

new/usr/src/common/crypto/aes/aes_impl.c

1

```
*****
61145 Thu Aug 28 11:51:42 2008
new/usr/src/common/crypto/aes/aes_impl.c
6171509 Need to use bswap/bswapq for byte swap of 64-bit integer on x32/x64 (fix
*****
unchanged portion omitted
1427 #endif /* sun4u */
1428 /* EXPORT DELETE END */

1431 /*
1432 * Initialize key schedules for AES
1433 *
1434 * Parameters:
1435 * cipherKey    User key
1436 * keyBits      AES key size (128, 192, or 256 bits)
1437 * keysched     AES key schedule to be initialized, of type aes_key_t.
1438 *              Allocated by aes_alloc_keysched().
1439 */
1440 void
1441 aes_init_keysched(const uint8_t *cipherKey, uint_t keyBits, void *keysched)
1442 {
1443 /* EXPORT DELETE START */
1444     aes_key_t    *newbie = keysched;
1445     uint_t        keysz, i, j;
1446     union {
1447         uint64_t    ka64[4];
1448         uint32_t    ka32[8];
1449     } keyarr;

1451     switch (keyBits) {
1452     case 128:
1453         newbie->nr = 10;
1454         break;

1456     case 192:
1457         newbie->nr = 12;
1458         break;

1460     case 256:
1461         newbie->nr = 14;
1462         break;

1464     default:
1465         /* should never get here */
1466         return;
1467     }
1468     keysz = keyBits >> 3;

1470     /*
1471     * For _LITTLE_ENDIAN machines (except AMD64), reverse every
1472     * 4 bytes in the key. On _BIG_ENDIAN and AMD64, copy the key
1473     * without reversing bytes.
1474     * For AMD64, do not byte swap for aes_setupkeys().
1475     *
1476     * SPARCv8/v9 uses a key schedule array with 64-bit elements.
1477     * X86/AMD64 uses a key schedule array with 32-bit elements.
1478     */
1479     #ifndef AES_BYTE_SWAP
1480     if (IS_P2ALIGNED(cipherKey, sizeof (uint64_t))) {
1481         for (i = 0, j = 0; j < keysz; i++, j += 8) {
1482             /* LINTED: pointer alignment */
1483             keyarr.ka64[i] = *((uint64_t *)&cipherKey[j]);
1484         }
1485     } else {
1486         bcopy(cipherKey, keyarr.ka32, keysz);
1487     }

1489 #else /* byte swap */
1490     for (i = 0, j = 0; j < keysz; i++, j += 4) {
1491         keyarr.ka32[i] = htonl(*(uint32_t *)&cipherKey[j]);
```

new/usr/src/common/crypto/aes/aes_impl.c

2

```
1491         keyarr.ka32[i] = htonl(*(uint32_t *)&cipherKey[j]);
1492     }
1493 #endif

1495     aes_setupkeys(newbie, keyarr.ka32, keyBits);
1496 /* EXPORT DELETE END */
1497 }

1499 /*
1500 * Encrypt one block using AES.
1501 * Align if needed and (for x86 32-bit only) byte-swap.
1502 *
1503 * Parameters:
1504 * ks    Key schedule, of type aes_key_t
1505 * pt    Input block (plain text)
1506 * ct    Output block (crypto text). Can overlap with pt
1507 */
1508 int
1509 aes_encrypt_block(const void *ks, const uint8_t *pt, uint8_t *ct)
1510 {
1511 /* EXPORT DELETE START */
1512     aes_key_t    *ksch = (aes_key_t *)ks;

1514     #ifndef AES_BYTE_SWAP
1515     if (IS_P2ALIGNED2(pt, ct, sizeof (uint32_t))) {
1516         AES_ENCRYPT_IMPL(&ksch->encr_ks.ks32[0], ksch->nr,
1517             /* LINTED: pointer alignment */
1518             (uint32_t *)pt, (uint32_t *)ct);
1519     } else {
1520     #endif
1521         uint32_t buffer[AES_BLOCK_LEN / sizeof (uint32_t)];

1523         /* Copy input block into buffer */
1524     #ifndef AES_BYTE_SWAP
1525         bcopy(pt, &buffer, AES_BLOCK_LEN);

1527     #else /* byte swap */
1528         buffer[0] = htonl(*(uint32_t *)&(void *)&pt[0]);
1529         buffer[1] = htonl(*(uint32_t *)&(void *)&pt[4]);
1530         buffer[2] = htonl(*(uint32_t *)&(void *)&pt[8]);
1531         buffer[3] = htonl(*(uint32_t *)&(void *)&pt[12]);
1528         buffer[0] = htonl(*(uint32_t *)&pt[0]);
1529         buffer[1] = htonl(*(uint32_t *)&pt[4]);
1530         buffer[2] = htonl(*(uint32_t *)&pt[8]);
1531         buffer[3] = htonl(*(uint32_t *)&pt[12]);
1532     #endif

1534         AES_ENCRYPT_IMPL(&ksch->encr_ks.ks32[0], ksch->nr,
1535             buffer, buffer);

1537         /* Copy result from buffer to output block */
1538     #ifndef AES_BYTE_SWAP
1539         bcopy(&buffer, ct, AES_BLOCK_LEN);
1540     }

1542     #else /* byte swap */
1543         *(uint32_t *)&(void *)&ct[0] = htonl(buffer[0]);
1544         *(uint32_t *)&(void *)&ct[4] = htonl(buffer[1]);
1545         *(uint32_t *)&(void *)&ct[8] = htonl(buffer[2]);
1546         *(uint32_t *)&(void *)&ct[12] = htonl(buffer[3]);
1543         *(uint32_t *)&ct[0] = htonl(buffer[0]);
1544         *(uint32_t *)&ct[4] = htonl(buffer[1]);
1545         *(uint32_t *)&ct[8] = htonl(buffer[2]);
1546         *(uint32_t *)&ct[12] = htonl(buffer[3]);
1547     #endif
1548 /* EXPORT DELETE END */
```

```

1549     return (CRYPTO_SUCCESS);
1550 }

1552 /*
1553  * Decrypt one block using AES.
1554  * Align and byte-swap if needed.
1555  * Parameters:
1556  *   ks    Key schedule, of type aes_key_t
1557  *   ct    Input block (crypto text)
1558  *   pt    Output block (plain text). Can overlap with ct
1559  */
1560 int
1561 aes_decrypt_block(const void *ks, const uint8_t *ct, uint8_t *pt)
1562 {
1563     /* EXPORT DELETE START */
1564     aes_key_t    *ksch = (aes_key_t *)ks;

1567 #ifndef AES_BYTE_SWAP
1568     if (IS_P2ALIGNED2(ct, pt, sizeof (uint32_t))) {
1569         AES_DECRYPT_IMPL(&ksch->decr_ks.ks32[0], ksch->nr,
1570             /* LINTED: pointer alignment */
1571             (uint32_t *)ct, (uint32_t *)pt);
1572     } else {
1573 #endif
1574         uint32_t buffer[AES_BLOCK_LEN / sizeof (uint32_t)];

1576         /* Copy input block into buffer */
1577 #ifndef AES_BYTE_SWAP
1578         bcopy(ct, &buffer, AES_BLOCK_LEN);

1580 #else /* byte swap */
1581         buffer[0] = htonl(*(uint32_t *) (void *)&ct[0]);
1582         buffer[1] = htonl(*(uint32_t *) (void *)&ct[4]);
1583         buffer[2] = htonl(*(uint32_t *) (void *)&ct[8]);
1584         buffer[3] = htonl(*(uint32_t *) (void *)&ct[12]);
1581         buffer[0] = htonl(*(uint32_t *)&ct[0]);
1582         buffer[1] = htonl(*(uint32_t *)&ct[4]);
1583         buffer[2] = htonl(*(uint32_t *)&ct[8]);
1584         buffer[3] = htonl(*(uint32_t *)&ct[12]);
1585 #endif

1587         AES_DECRYPT_IMPL(&ksch->decr_ks.ks32[0], ksch->nr,
1588             buffer, buffer);

1590         /* Copy result from buffer to output block */
1591 #ifndef AES_BYTE_SWAP
1592         bcopy(&buffer, pt, AES_BLOCK_LEN);
1593     }

1595 #else /* byte swap */
1596     *(uint32_t *) (void *)&pt[0] = htonl(buffer[0]);
1597     *(uint32_t *) (void *)&pt[4] = htonl(buffer[1]);
1598     *(uint32_t *) (void *)&pt[8] = htonl(buffer[2]);
1599     *(uint32_t *) (void *)&pt[12] = htonl(buffer[3]);
1596     *(uint32_t *)&pt[0] = htonl(buffer[0]);
1597     *(uint32_t *)&pt[4] = htonl(buffer[1]);
1598     *(uint32_t *)&pt[8] = htonl(buffer[2]);
1599     *(uint32_t *)&pt[12] = htonl(buffer[3]);
1600 #endif

1602 /* EXPORT DELETE END */
1603     return (CRYPTO_SUCCESS);
1604 }

