```
**********************************************************
   69927 Thu Apr 30 20:52:29 2015
new/usr/src/common/crypto/aes/aes_impl.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <netinet/in.h>
#include "aes_impl.h"
#ifndef _KERNEL
#include <strings.h>
#include <stdlib.h>
#include <sys/note.h>
#endif  /* !_KERNEL */

#ifdef __amd64

#ifdef _KERNEL
#include <sys/cpuvar.h>         /* cpu_t, CPU */
#include <sys/x86_archext.h>    /* x86_featureset, X86FSET_AES */
#include <sys/disp.h>           /* kpreempt_disable(), kpreempt_enable */

/* Workaround for no XMM kernel thread save/restore */
#define KPREEMPT_DISABLE        kpreempt_disable()
#define KPREEMPT_ENABLE         kpreempt_enable()

#else
#include <sys/auxv.h>           /* getisax() */
#include <sys/auxv_386.h>       /* AV_386_AES bit */
#define KPREEMPT_DISABLE
#define KPREEMPT_ENABLE
#endif  /* _KERNEL */
#endif  /* __amd64 */


/*
 * This file is derived from the file  rijndael-alg-fst.c  taken from the
 * "optimized C code v3.0" on the "rijndael home page"
 * http://www.iaik.tu-graz.ac.at/research/krypto/AES/old/~rijmen/rijndael/
 * pointed by the NIST web-site http://csrc.nist.gov/archive/aes/
```



```
 *
 * The following note is from the original file:
 */

/*
 * rijndael-alg-fst.c
 *
 * @version 3.0 (December 2000)
 *
 * Optimised ANSI C code for the Rijndael cipher (now AES)
 *
 * @author Vincent Rijmen <vincent.rijmen@esat.kuleuven.ac.be>
 * @author Antoon Bosselaers <antoon.bosselaers@esat.kuleuven.ac.be>
 * @author Paulo Barreto <paulo.barreto@terra.com.br>
 *
 * This code is hereby placed in the public domain.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHORS ''AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#if defined(sun4u)
/* External assembly functions: */
extern void aes_encrypt_impl(const uint32_t rk[], int Nr, const uint32_t pt[4],
        uint32_t ct[4]);
extern void aes_decrypt_impl(const uint32_t rk[], int Nr, const uint32_t ct[4],
        uint32_t pt[4]);

#define AES_ENCRYPT_IMPL(a, b, c, d)    aes_encrypt_impl(a, b, c, d)
#define AES_DECRYPT_IMPL(a, b, c, d)    aes_decrypt_impl(a, b, c, d)
#define AES_ENCRYPT_IMPL(a, b, c, d, e) aes_encrypt_impl(a, b, c, d)
#define AES_DECRYPT_IMPL(a, b, c, d, e) aes_decrypt_impl(a, b, c, d)

#elif defined(__amd64)

/* These functions are used to execute amd64 instructions for AMD or Intel: */
extern int rijndael_key_setup_enc_amd64(uint32_t rk[],
        const uint32_t cipherKey[], int keyBits);
extern int rijndael_key_setup_dec_amd64(uint32_t rk[],
        const uint32_t cipherKey[], int keyBits);
extern void aes_encrypt_amd64(const uint32_t rk[], int Nr,
        const uint32_t pt[4], uint32_t ct[4]);
extern void aes_decrypt_amd64(const uint32_t rk[], int Nr,
        const uint32_t ct[4], uint32_t pt[4]);

/* These functions are used to execute Intel-specific AES-NI instructions: */
extern int rijndael_key_setup_enc_intel(uint32_t rk[],
        const uint32_t cipherKey[], uint64_t keyBits);
extern int rijndael_key_setup_dec_intel(uint32_t rk[],
        const uint32_t cipherKey[], uint64_t keyBits);
extern void aes_encrypt_intel(const uint32_t rk[], int Nr,
        const uint32_t pt[4], uint32_t ct[4]);
extern void aes_decrypt_intel(const uint32_t rk[], int Nr,
        const uint32_t ct[4], uint32_t pt[4]);
extern void aes_encrypt_intel8(const uint32_t rk[], int Nr,
        const void *pt, void *ct);
extern void aes_decrypt_intel8(const uint32_t rk[], int Nr,
```

```
126         const void *ct, void *pt);
127 extern void aes_encrypt_cbc_intel8(const uint32_t rk[], int Nr,
128         const void *pt, void *ct, const void *iv);
129 extern void aes_ctr_intel8(const uint32_t rk[], int Nr,
130         const void *input, void *output, uint64_t counter_upper_BE,
131         uint64_t counter_lower_LE);
132 extern void aes_xor_intel(const uint8_t *, uint8_t *);

134 static inline int intel_aes_instructions_present(void);
122 static int intel_aes_instructions_present(void);

136 #ifdef  _KERNEL
137 /*
138  * Some form of floating-point acceleration is available, so declare these.
