(10) ^ W(8)), 1);            /* 24 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(8) + SHA1_CONST(1);
        c = ROTATE_LEFT(c, 30);

        W(9) = ROTATE_LEFT((W(6) ^ W(1) ^ W(11) ^ W(9)), 1);            /* 25 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(9) + SHA1_CONST(1);
        b = ROTATE_LEFT(b, 30);

        W(10) = ROTATE_LEFT((W(7) ^ W(2) ^ W(12) ^ W(10)), 1);  /* 26 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(10) + SHA1_CONST(1);
        a = ROTATE_LEFT(a, 30);

        W(11) = ROTATE_LEFT((W(8) ^ W(3) ^ W(13) ^ W(11)), 1);  /* 27 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(11) + SHA1_CONST(1);
        e = ROTATE_LEFT(e, 30);

        W(12) = ROTATE_LEFT((W(9) ^ W(4) ^ W(14) ^ W(12)), 1);  /* 28 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(12) + SHA1_CONST(1);
        d = ROTATE_LEFT(d, 30);

        W(13) = ROTATE_LEFT((W(10) ^ W(5) ^ W(15) ^ W(13)), 1); /* 29 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(13) + SHA1_CONST(1);
        c = ROTATE_LEFT(c, 30);

        W(14) = ROTATE_LEFT((W(11) ^ W(6) ^ W(0) ^ W(14)), 1);  /* 30 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(14) + SHA1_CONST(1);
        b = ROTATE_LEFT(b, 30);

        W(15) = ROTATE_LEFT((W(12) ^ W(7) ^ W(1) ^ W(15)), 1);  /* 31 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(15) + SHA1_CONST(1);
        a = ROTATE_LEFT(a, 30);

        W(0) = ROTATE_LEFT((W(13) ^ W(8) ^ W(2) ^ W(0)), 1);            /* 32 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(0) + SHA1_CONST(1);
        e = ROTATE_LEFT(e, 30);

        W(1) = ROTATE_LEFT((W(14) ^ W(9) ^ W(3) ^ W(1)), 1);            /* 33 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(1) + SHA1_CONST(1);
        d = ROTATE_LEFT(d, 30);

        W(2) = ROTATE_LEFT((W(15) ^ W(10) ^ W(4) ^ W(2)), 1);   /* 34 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(2) + SHA1_CONST(1);
        c = ROTATE_LEFT(c, 30);

        W(3) = ROTATE_LEFT((W(0) ^ W(11) ^ W(5) ^ W(3)), 1);            /* 35 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(3) + SHA1_CONST(1);
        b = ROTATE_LEFT(b, 30);

        W(4) = ROTATE_LEFT((W(1) ^ W(12) ^ W(6) ^ W(4)), 1);            /* 36 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(4) + SHA1_CONST(1);
        a = ROTATE_LEFT(a, 30);

        W(5) = ROTATE_LEFT((W(2) ^ W(13) ^ W(7) ^ W(5)), 1);            /* 37 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(5) + SHA1_CONST(1);
        e = ROTATE_LEFT(e, 30);
```

```
        W(6) = ROTATE_LEFT((W(3) ^ W(14) ^ W(8) ^ W(6)), 1);           /* 38 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(6) + SHA1_CONST(1);
        d = ROTATE_LEFT(d, 30);

        W(7) = ROTATE_LEFT((W(4) ^ W(15) ^ W(9) ^ W(7)), 1);           /* 39 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(7) + SHA1_CONST(1);
        c = ROTATE_LEFT(c, 30);

        /* round 3 */
        W(8) = ROTATE_LEFT((W(5) ^ W(0) ^ W(10) ^ W(8)), 1);           /* 40 */
        e = ROTATE_LEFT(a, 5) + H(b, c, d) + e + W(8) + SHA1_CONST(2);
        b = ROTATE_LEFT(b, 30);

        W(9) = ROTATE_LEFT((W(6) ^ W(1) ^ W(11) ^ W(9)), 1);           /* 41 */
        d = ROTATE_LEFT(e, 5) + H(a, b, c) + d + W(9) + SHA1_CONST(2);
        a = ROTATE_LEFT(a, 30);

        W(10) = ROTATE_LEFT((W(7) ^ W(2) ^ W(12) ^ W(10)), 1);  /* 42 */
        c = ROTATE_LEFT(d, 5) + H(e, a, b) + c + W(10) + SHA1_CONST(2);
        e = ROTATE_LEFT(e, 30);

        W(11) = ROTATE_LEFT((W(8) ^ W(3) ^ W(13) ^ W(11)), 1);  /* 43 */
        b = ROTATE_LEFT(c, 5) + H(d, e, a) + b + W(11) + SHA1_CONST(2);
        d = ROTATE_LEFT(d, 30);

        W(12) = ROTATE_LEFT((W(9) ^ W(4) ^ W(14) ^ W(12)), 1);  /* 44 */
        a = ROTATE_LEFT(b, 5) + H(c, d, e) + a + W(12) + SHA1_CONST(2);
        c = ROTATE_LEFT(c, 30);