```

unchanged portion omitted

new/usr/src/common/crypto/blowfish/blowfish_impl.c

1

```
*****
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_____

345 /*
346  * Since ROUND() is a macro, make sure that the things inside can be
347  * evaluated more than once. Especially when calling F().
348  * Assume the presence of local variables:
349  *
350  *     uint32_t *P;
351  *     uint32_t *S;
352  *     uint32_t tmp;
353  *
354  *
355  * And to Microsoft interview survivors out there, perhaps I should do the
356  * XOR swap trick, or at least #ifdef (__i386) the tmp = ... = tmp; stuff.
357  */

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

363 #define ROUND(left, right, i) \
364     (left) ^= P[i]; \
365     (right) ^= F((left)); \
366     tmp = (left); \
367     (left) = (right); \
368     (right) = tmp;

370 /* EXPORT DELETE END */

372 /*
373  * Encrypt a block of data. Because of addition operations, convert blocks
374  * to their big-endian representation, even on Intel boxen.
375  */
376 /* ARGSUSED */
377 int
378 blowfish_encrypt_block(const void *cookie, const uint8_t *block,
379     uint8_t *out_block)
380 {
381     /* EXPORT DELETE START */
382     keysched_t *ksch = (keysched_t *)cookie;

384     uint32_t left, right, tmp;
385     uint32_t *P = ksch->ksch_P;
386     uint32_t *S = ksch->ksch_S;
387     #ifdef _BIG_ENDIAN
388     uint32_t *b32;

390     if (IS_P2ALIGNED(block, sizeof (uint32_t))) {
391         /* LINTED: pointer alignment */
392         b32 = (uint32_t *)block;
393         left = b32[0];
394         right = b32[1];
395     } else
396     #endif
397     {
398         /*
399          * Read input block and place in left/right in big-endian order.
400          */
401     #ifdef UNALIGNED_POINTERS_PERMITTED
402         left = htonl(*(uint32_t *)(&block[0]));
403         right = htonl(*(uint32_t *)(&block[4]));
404         left = htonl(*(uint32_t *)(&block[0]));
405         right = htonl(*(uint32_t *)(&block[4]));
406     #else
407         left = ((uint32_t)block[0] << 24)
408             | ((uint32_t)block[1] << 16)
```

new/usr/src/common/crypto/blowfish/blowfish_impl.c

2

```
407     | ((uint32_t)block[2] << 8)
408     | (uint32_t)block[3];
409     right = ((uint32_t)block[4] << 24)
410     | ((uint32_t)block[5] << 16)
411     | ((uint32_t)block[6] << 8)
412     | (uint32_t)block[7];
413 #endif /* UNALIGNED_POINTERS_PERMITTED */
414 }

416     ROUND(left, right, 0);
417     ROUND(left, right, 1);
418     ROUND(left, right, 2);
419     ROUND(left, right, 3);
420     ROUND(left, right, 4);
421     ROUND(left, right, 5);
422     ROUND(left, right, 6);
423     ROUND(left, right, 7);
424     ROUND(left, right, 8);
425     ROUND(left, right, 9);
426     ROUND(left, right, 10);
427     ROUND(left, right, 11);
428     ROUND(left, right, 12);
429     ROUND(left, right, 13);
430     ROUND(left, right, 14);
431     ROUND(left, right, 15);

433     tmp = left;
434     left = right;
435     right = tmp;
436     right ^= P[16];
437     left ^= P[17];

439 #ifdef _BIG_ENDIAN
440     if (IS_P2ALIGNED(out_block, sizeof (uint32_t))) {
441         /* LINTED: pointer alignment */
442         b32 = (uint32_t *)out_block;
443         b32[0] = left;
444         b32[1] = right;
445     } else
446     #endif
447     {
448         /* Put the block back into the user's block with final swap */
449     #ifdef UNALIGNED_POINTERS_PERMITTED
450         *(uint32_t *)(&out_block[0]) = htonl(left);
451         *(uint32_t *)(&out_block[4]) = htonl(right);
452         *(uint32_t *)(&out_block[0]) = htonl(left);
453         *(uint32_t *)(&out_block[4]) = htonl(right);
454     #else
455         out_block[0] = left >> 24;
456         out_block[1] = left >> 16;
457         out_block[2] = left >> 8;
458         out_block[3] = left;
459         out_block[4] = right >> 24;
460         out_block[5] = right >> 16;
461         out_block[6] = right >> 8;
462         out_block[7] = right;
463     #endif /* UNALIGNED_POINTERS_PERMITTED */
464 }
465 /* EXPORT DELETE END */
466 return (CRYPTO_SUCCESS);
467 }

468 /*
469  * Decrypt a block of data. Because of addition operations, convert blocks
470  * to their big-endian representation, even on Intel boxen.
471  * It should look like the blowfish_encrypt_block() operation
```

```

471  * except for the order in which the S/P boxes are accessed.
472  */
473  /* ARGSUSED */
474  int
475  blowfish_decrypt_block(const void *cookie, const uint8_t *block,
476                        uint8_t *out_block)
477  {
478  /* EXPORT DELETE START */
479      keysched_t *ksch = (keysched_t *)cookie;

481      uint32_t left, right, tmp;
482      uint32_t *P = ksch->ksch_P;
483      uint32_t *S = ksch->ksch_S;
484  #ifdef _BIG_ENDIAN
485      uint32_t *b32;

487      if (IS_P2ALIGNED(block, sizeof (uint32_t))) {
488          /* LINTED: pointer alignment */
489          b32 = (uint32_t *)block;
490          left = b32[0];
491          right = b32[1];
492      } else
493  #endif
494      {
495          /*
496           * Read input block and place in left/right in big-endian order.
497           */
498  #ifdef UNALIGNED_POINTERS_PERMITTED
499          left = htonl(*(uint32_t *)(&block[0]));
500          right = htonl(*(uint32_t *)(&block[4]));
501          left = htonl(*(uint32_t *)(&block[0]));
502          right = htonl(*(uint32_t *)(&block[4]));
503  #else
504          left = ((uint32_t)block[0] << 24)
505                | ((uint32_t)block[1] << 16)
506                | ((uint32_t)block[2] << 8)
507                | (uint32_t)block[3];
508          right = ((uint32_t)block[4] << 24)
509                 | ((uint32_t)block[5] << 16)
510                 | ((uint32_t)block[6] << 8)
511                 | (uint32_t)block[7];
512  #endif /* UNALIGNED_POINTERS_PERMITTED */
513      }

514      ROUND(left, right, 17);
515      ROUND(left, right, 16);
516      ROUND(left, right, 15);
517      ROUND(left, right, 14);
518      ROUND(left, right, 13);
519      ROUND(left, right, 12);
520      ROUND(left, right, 11);
521      ROUND(left, right, 10);
522      ROUND(left, right, 9);
523      ROUND(left, right, 8);
524      ROUND(left, right, 7);
525      ROUND(left, right, 6);
526      ROUND(left, right, 5);
527      ROUND(left, right, 4);
528      ROUND(left, right, 3);
529      ROUND(left, right, 2);

530      tmp = left;
531      left = right;
532      right = tmp;
533      right ^= P[1];
534      left ^= P[0];