139  * The implementations will be in a platform-specific assembly file (e.g.
140  * amd64/aes_intel.s for SSE2/AES-NI).
141  */
142 extern void aes_accel_save(void *savestate);
143 extern void aes_accel_restore(void *savestate);
144 #endif  /* _KERNEL */
124 #define AES_ENCRYPT_IMPL(a, b, c, d, e) rijndael_encrypt(a, b, c, d, e)
125 #define AES_DECRYPT_IMPL(a, b, c, d, e) rijndael_decrypt(a, b, c, d, e)

146 #else /* Generic C implementation */
147 static void rijndael_encrypt(const uint32_t rk[], int Nr, const uint32_t pt[4],
148     uint32_t ct[4]);
149 static void rijndael_decrypt(const uint32_t rk[], int Nr, const uint32_t pt[4],
150     uint32_t ct[4]);

129 #define AES_ENCRYPT_IMPL(a, b, c, d, e) rijndael_encrypt(a, b, c, d)
130 #define AES_DECRYPT_IMPL(a, b, c, d, e) rijndael_decrypt(a, b, c, d)
151 #define rijndael_key_setup_enc_raw      rijndael_key_setup_enc
152 #define AES_ENCRYPT_IMPL(a, b, c, d)    rijndael_encrypt(a, b, c, d)
153 #define AES_DECRYPT_IMPL(a, b, c, d)    rijndael_decrypt(a, b, c, d)
154 #endif  /* sun4u || __amd64 */

156 #if defined(_LITTLE_ENDIAN) && !defined(__amd64)
157 #define AES_BYTE_SWAP
158 #endif


161 #if !defined(__amd64)
162 /*
163  *  Constant tables
164  */

166 /*
167  * Te0[x] = S [x].[02, 01, 01, 03];
168  * Te1[x] = S [x].[03, 02, 01, 01];
169  * Te2[x] = S [x].[01, 03, 02, 01];
170  * Te3[x] = S [x].[01, 01, 03, 02];
171  * Te4[x] = S [x].[01, 01, 01, 01];
172  *
173  * Td0[x] = Si[x].[0e, 09, 0d, 0b];
174  * Td1[x] = Si[x].[0b, 0e, 09, 0d];
175  * Td2[x] = Si[x].[0d, 0b, 0e, 09];
176  * Td3[x] = Si[x].[09, 0d, 0b, 0e];
177  * Td4[x] = Si[x].[01, 01, 01, 01];
178  */

180 /* Encrypt Sbox constants (for the substitute bytes operation) */

182 #ifndef sun4u

184 static const uint32_t Te0[256] =
185 {
```



```
186         0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
187         0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
188         0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
189         0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
190         0x8fcaca45U, 0x1f82829dU, 0x89c9c940U, 0xfa7d7d87U,
191         0xeffafa15U, 0xb25959ebU, 0x8e4747c9U, 0xfbf0f00bU,
192         0x41adadecU, 0xb3d4d467U, 0x5fa2a2fdU, 0x45afafeaU,
193         0x239c9cbfU, 0x53a4a4f7U, 0xe4727296U, 0x9bc0c05bU,
194         0x75b7b7c2U, 0xe1fdfd1cU, 0x3d9393aeU, 0x4c26266aU,