        W(13) = ROTATE_LEFT((W(10) ^ W(5) ^ W(15) ^ W(13)), 1); /* 45 */
        e = ROTATE_LEFT(a, 5) + H(b, c, d) + e + W(13) + SHA1_CONST(2);
        b = ROTATE_LEFT(b, 30);

        W(14) = ROTATE_LEFT((W(11) ^ W(6) ^ W(0) ^ W(14)), 1);  /* 46 */
        d = ROTATE_LEFT(e, 5) + H(a, b, c) + d + W(14) + SHA1_CONST(2);
        a = ROTATE_LEFT(a, 30);

        W(15) = ROTATE_LEFT((W(12) ^ W(7) ^ W(1) ^ W(15)), 1);  /* 47 */
        c = ROTATE_LEFT(d, 5) + H(e, a, b) + c + W(15) + SHA1_CONST(2);
        e = ROTATE_LEFT(e, 30);

        W(0) = ROTATE_LEFT((W(13) ^ W(8) ^ W(2) ^ W(0)), 1);           /* 48 */
        b = ROTATE_LEFT(c, 5) + H(d, e, a) + b + W(0) + SHA1_CONST(2);
        d = ROTATE_LEFT(d, 30);

        W(1) = ROTATE_LEFT((W(14) ^ W(9) ^ W(3) ^ W(1)), 1);           /* 49 */
        a = ROTATE_LEFT(b, 5) + H(c, d, e) + a + W(1) + SHA1_CONST(2);
        c = ROTATE_LEFT(c, 30);

        W(2) = ROTATE_LEFT((W(15) ^ W(10) ^ W(4) ^ W(2)), 1);   /* 50 */
        e = ROTATE_LEFT(a, 5) + H(b, c, d) + e + W(2) + SHA1_CONST(2);
        b = ROTATE_LEFT(b, 30);

        W(3) = ROTATE_LEFT((W(0) ^ W(11) ^ W(5) ^ W(3)), 1);           /* 51 */
        d = ROTATE_LEFT(e, 5) + H(a, b, c) + d + W(3) + SHA1_CONST(2);
        a = ROTATE_LEFT(a, 30);

        W(4) = ROTATE_LEFT((W(1) ^ W(12) ^ W(6) ^ W(4)), 1);           /* 52 */
        c = ROTATE_LEFT(d, 5) + H(e, a, b) + c + W(4) + SHA1_CONST(2);
        e = ROTATE_LEFT(e, 30);

        W(5) = ROTATE_LEFT((W(2) ^ W(13) ^ W(7) ^ W(5)), 1);           /* 53 */
        b = ROTATE_LEFT(c, 5) + H(d, e, a) + b + W(5) + SHA1_CONST(2);
        d = ROTATE_LEFT(d, 30);
```



```
        W(6) = ROTATE_LEFT((W(3) ^ W(14) ^ W(8) ^ W(6)), 1);           /* 54 */
        a = ROTATE_LEFT(b, 5) + H(c, d, e) + a + W(6) + SHA1_CONST(2);
        c = ROTATE_LEFT(c, 30);

        W(7) = ROTATE_LEFT((W(4) ^ W(15) ^ W(9) ^ W(7)), 1);           /* 55 */
        e = ROTATE_LEFT(a, 5) + H(b, c, d) + e + W(7) + SHA1_CONST(2);
        b = ROTATE_LEFT(b, 30);

        W(8) = ROTATE_LEFT((W(5) ^ W(0) ^ W(10) ^ W(8)), 1);           /* 56 */
        d = ROTATE_LEFT(e, 5) + H(a, b, c) + d + W(8) + SHA1_CONST(2);
        a = ROTATE_LEFT(a, 30);

        W(9) = ROTATE_LEFT((W(6) ^ W(1) ^ W(11) ^ W(9)), 1);           /* 57 */
        c = ROTATE_LEFT(d, 5) + H(e, a, b) + c + W(9) + SHA1_CONST(2);
        e = ROTATE_LEFT(e, 30);

        W(10) = ROTATE_LEFT((W(7) ^ W(2) ^ W(12) ^ W(10)), 1);  /* 58 */
        b = ROTATE_LEFT(c, 5) + H(d, e, a) + b + W(10) + SHA1_CONST(2);
        d = ROTATE_LEFT(d, 30);

        W(11) = ROTATE_LEFT((W(8) ^ W(3) ^ W(13) ^ W(11)), 1);  /* 59 */
        a = ROTATE_LEFT(b, 5) + H(c, d, e) + a + W(11) + SHA1_CONST(2);
        c = ROTATE_LEFT(c, 30);

        /* round 4 */
        W(12) = ROTATE_LEFT((W(9) ^ W(4) ^ W(14) ^ W(12)), 1);  /* 60 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(12) + SHA1_CONST(3);
        b = ROTATE_LEFT(b, 30);