```

```

536  #ifdef _BIG_ENDIAN
537      if (IS_P2ALIGNED(out_block, sizeof (uint32_t))) {
538          /* LINTED: pointer alignment */
539          b32 = (uint32_t *)out_block;
540          b32[0] = left;
541          b32[1] = right;
542      } else
543  #endif
544      {
545          /* Put the block back into the user's block with final swap */
546  #ifdef UNALIGNED_POINTERS_PERMITTED
547          *(uint32_t *)(&out_block[0]) = htonl(left);
548          *(uint32_t *)(&out_block[4]) = htonl(right);
549          *(uint32_t *)(&out_block[0]) = htonl(right);
550          *(uint32_t *)(&out_block[4]) = htonl(left);
551  #else
552          out_block[0] = left >> 24;
553          out_block[1] = left >> 16;
554          out_block[2] = left >> 8;
555          out_block[3] = left;
556          out_block[4] = right >> 24;
557          out_block[5] = right >> 16;
558          out_block[6] = right >> 8;
559          out_block[7] = right;
560  #endif /* UNALIGNED_POINTERS_PERMITTED */
561      }
562  /* EXPORT DELETE END */
563      return (CRYPTO_SUCCESS);
564  }

```

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
502 #endif /* !sun4u */

504 /* EXPORT DELETE END */

506 int
507 des3_crunch_block(const void *cookie, const uint8_t block[DES_BLOCK_LEN],
508                  uint8_t out_block[DES_BLOCK_LEN], boolean_t decrypt)
509 {
510 /* EXPORT DELETE START */
511     keysched3_t *ksch = (keysched3_t *)cookie;
512
513     /*
514      * The code below, that is always executed on LITTLE_ENDIAN machines,
515      * reverses bytes in the block. On BIG_ENDIAN, the same code
516      * copies the block without reversing bytes.
517      */
518 #ifdef _BIG_ENDIAN
519     if (IS_P2ALIGNED(block, sizeof (uint64_t)) &&
520         IS_P2ALIGNED(out_block, sizeof (uint64_t))) {
521         if (decrypt == B_TRUE)
522             /* LINTED */
523             *(uint64_t *)out_block = des_crypt_impl(
524                 ksch->ksch_decrypt, /* LINTED */
525                 *(uint64_t *)block, 3);
526         else
527             /* LINTED */
528             *(uint64_t *)out_block = des_crypt_impl(
529                 ksch->ksch_encrypt, /* LINTED */
530                 *(uint64_t *)block, 3);
531     } else
532 #endif /* _BIG_ENDIAN */
533     {
534         uint64_t tmp;
535
536 #ifdef UNALIGNED_POINTERS_PERMITTED
537         tmp = htonll(*(uint64_t *)(&block[0]));
538         tmp = htonll(*(uint64_t *)(&block[0]));
539 #else
540         tmp = (((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
541              ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
542              ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
543              ((uint64_t)block[6] << 8) | (uint64_t)block[7]);
544 #endif /* UNALIGNED_POINTERS_PERMITTED */
545
546         if (decrypt == B_TRUE)
547             tmp = des_crypt_impl(ksch->ksch_decrypt, tmp, 3);
548         else
549             tmp = des_crypt_impl(ksch->ksch_encrypt, tmp, 3);
550
551 #ifdef UNALIGNED_POINTERS_PERMITTED
552         *(uint64_t *)(&out_block[0]) = htonll(tmp);
553         *(uint64_t *)(&out_block[0]) = htonll(tmp);
554 #else
555         out_block[0] = tmp >> 56;
556         out_block[1] = tmp >> 48;
557         out_block[2] = tmp >> 40;
558         out_block[3] = tmp >> 32;
559         out_block[4] = tmp >> 24;
560         out_block[5] = tmp >> 16;
561         out_block[6] = tmp >> 8;
562         out_block[7] = (uint8_t)tmp;
563 #endif /* UNALIGNED_POINTERS_PERMITTED */
564     }
565
566 /* EXPORT DELETE END */
567     return (CRYPTO_SUCCESS);

```