195         0x6c36365aU, 0x7e3f3f41U, 0xf5f7f702U, 0x83cccc4fU,
196         0x6834345cU, 0x51a5a5f4U, 0xd1e5e534U, 0xf9f1f108U,
197         0xe2717193U, 0xabd8d873U, 0x62313153U, 0x2a15153fU,
198         0x0804040cU, 0x95c7c752U, 0x46232365U, 0x9dc3c35eU,
199         0x30181828U, 0x379696a1U, 0x0a05050fU, 0x2f9a9ab5U,
200         0x0e070709U, 0x24121236U, 0x1b80809bU, 0xdfe2e23dU,
201         0xcdebeb26U, 0x4e272769U, 0x7fb2b2cdU, 0xea75759fU,
202         0x1209091bU, 0x1d83839eU, 0x582c2c74U, 0x341a1a2eU,
203         0x361b1b2dU, 0xdc6e6eb2U, 0xb45a5aeeU, 0x5ba0a0fbU,
204         0xa45252f6U, 0x763b3b4dU, 0xb7d6d661U, 0x7db3b3ceU,
205         0x5229297bU, 0xdde3e33eU, 0x5e2f2f71U, 0x13848497U,
206         0xa65353f5U, 0xb9d1d168U, 0x00000000U, 0xc1eded2cU,
207         0x40202060U, 0xe3fcfc1fU, 0x79b1b1c8U, 0xb65b5bedU,
208         0xd46a6abeU, 0x8dcbcb46U, 0x67bebed9U, 0x7239394bU,
209         0x944a4adeU, 0x984c4cd4U, 0xb05858e8U, 0x85cfcf4aU,
210         0xbbd0d06bU, 0xc5efef2aU, 0x4faaaae5U, 0xedfbfb16U,
211         0x864343c5U, 0x9a4d4dd7U, 0x66333355U, 0x11858594U,
212         0x8a4545cfU, 0xe9f9f910U, 0x04020206U, 0xfe7f7f81U,
213         0xa05050f0U, 0x783c3c44U, 0x259f9fbaU, 0x4ba8a8e3U,
214         0xa25151f3U, 0x5da3a3feU, 0x804040c0U, 0x058f8f8aU,
215         0x3f9292adU, 0x219d9dbcU, 0x70383848U, 0xf1f5f504U,
216         0x63bcbcdfU, 0x77b6b6c1U, 0xafdada75U, 0x42212163U,
217         0x20101030U, 0xe5ffff1aU, 0xfdf3f30eU, 0xbfd2d26dU,
218         0x81cdcd4cU, 0x180c0c14U, 0x26131335U, 0xc3ecec2fU,
219         0xbe5f5fe1U, 0x359797a2U, 0x884444ccU, 0x2e171739U,
220         0x93c4c457U, 0x55a7a7f2U, 0xfc7e7e82U, 0x7a3d3d47U,
221         0xc86464acU, 0xba5d5de7U, 0x3219192bU, 0xe6737395U,
222         0xc06060a0U, 0x19818198U, 0x9e4f4fd1U, 0xa3dcdc7fU,
223         0x44222266U, 0x542a2a7eU, 0x3b9090abU, 0x0b888883U,
224         0x8c4646caU, 0xc7eeee29U, 0x6bb8b8d3U, 0x2814143cU,
225         0xa7dede79U, 0xbc5e5ee2U, 0x160b0b1dU, 0xaddbdb76U,
226         0xdbe0e03bU, 0x64323256U, 0x743a3a4eU, 0x140a0a1eU,
227         0x924949dbU, 0x0c06060aU, 0x4824246cU, 0xb85c5ce4U,
228         0x9fc2c25dU, 0xbdd3d36eU, 0x43acacefU, 0xc46262a6U,
229         0x399191a8U, 0x319595a4U, 0xd3e4e437U, 0xf279798bU,
230         0xd5e7e732U, 0x8bc8c843U, 0x6e373759U, 0xda6d6db7U,
231         0x018d8d8cU, 0xb1d5d564U, 0x9c4e4ed2U, 0x49a9a9e0U,
232         0xd86c6cb4U, 0xac5656faU, 0xf3f4f407U, 0xcfeaea25U,