        W(13) = ROTATE_LEFT((W(10) ^ W(5) ^ W(15) ^ W(13)), 1); /* 61 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(13) + SHA1_CONST(3);
        a = ROTATE_LEFT(a, 30);

        W(14) = ROTATE_LEFT((W(11) ^ W(6) ^ W(0) ^ W(14)), 1);  /* 62 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(14) + SHA1_CONST(3);
        e = ROTATE_LEFT(e, 30);

        W(15) = ROTATE_LEFT((W(12) ^ W(7) ^ W(1) ^ W(15)), 1);  /* 63 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(15) + SHA1_CONST(3);
        d = ROTATE_LEFT(d, 30);

        W(0) = ROTATE_LEFT((W(13) ^ W(8) ^ W(2) ^ W(0)), 1);           /* 64 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(0) + SHA1_CONST(3);
        c = ROTATE_LEFT(c, 30);

        W(1) = ROTATE_LEFT((W(14) ^ W(9) ^ W(3) ^ W(1)), 1);           /* 65 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(1) + SHA1_CONST(3);
        b = ROTATE_LEFT(b, 30);

        W(2) = ROTATE_LEFT((W(15) ^ W(10) ^ W(4) ^ W(2)), 1);   /* 66 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(2) + SHA1_CONST(3);
        a = ROTATE_LEFT(a, 30);

        W(3) = ROTATE_LEFT((W(0) ^ W(11) ^ W(5) ^ W(3)), 1);           /* 67 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(3) + SHA1_CONST(3);
        e = ROTATE_LEFT(e, 30);

        W(4) = ROTATE_LEFT((W(1) ^ W(12) ^ W(6) ^ W(4)), 1);           /* 68 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(4) + SHA1_CONST(3);
        d = ROTATE_LEFT(d, 30);

        W(5) = ROTATE_LEFT((W(2) ^ W(13) ^ W(7) ^ W(5)), 1);           /* 69 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(5) + SHA1_CONST(3);
        c = ROTATE_LEFT(c, 30);

        W(6) = ROTATE_LEFT((W(3) ^ W(14) ^ W(8) ^ W(6)), 1);           /* 70 */
```

```
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(6) + SHA1_CONST(3);
        b = ROTATE_LEFT(b, 30);

        W(7) = ROTATE_LEFT((W(4) ^ W(15) ^ W(9) ^ W(7)), 1);            /* 71 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(7) + SHA1_CONST(3);
        a = ROTATE_LEFT(a, 30);

        W(8) = ROTATE_LEFT((W(5) ^ W(0) ^ W(10) ^ W(8)), 1);            /* 72 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(8) + SHA1_CONST(3);
        e = ROTATE_LEFT(e, 30);

        W(9) = ROTATE_LEFT((W(6) ^ W(1) ^ W(11) ^ W(9)), 1);            /* 73 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(9) + SHA1_CONST(3);
        d = ROTATE_LEFT(d, 30);

        W(10) = ROTATE_LEFT((W(7) ^ W(2) ^ W(12) ^ W(10)), 1);  /* 74 */
        a = ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(10) + SHA1_CONST(3);
        c = ROTATE_LEFT(c, 30);

        W(11) = ROTATE_LEFT((W(8) ^ W(3) ^ W(13) ^ W(11)), 1);  /* 75 */
        e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + W(11) + SHA1_CONST(3);
        b = ROTATE_LEFT(b, 30);

        W(12) = ROTATE_LEFT((W(9) ^ W(4) ^ W(14) ^ W(12)), 1);  /* 76 */
        d = ROTATE_LEFT(e, 5) + G(a, b, c) + d + W(12) + SHA1_CONST(3);
        a = ROTATE_LEFT(a, 30);

        W(13) = ROTATE_LEFT((W(10) ^ W(5) ^ W(15) ^ W(13)), 1); /* 77 */
        c = ROTATE_LEFT(d, 5) + G(e, a, b) + c + W(13) + SHA1_CONST(3);
        e = ROTATE_LEFT(e, 30);

        W(14) = ROTATE_LEFT((W(11) ^ W(6) ^ W(0) ^ W(14)), 1);  /* 78 */
        b = ROTATE_LEFT(c, 5) + G(d, e, a) + b + W(14) + SHA1_CONST(3);
        d = ROTATE_LEFT(d, 30);

        W(15) = ROTATE_LEFT((W(12) ^ W(7) ^ W(1) ^ W(15)), 1);  /* 79 */

        ctx->state[0] += ROTATE_LEFT(b, 5) + G(c, d, e) + a + W(15) +
            SHA1_CONST(3);
        ctx->state[1] += b;
        ctx->state[2] += ROTATE_LEFT(c, 30);
        ctx->state[3] += d;
        ctx->state[4] += e;

        /* zeroize sensitive information */
        W(0) = W(1) = W(2) = W(3) = W(4) = W(5) = W(6) = W(7) = W(8) = 0;
        W(9) = W(10) = W(11) = W(12) = W(13) = W(14) = W(15) = 0;
}
```

______***unchanged_portion_omitted_***