```

565 }

567 int
568 des_crunch_block(const void *cookie, const uint8_t block[DES_BLOCK_LEN],
569                 uint8_t out_block[DES_BLOCK_LEN], boolean_t decrypt)
570 {
571 /* EXPORT DELETE START */
572     keysched_t *ksch = (keysched_t *)cookie;
573
574     /*
575      * The code below, that is always executed on LITTLE_ENDIAN machines,
576      * reverses bytes in the block. On BIG_ENDIAN, the same code
577      * copies the block without reversing bytes.
578      */
579 #ifdef _BIG_ENDIAN
580     if (IS_P2ALIGNED(block, sizeof (uint64_t)) &&
581         IS_P2ALIGNED(out_block, sizeof (uint64_t))) {
582         if (decrypt == B_TRUE)
583             /* LINTED */
584             *(uint64_t *)out_block = des_crypt_impl(
585                 ksch->ksch_decrypt, /* LINTED */
586                 *(uint64_t *)block, 1);
587         else
588             /* LINTED */
589             *(uint64_t *)out_block = des_crypt_impl(
590                 ksch->ksch_encrypt, /* LINTED */
591                 *(uint64_t *)block, 1);
592     } else
593 #endif /* _BIG_ENDIAN */
594     {
595         uint64_t tmp;
596
597 #ifdef UNALIGNED_POINTERS_PERMITTED
598         tmp = htonll(*(uint64_t *)(&block[0]));
599         tmp = htonll(*(uint64_t *)(&block[0]));
600 #else
601         tmp = (((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
602              ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
603              ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
604              ((uint64_t)block[6] << 8) | (uint64_t)block[7]);
605 #endif /* UNALIGNED_POINTERS_PERMITTED */
606
607         if (decrypt == B_TRUE)
608             tmp = des_crypt_impl(ksch->ksch_decrypt, tmp, 1);
609         else
610             tmp = des_crypt_impl(ksch->ksch_encrypt, tmp, 1);
611
612 #ifdef UNALIGNED_POINTERS_PERMITTED
613         *(uint64_t *)(&out_block[0]) = htonll(tmp);
614         *(uint64_t *)(&out_block[0]) = htonll(tmp);
615 #else
616         out_block[0] = tmp >> 56;
617         out_block[1] = tmp >> 48;
618         out_block[2] = tmp >> 40;
619         out_block[3] = tmp >> 32;
620         out_block[4] = tmp >> 24;
621         out_block[5] = tmp >> 16;
622         out_block[6] = tmp >> 8;
623         out_block[7] = (uint8_t)tmp;
624 #endif /* UNALIGNED_POINTERS_PERMITTED */
625     }
626
627 /* EXPORT DELETE END */
628     return (CRYPTO_SUCCESS);

```

```

630 static boolean_t
631 keycheck(uint8_t *key, uint8_t *corrected_key)
632 {
633 /* EXPORT DELETE START */
634     uint64_t key_so_far;
635     uint_t i;
636     /*
637      * Table of weak and semi-weak keys. Fortunately, weak keys are
638      * endian-independent, and some semi-weak keys can be paired up in
639      * endian-opposite order. Since keys are stored as uint64_t's,
640      * use the ifdef _LITTLE_ENDIAN where appropriate.
641      */
642     static uint64_t des_weak_keys[] = {
643         /* Really weak keys. Byte-order independent values. */
644         0x0101010101010101ULL,
645         0x1f1f1f1f0e0e0e0eULL,
646         0xe0e0e0e0f1f1f1f1ULL,
647         0xfefefefefefefefefULL,
648
649         /* Semi-weak (and a few possibly-weak) keys. */
650
651         /* Byte-order independent semi-weak keys. */
652         0x01fe01fe01fe01feULL, 0xfe01fe01fe01feULL,
653
654         /* Byte-order dependent semi-weak keys. */
655 #ifdef _LITTLE_ENDIAN
656         0xf10ef10ee01fe01fULL, 0x0ef10ef11fe01fe0ULL,
657         0x01f101f101e001e0ULL, 0xf101f101e001e0ULL,
658         0x0efe0efe1ffe1ffeULL, 0xfe0efe0efe1ffe1fULL,
659         0x010e010e011f011fULL, 0x0e010e011f011f01ULL,
660         0xf1fef1fee0fee0fee0ULL, 0xfef1fef1fee0fee0ULL,
661 #else /* Big endian */
662         0x1fe01fe00ef10ef1ULL, 0xe01fe01ff10ef10eULL,
663         0x01e001e001f101f1ULL, 0xe001e001f101f101ULL,
664         0x1ffe1ffe0efe0efeULL, 0xfe1ffe1ffe0efe0eULL,
665         0x011f011f010e010eULL, 0x1f011f010e010e01ULL,
666         0xe0fee0fef1fef1feULL, 0xfef0fee0fef1fef1ULL,
667 #endif /* _LITTLE_ENDIAN */
668
669         /* We'll save the other possibly-weak keys for the future. */
670     };
671
672     if (key == NULL)
673         return (B_FALSE);
674
675 #ifdef UNALIGNED_POINTERS_PERMITTED
676     key_so_far = htonl(*(uint64_t *) (void *) &key[0]);
677 #else
678     /*
679      * The code below reverses the bytes on LITTLE_ENDIAN machines.
680      * On BIG_ENDIAN, the same code copies without reversing
681      * the bytes.
682      */
683     key_so_far = (((uint64_t)key[0] << 56) | ((uint64_t)key[1] << 48) |
684         ((uint64_t)key[2] << 40) | ((uint64_t)key[3] << 32) |
685         ((uint64_t)key[4] << 24) | ((uint64_t)key[5] << 16) |
686         ((uint64_t)key[6] << 8) | (uint64_t)key[7]);
687 #endif /* UNALIGNED_POINTERS_PERMITTED */
688
689     /*
690      * Fix parity.
691      */
692     fix_des_parity(&key_so_far);

```

```

694     /* Do weak key check itself. */
695     for (i = 0; i < (sizeof (des_weak_keys) / sizeof (uint64_t)); i++)
696         if (key_so_far == des_weak_keys[i]) {
697             return (B_FALSE);
698         }
699
700     if (corrected_key != NULL) {
701 #ifdef UNALIGNED_POINTERS_PERMITTED
702         *(uint64_t *) (void *) &corrected_key[0] = htonl(key_so_far);
703 #else
704         /*
705          * The code below reverses the bytes on LITTLE_ENDIAN machines.
706          * On BIG_ENDIAN, the same code copies without reversing
707          * the bytes.
708          */
709         corrected_key[0] = key_so_far >> 56;
710         corrected_key[1] = key_so_far >> 48;
711         corrected_key[2] = key_so_far >> 40;
712         corrected_key[3] = key_so_far >> 32;
713         corrected_key[4] = key_so_far >> 24;
714         corrected_key[5] = key_so_far >> 16;
715         corrected_key[6] = key_so_far >> 8;
716         corrected_key[7] = (uint8_t)key_so_far;
717 #endif /* UNALIGNED_POINTERS_PERMITTED */
718     }
719     /* EXPORT DELETE END */
720     return (B_TRUE);
721 }
722
723 void
724 des_parity_fix(uint8_t *key, des_strength_t strength, uint8_t *corrected_key)
725 {
726 /* EXPORT DELETE START */
727     uint64_t aligned_key[DES3_KEYSIZE / sizeof (uint64_t)];
728     uint8_t *paritied_key;
729     uint64_t key_so_far;
730     int i = 0, offset = 0;
731
732     if (strength == DES)
733         bcopy(key, aligned_key, DES_KEYSIZE);
734     else
735         bcopy(key, aligned_key, DES3_KEYSIZE);
736
737     paritied_key = (uint8_t *)aligned_key;
738     while (strength > i) {
739         offset = 8 * i;
740 #ifdef UNALIGNED_POINTERS_PERMITTED
741         key_so_far = htonl(*(uint64_t *) (void *) &paritied_key[offset]);
742 #else
743         key_so_far = htonl(*(uint64_t *) &paritied_key[offset]);
744 #endif
745         key_so_far = (((uint64_t)paritied_key[offset + 0] << 56) |
746             ((uint64_t)paritied_key[offset + 1] << 48) |
747             ((uint64_t)paritied_key[offset + 2] << 40) |
748             ((uint64_t)paritied_key[offset + 3] << 32) |
749             ((uint64_t)paritied_key[offset + 4] << 24) |
750             ((uint64_t)paritied_key[offset + 5] << 16) |
751             ((uint64_t)paritied_key[offset + 6] << 8) |
752             (uint64_t)paritied_key[offset + 7]);
753 #endif /* UNALIGNED_POINTERS_PERMITTED */
754
755         fix_des_parity(&key_so_far);
756
757 #ifdef UNALIGNED_POINTERS_PERMITTED
758         *(uint64_t *) (void *) &paritied_key[offset] = htonl(key_so_far);

```

```

826         *(uint64_t *)&paritied_key[offset] = htonl(key_so_far);
827 #else
828         paritied_key[offset + 0] = key_so_far >> 56;
829         paritied_key[offset + 1] = key_so_far >> 48;
830         paritied_key[offset + 2] = key_so_far >> 40;
831         paritied_key[offset + 3] = key_so_far >> 32;
832         paritied_key[offset + 4] = key_so_far >> 24;
833         paritied_key[offset + 5] = key_so_far >> 16;
834         paritied_key[offset + 6] = key_so_far >> 8;
835         paritied_key[offset + 7] = (uint8_t)key_so_far;
836 #endif /* UNALIGNED_POINTERS_PERMITTED */

838         i++;
839     }

841     bcopy(paritied_key, corrected_key, DES_KEYSIZE * strength);
842 /* EXPORT DELETE END */
843 }

846 /*
847  * Initialize key schedule for DES, DES2, and DES3
848  */
849 void
850 des_init_keysched(uint8_t *cipherKey, des_strength_t strength, void *ks)
851 {
852 /* EXPORT DELETE START */
853     uint64_t *encryption_ks;
854     uint64_t *decryption_ks;
855     uint64_t keysched[48];
856     uint64_t key_uint64[3];
857     uint64_t tmp;
858     uint_t keysize, i, j;

860     switch (strength) {
861     case DES:
862         keysize = DES_KEYSIZE;
863         encryption_ks = ((keysched_t *)ks)->ksch_encrypt;
864         decryption_ks = ((keysched_t *)ks)->ksch_decrypt;
865         break;
866     case DES2:
867         keysize = DES2_KEYSIZE;
868         encryption_ks = ((keysched3_t *)ks)->ksch_encrypt;
869         decryption_ks = ((keysched3_t *)ks)->ksch_decrypt;
870         break;
871     case DES3:
872         keysize = DES3_KEYSIZE;
873         encryption_ks = ((keysched3_t *)ks)->ksch_encrypt;
874         decryption_ks = ((keysched3_t *)ks)->ksch_decrypt;
875     }

877     /*
878     * The code below, that is always executed on LITTLE_ENDIAN machines,
879     * reverses every 8 bytes in the key. On BIG_ENDIAN, the same code
880     * copies the key without reversing bytes.
881     */
882 #ifdef _BIG_ENDIAN
883     if (IS_P2ALIGNED(cipherKey, sizeof (uint64_t))) {
884         for (i = 0, j = 0; j < keysize; i++, j += 8) {
885             /* LINTED: pointer alignment */
886             key_uint64[i] = *((uint64_t *)&cipherKey[j]);
887         }
888     } else
889 #endif /* _BIG_ENDIAN */
890     {
891         for (i = 0, j = 0; j < keysize; i++, j += 8) {

```

```

892 #ifdef UNALIGNED_POINTERS_PERMITTED
893     key_uint64[i] =
894         htonl(*(uint64_t *)(void *)&cipherKey[j]);
895     key_uint64[i] = htonl(*(uint64_t *)&cipherKey[j]);
896 #else
897     key_uint64[i] = (((uint64_t)cipherKey[j] << 56) |
898         ((uint64_t)cipherKey[j + 1] << 48) |
899         ((uint64_t)cipherKey[j + 2] << 40) |
900         ((uint64_t)cipherKey[j + 3] << 32) |
901         ((uint64_t)cipherKey[j + 4] << 24) |
902         ((uint64_t)cipherKey[j + 5] << 16) |
903         ((uint64_t)cipherKey[j + 6] << 8) |
904         (uint64_t)cipherKey[j + 7]);
905 #endif /* UNALIGNED_POINTERS_PERMITTED */
906 }

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

913     case DES2:
914         /* DES2 is just DES3 with the first and third keys the same */
915         bcopy(key_uint64, key_uint64 + 2, DES_KEYSIZE);
916         /* FALLTHRU */
917     case DES3:
918         des_ks(keysched, key_uint64[0]);
919         des_ks(keysched + 16, key_uint64[1]);
920         for (i = 0; i < 8; i++) {
921             tmp = keysched[16+i];
922             keysched[16+i] = keysched[31-i];
923             keysched[31-i] = tmp;
924         }
925         des_ks(keysched+32, key_uint64[2]);
926         keysize = DES3_KEYSIZE;
927     }

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

932     /* reverse the key schedule */
933     for (i = 0; i < keysize; i++) {
934         tmp = keysched[i];
935         keysched[i] = keysched[2 * keysize - 1 - i];
936         keysched[2 * keysize - 1 - i] = tmp;
937     }

939     /* save the decryption keyschedule */
940     bcopy(keysched, decryption_ks, keysize * 16);
941 /* EXPORT DELETE END */
942 }

```

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_____

```
262 /*
263  * Encodes input (uint32_t) into output (unsigned char). Assumes len is
264  * a multiple of 4.
265  */
266 static void
267 Encode(unsigned char *output, uint32_t *input, unsigned int len)
268 {
269     unsigned int i, j;
```

```
    for (i = 0, j = 0; j < len; i++, j += 4) {
272 #if defined(_LITTLE_ENDIAN) && defined(UNALIGNED_POINTERS_PERMITTED)
273     *(uint32_t *)(&output[j]) = input[i];
273     *(uint32_t *)(&output[j]) = input[i];
274 #else
275     /* endian-independent code */
276     output[j] = (unsigned char)(input[i] & 0xff);
277     output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
278     output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
279     output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
280 #endif /* _LITTLE_ENDIAN && UNALIGNED_POINTERS_PERMITTED */
281     }
282 }
```

```
284 /*
285  * Decodes input (unsigned char) into output (uint32_t). Assumes len is
286  * a multiple of 4.
287  */
288 static void
289 Decode(uint32_t *output, unsigned char *input, unsigned int len)
290 {
291     unsigned int i, j;
```

```
    for (i = 0, j = 0; j < len; i++, j += 4) {
294 #if defined(_LITTLE_ENDIAN) && defined(UNALIGNED_POINTERS_PERMITTED)
295     output[i] = *(uint32_t *)(&input[j]);
295     output[i] = *(uint32_t *)(&input[j]);
296 #else
297     /* endian-independent code */
298     output[i] = ((uint32_t)input[j]) |
299     (((uint32_t)input[j+1]) << 8) |
300     (((uint32_t)input[j+2]) << 16) |
301     (((uint32_t)input[j+3]) << 24);
302 #endif /* _LITTLE_ENDIAN && UNALIGNED_POINTERS_PERMITTED */
303     }
```

```
305 }
_____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_____

358 /*
359  * This will decrypt the cipher text.  However, the plaintext won't be
360  * returned to the caller.  It will be returned when decrypt_final() is
361  * called if the MAC matches
362  */
363 /* ARGSUSED */
364 int
365 ccm_mode_decrypt_contiguous_blocks(ccm_ctx_t *ctx, char *data, size_t length,
366     crypto_data_t *out, size_t block_size,
367     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
368     void (*copy_block)(uint8_t *, uint8_t *),
369     void (*xor_block)(uint8_t *, uint8_t *))
370 {
371     size_t remainder = length;
372     size_t need;
373     uint8_t *datap = (uint8_t *)data;
374     uint8_t *blockp;
375     uint8_t *cbp;
376     uint64_t counter;
377     size_t pt_len, total_decrypted_len, mac_len, pm_len, pd_len;
378     uint8_t *resultp;
379 #ifdef _LITTLE_ENDIAN
380     uint8_t *p;
381 #endif /* _LITTLE_ENDIAN */

381     pm_len = ctx->ccm_processed_mac_len;

383     if (pm_len > 0) {
384         uint8_t *tmp;
385         /*
386          * all ciphertext has been processed, just waiting for
387          * part of the value of the mac
388          */
389         if ((pm_len + length) > ctx->ccm_mac_len) {
390             return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
391         }
392         tmp = (uint8_t *)ctx->ccm_mac_input_buf;
394         bcopy(datap, tmp + pm_len, length);

396         ctx->ccm_processed_mac_len += length;
397         return (CRYPTO_SUCCESS);
398     }

400     /*
401     * If we decrypt the given data, what total amount of data would
402     * have been decrypted?
403     */
404     pd_len = ctx->ccm_processed_data_len;
405     total_decrypted_len = pd_len + length + ctx->ccm_remainder_len;

407     if (total_decrypted_len >
408         (ctx->ccm_data_len + ctx->ccm_mac_len)) {
409         return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
410     }

412     pt_len = ctx->ccm_data_len;

414     if (total_decrypted_len > pt_len) {
415         /*
416          * part of the input will be the MAC, need to isolate that
417          * to be dealt with later.  The left-over data in
418          * ccm_remainder_len from last time will not be part of the

```

```

419     * MAC.  Otherwise, it would have already been taken out
420     * when this call is made last time.
421     */
422     size_t pt_part = pt_len - pd_len - ctx->ccm_remainder_len;

424     mac_len = length - pt_part;

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

429     if (pt_part + ctx->ccm_remainder_len < block_size) {
430         /*
431          * since this is last of the ciphertext, will
432          * just decrypt with it here
433          */
434         bcopy(datap, &((uint8_t *)ctx->ccm_remainder)
435             [ctx->ccm_remainder_len], pt_part);
436         ctx->ccm_remainder_len += pt_part;
437         ccm_decrypt_incomplete_block(ctx, encrypt_block);
438         ctx->ccm_remainder_len = 0;
439         ctx->ccm_processed_data_len += pt_part;
440         return (CRYPTO_SUCCESS);
441     } else {
442         /* let rest of the code handle this */
443         length = pt_part;
444     }
445     } else if (length + ctx->ccm_remainder_len < block_size) {
446         /* accumulate bytes here and return */
447         bcopy(datap,
448             (uint8_t *)ctx->ccm_remainder + ctx->ccm_remainder_len,
449             length);
450         ctx->ccm_remainder_len += length;
451         ctx->ccm_copy_to = datap;
452         return (CRYPTO_SUCCESS);
453     }

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

460             if (need > remainder)
461                 return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);

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

466             blockp = (uint8_t *)ctx->ccm_remainder;
467         } else {
468             blockp = datap;
469         }

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

475         /*
476          * Increment counter.
477          * Counter bits are confined to the bottom 64 bits
478          */
479 #ifdef _LITTLE_ENDIAN
480         counter = ntohl(ctx->ccm_cb[1] & ctx->ccm_counter_mask);
481         counter = htonl(counter + 1);
482 #else
483         counter = ctx->ccm_cb[1] & ctx->ccm_counter_mask;
484         counter++;

```

```
485 #endif /* _LITTLE_ENDIAN */
486         counter &= ctx->ccm_counter_mask;
487         ctx->ccm_cb[1] =
488             (ctx->ccm_cb[1] & ~(ctx->ccm_counter_mask)) | counter;
489
490         /* XOR with the ciphertext */
491         xor_block(blockp, cbp);
492
493         /* Copy the plaintext to the "holding buffer" */
494         resultp = (uint8_t *)ctx->ccm_pt_buf +
495             ctx->ccm_processed_data_len;
496         copy_block(cbp, resultp);
497
498         ctx->ccm_processed_data_len += block_size;
499
500         ctx->ccm_lastp = blockp;
501
502         /* Update pointer to next block of data to be processed. */
503         if (ctx->ccm_remainder_len != 0) {
504             datap += need;
505             ctx->ccm_remainder_len = 0;
506         } else {
507             datap += block_size;
508         }
509
510         remainder = (size_t)&data[length] - (size_t)datap;
511
512         /* Incomplete last block */
513         if (remainder > 0 && remainder < block_size) {
514             bcopy(datap, ctx->ccm_remainder, remainder);
515             ctx->ccm_remainder_len = remainder;
516             ctx->ccm_copy_to = datap;
517             if (ctx->ccm_processed_mac_len > 0) {
518                 /*
519                  * not expecting anymore ciphertext, just
520                  * compute plaintext for the remaining input
521                  */
522                 ccm_decrypt_incomplete_block(ctx,
523                     encrypt_block);
524                 ctx->ccm_processed_data_len += remainder;
525                 ctx->ccm_remainder_len = 0;
526             }
527             goto out;
528         }
529         ctx->ccm_copy_to = NULL;
530
531     } while (remainder > 0);
532
533 out:
534     return (CRYPTO_SUCCESS);
535 }
```

unchanged_portion_omitted

new/usr/src/common/crypto/modes/ctr.c

1

```
*****
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
*****
```

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
```

```
26 #ifndef _KERNEL
27 #include <strings.h>
28 #include <limits.h>
29 #include <assert.h>
30 #include <security/cryptoki.h>
31 #endif
```

```
33 #include <sys/types.h>
34 #include <modes/modes.h>
35 #include <sys/crypto/common.h>
36 #include <sys/crypto/impl.h>
```

```
38 #ifdef _LITTLE_ENDIAN
39 #include <sys/byteorder.h>
40 #endif
```

```
42 /*
43  * Encrypt and decrypt multiple blocks of data in counter mode.
44  */
45 int
46 ctr_mode_contiguous_blocks(ctr_ctx_t *ctx, char *data, size_t length,
47     crypto_data_t *out, size_t block_size,
48     int (*cipher)(const void *ks, const uint8_t *pt, uint8_t *ct),
49     void (*xor_block)(uint8_t *, uint8_t *))
50 {
51     size_t remainder = length;
52     size_t need;
53     uint8_t *datap = (uint8_t *)data;
54     uint8_t *blockp;
55     uint8_t *lastp;
56     void *iov_or_mp;
57     offset_t offset;
58     uint8_t *out_data_1;
59     uint8_t *out_data_2;
60     size_t out_data_1_len;
61     uint64_t counter;
```

new/usr/src/common/crypto/modes/ctr.c

2

```
62 #ifdef _LITTLE_ENDIAN
63     uint8_t *p;
64 #endif

63     if (length + ctx->ctr_remainder_len < block_size) {
64         /* accumulate bytes here and return */
65         bcopy(datap,
66             (uint8_t *)ctx->ctr_remainder + ctx->ctr_remainder_len,
67             length);
68         ctx->ctr_remainder_len += length;
69         ctx->ctr_copy_to = datap;
70         return (CRYPTO_SUCCESS);
71     }

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

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

82             if (need > remainder)
83                 return (CRYPTO_DATA_LEN_RANGE);

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

88             blockp = (uint8_t *)ctx->ctr_remainder;
89         } else {
90             blockp = datap;
91         }

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

97         lastp = (uint8_t *)ctx->ctr_tmp;

99         /*
100          * Increment counter. Counter bits are confined
101          * to the bottom 64 bits of the counter block.
102          */
103 #ifdef _LITTLE_ENDIAN
104         counter = ntohl(ctx->ctr_cb[1] & ctx->ctr_counter_mask);
105         counter = htonl(counter + 1);
106 #else
107         counter = ctx->ctr_cb[1] & ctx->ctr_counter_mask;
108         counter++;
109 #endif /* _LITTLE_ENDIAN */
110         counter &= ctx->ctr_counter_mask;
111         ctx->ctr_cb[1] =
112             (ctx->ctr_cb[1] & ~(ctx->ctr_counter_mask)) | counter;

114         /*
115          * XOR the previous cipher block or IV with the
116          * current clear block.
117          */
118         xor_block(blockp, lastp);

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

```
125         need);
126     }
127 } else {
128     crypto_get_ptrs(out, &iiov_or_mp, &offset, &out_data_1,
129                     &out_data_1_len, &out_data_2, block_size);
130
131     /* copy block to where it belongs */
132     bcopy(lastp, out_data_1, out_data_1_len);
133     if (out_data_2 != NULL) {
134         bcopy(lastp + out_data_1_len, out_data_2,
135             block_size - out_data_1_len);
136     }
137     /* update offset */
138     out->cd_offset += block_size;
139 }
140
141 /* Update pointer to next block of data to be processed. */
142 if (ctx->ctr_remainder_len != 0) {
143     datap += need;
144     ctx->ctr_remainder_len = 0;
145 } else {
146     datap += block_size;
147 }
148
149 remainder = (size_t)&data[length] - (size_t)datap;
150
151 /* Incomplete last block. */
152 if (remainder > 0 && remainder < block_size) {
153     bcopy(datap, ctx->ctr_remainder, remainder);
154     ctx->ctr_remainder_len = remainder;
155     ctx->ctr_copy_to = datap;
156     goto out;
157 }
158 ctx->ctr_copy_to = NULL;
159
160 } while (remainder > 0);
161
162 out:
163     return (CRYPTO_SUCCESS);
164 }
```

unchanged_portion_omitted

new/usr/src/common/crypto/sha1/sha1.c

1

```
*****
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_____

436 #if !defined(__amd64)

438 typedef uint32_t sha1word;

440 /*
441  * sparc optimization:
442  *
443  * on the sparc, we can load big endian 32-bit data easily. note that
444  * special care must be taken to ensure the address is 32-bit aligned.
445  * in the interest of speed, we don't check to make sure, since
446  * careful programming can guarantee this for us.
447  */

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

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

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

461 /*
462  * SHA1Transform()
463  */
464 #if defined(W_ARRAY)
465 #define W(n) w[n]
466 #else /* !defined(W_ARRAY) */
467 #define W(n) w_ ## n
468 #endif /* !defined(W_ARRAY) */

471 #if defined(__sparc)

473 /*
474  * sparc register window optimization:
475  *
476  * 'a', 'b', 'c', 'd', and 'e' are passed into SHA1Transform
477  * explicitly since it increases the number of registers available to
478  * the compiler. under this scheme, these variables can be held in
479  * %i0 - %i4, which leaves more local and out registers available.
480  *
481  * purpose: sha1 transformation -- updates the digest based on 'block'
482  * input:  uint32_t      : bytes 1 - 4 of the digest
483  *          uint32_t      : bytes 5 - 8 of the digest
484  *          uint32_t      : bytes 9 - 12 of the digest
485  *          uint32_t      : bytes 12 - 16 of the digest
486  *          uint32_t      : bytes 16 - 20 of the digest
487  *          SHA1_CTX *    : the context to update
488  *          uint8_t [64] : the block to use to update the digest
489  * output: void
490  */

492 void
493 SHA1Transform(uint32_t a, uint32_t b, uint32_t c, uint32_t d, uint32_t e,
494              SHA1_CTX *ctx, const uint8_t blk[64])
495 {
496     /*
497      * sparc optimization:
498      */

```

new/usr/src/common/crypto/sha1/sha1.c

2

```
499     * while it is somewhat counter-intuitive, on sparc, it is
500     * more efficient to place all the constants used in this
501     * function in an array and load the values out of the array
502     * than to manually load the constants. this is because
503     * setting a register to a 32-bit value takes two ops in most
504     * cases: a 'sethi' and an 'or', but loading a 32-bit value
505     * from memory only takes one 'ld' (or 'lduw' on v9). while
506     * this increases memory usage, the compiler can find enough
507     * other things to do while waiting to keep the pipeline does
508     * not stall. additionally, it is likely that many of these
509     * constants are cached so that later accesses do not even go
510     * out to the bus.
511     *
512     * this array is declared 'static' to keep the compiler from
513     * having to bcopy() this array onto the stack frame of
514     * SHA1Transform() each time it is called -- which is
515     * unacceptably expensive.
516     *
517     * the 'const' is to ensure that callers are good citizens and
518     * do not try to munge the array. since these routines are
519     * going to be called from inside multithreaded kernelland,
520     * this is a good safety check. -- 'sha1_consts' will end up in
521     * .rodata.
522     *
523     * unfortunately, loading from an array in this manner hurts
524     * performance under Intel. So, there is a macro,
525     * SHA1_CONST(), used in SHA1Transform(), that either expands to
526     * a reference to this array, or to the actual constant,
527     * depending on what platform this code is compiled for.
528     */

530     static const uint32_t sha1_consts[] = {
531         SHA1_CONST_0, SHA1_CONST_1, SHA1_CONST_2, SHA1_CONST_3
532     };

534     /*
535      * general optimization:
536      *
537      * use individual integers instead of using an array. this is a
538      * win, although the amount it wins by seems to vary quite a bit.
539      */

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

544     /*
545      * sparc optimization:
546      *
547      * if 'block' is already aligned on a 4-byte boundary, use
548      * LOAD_BIG_32() directly. otherwise, bcopy() into a
549      * buffer that 'is' aligned on a 4-byte boundary and then do
550      * the LOAD_BIG_32() on that buffer. benchmarks have shown
551      * that using the bcopy() is better than loading the bytes
552      * individually and doing the endian-swap by hand.
553      *
554      * even though it's quite tempting to assign to do:
555      *
556      * blk = bcopy(ctx->buf_un.buf32, blk, sizeof (ctx->buf_un.buf32));
557      *
558      * and only have one set of LOAD_BIG_32()'s, the compiler
559      * *does not* like that, so please resist the urge.
560      */

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

```

```

565     w_14 = LOAD_BIG_32(ctx->buf_un.buf32 + 14);
566     w_13 = LOAD_BIG_32(ctx->buf_un.buf32 + 13);
567     w_12 = LOAD_BIG_32(ctx->buf_un.buf32 + 12);
568     w_11 = LOAD_BIG_32(ctx->buf_un.buf32 + 11);
569     w_10 = LOAD_BIG_32(ctx->buf_un.buf32 + 10);
570     w_9 = LOAD_BIG_32(ctx->buf_un.buf32 + 9);
571     w_8 = LOAD_BIG_32(ctx->buf_un.buf32 + 8);
572     w_7 = LOAD_BIG_32(ctx->buf_un.buf32 + 7);
573     w_6 = LOAD_BIG_32(ctx->buf_un.buf32 + 6);
574     w_5 = LOAD_BIG_32(ctx->buf_un.buf32 + 5);
575     w_4 = LOAD_BIG_32(ctx->buf_un.buf32 + 4);
576     w_3 = LOAD_BIG_32(ctx->buf_un.buf32 + 3);
577     w_2 = LOAD_BIG_32(ctx->buf_un.buf32 + 2);
578     w_1 = LOAD_BIG_32(ctx->buf_un.buf32 + 1);
579     w_0 = LOAD_BIG_32(ctx->buf_un.buf32 + 0);
580 } else {
581     /*LINTED*/
582     w_15 = LOAD_BIG_32(blk + 60);
583     /*LINTED*/
584     w_14 = LOAD_BIG_32(blk + 56);
585     /*LINTED*/
586     w_13 = LOAD_BIG_32(blk + 52);
587     /*LINTED*/
588     w_12 = LOAD_BIG_32(blk + 48);
589     /*LINTED*/
590     w_11 = LOAD_BIG_32(blk + 44);
591     /*LINTED*/
592     w_10 = LOAD_BIG_32(blk + 40);
593     /*LINTED*/
594     w_9 = LOAD_BIG_32(blk + 36);
595     /*LINTED*/
596     w_8 = LOAD_BIG_32(blk + 32);
597     /*LINTED*/
598     w_7 = LOAD_BIG_32(blk + 28);
599     /*LINTED*/
600     w_6 = LOAD_BIG_32(blk + 24);
601     /*LINTED*/
602     w_5 = LOAD_BIG_32(blk + 20);
603     /*LINTED*/
604     w_4 = LOAD_BIG_32(blk + 16);
605     /*LINTED*/
606     w_3 = LOAD_BIG_32(blk + 12);
607     /*LINTED*/
608     w_2 = LOAD_BIG_32(blk + 8);
609     /*LINTED*/
610     w_1 = LOAD_BIG_32(blk + 4);
611     /*LINTED*/
612     w_0 = LOAD_BIG_32(blk + 0);
613 }
614 #else /* !defined(__sparc) */
615
616 void /* CSTYLE */
617 SHA1Transform(SHA1_CTX *ctx, const uint8_t blk[64])
618 {
619     /* CSTYLE */
620     shalword a = ctx->state[0];
621     shalword b = ctx->state[1];
622     shalword c = ctx->state[2];
623     shalword d = ctx->state[3];
624     shalword e = ctx->state[4];
625
626 #if defined(W_ARRAY)
627     shalword w[16];
628 #else /* !defined(W_ARRAY) */
629     shalword w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7;
630     shalword w_8, w_9, w_10, w_11, w_12, w_13, w_14, w_15;

```

```

631 #endif /* !defined(W_ARRAY) */
632
633 W(0) = LOAD_BIG_32((void *) (blk + 0));
634 W(1) = LOAD_BIG_32((void *) (blk + 4));
635 W(2) = LOAD_BIG_32((void *) (blk + 8));
636 W(3) = LOAD_BIG_32((void *) (blk + 12));
637 W(4) = LOAD_BIG_32((void *) (blk + 16));
638 W(5) = LOAD_BIG_32((void *) (blk + 20));
639 W(6) = LOAD_BIG_32((void *) (blk + 24));
640 W(7) = LOAD_BIG_32((void *) (blk + 28));
641 W(8) = LOAD_BIG_32((void *) (blk + 32));
642 W(9) = LOAD_BIG_32((void *) (blk + 36));
643 W(10) = LOAD_BIG_32((void *) (blk + 40));
644 W(11) = LOAD_BIG_32((void *) (blk + 44));
645 W(12) = LOAD_BIG_32((void *) (blk + 48));
646 W(13) = LOAD_BIG_32((void *) (blk + 52));
647 W(14) = LOAD_BIG_32((void *) (blk + 56));
648 W(15) = LOAD_BIG_32((void *) (blk + 60));
649
650 #endif /* !defined(__sparc) */
651
652 /*
653  * general optimization:
654  *
655  * even though this approach is described in the standard as
656  * being slower algorithmically, it is 30-40% faster than the
657  * "faster" version under SPARC, because this version has more
658  * of the constraints specified at compile-time and uses fewer
659  * variables (and therefore has better register utilization)
660  * than its "speedier" brother. (i've tried both, trust me)
661  *
662  * for either method given in the spec, there is an "assignment"
663  * phase where the following takes place:
664  *
665  *     tmp = (main_computation);
666  *     e = d; d = c; c = rotate_left(b, 30); b = a; a = tmp;
667  *
668  * we can make the algorithm go faster by not doing this work,
669  * but just pretending that 'd' is now 'e', etc. this works
670  * really well and obviates the need for a temporary variable.
671  * however, we still explicitly perform the rotate action,
672  * since it is cheaper on SPARC to do it once than to have to
673  * do it over and over again.
674  */
675
676 /* round 1 */
677 e = ROTATE_LEFT(a, 5) + F(b, c, d) + e + W(0) + SHA1_CONST(0); /* 0 */
678 b = ROTATE_LEFT(b, 30);
679
680 d = ROTATE_LEFT(e, 5) + F(a, b, c) + d + W(1) + SHA1_CONST(0); /* 1 */

```

```

681     a = ROTATE_LEFT(a, 30);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

945     e = ROTATE_LEFT(a, 5) + G(b, c, d) + e + w(6) + SHA1_CONST(3);
946     b = ROTATE_LEFT(b, 30);

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

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

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

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

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

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

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

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

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

982     ctx->state[0] += ROTATE_LEFT(b, 5) + G(c, d, e) + a + w(15) +
983     SHA1_CONST(3);
984     ctx->state[1] += b;
985     ctx->state[2] += ROTATE_LEFT(c, 30);
986     ctx->state[3] += d;
987     ctx->state[4] += e;

989     /* zeroize sensitive information */
990     w(0) = w(1) = w(2) = w(3) = w(4) = w(5) = w(6) = w(7) = w(8) = 0;
991     w(9) = w(10) = w(11) = w(12) = w(13) = w(14) = w(15) = 0;
992 }

```

unchanged_portion_omitted