233         0xca6565afU, 0xf47a7a8eU, 0x47aeaee9U, 0x10080818U,
234         0x6fbabad5U, 0xf0787888U, 0x4a25256fU, 0x5c2e2e72U,
235         0x381c1c24U, 0x57a6a6f1U, 0x73b4b4c7U, 0x97c6c651U,
236         0xcbe8e823U, 0xa1dddd7cU, 0xe874749cU, 0x3e1f1f21U,
237         0x964b4bddU, 0x61bdbddcU, 0x0d8b8b86U, 0x0f8a8a85U,
238         0xe0707090U, 0x7c3e3e42U, 0x71b5b5c4U, 0xcc6666aaU,
239         0x904848d8U, 0x06030305U, 0xf7f6f601U, 0x1c0e0e12U,
240         0xc26161a3U, 0x6a35355fU, 0xae5757f9U, 0x69b9b9d0U,
241         0x17868691U, 0x99c1c158U, 0x3a1d1d27U, 0x279e9eb9U,
242         0xd9e1e138U, 0xebf8f813U, 0x2b9898b3U, 0x22111133U,
243         0xd26969bbU, 0xa9d9d970U, 0x078e8e89U, 0x339494a7U,
244         0x2d9b9bb6U, 0x3c1e1e22U, 0x15878792U, 0xc9e9e920U,
245         0x87cece49U, 0xaa5555ffU, 0x50282878U, 0xa5dfdf7aU,
246         0x038c8c8fU, 0x59a1a1f8U, 0x09898980U, 0x1a0d0d17U,
247         0x65bfbfdaU, 0xd7e6e631U, 0x844242c6U, 0xd06868b8U,
248         0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,
249         0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU
250 };
```

_____***unchanged_portion_omitted_***

```
#elif defined(__amd64)

/*
 * Expand the 32-bit AES cipher key array into the encryption and decryption
 * key schedules.
 *
 * Parameters:
 * key          AES key schedule to be initialized
 * keyarr32     User key
 * keyBits      AES key size (128, 192, or 256 bits)
 */
static void
aes_setupkeys(aes_key_t *key, const uint32_t *keyarr32, int keybits)
{
        AES_ACCEL_SAVESTATE(savestate);
        aes_accel_enter(savestate);

        if (intel_aes_instructions_present()) {
                key->flags = INTEL_AES_NI_CAPABLE;
                KPREEMPT_DISABLE;
                key->nr = rijndael_key_setup_enc_intel(&(key->encr_ks.ks32[0]),
                    keyarr32, keybits);
                key->nr = rijndael_key_setup_dec_intel(&(key->decr_ks.ks32[0]),
                    keyarr32, keybits);
                KPREEMPT_ENABLE;
        } else {
                key->flags = 0;
                key->nr = rijndael_key_setup_enc_amd64(&(key->encr_ks.ks32[0]),
                    keyarr32, keybits);
                key->nr = rijndael_key_setup_dec_amd64(&(key->decr_ks.ks32[0]),
                    keyarr32, keybits);
        }

        aes_accel_exit(savestate);
        key->type = AES_32BIT_KS;
}

/*
 * Encrypt one block of data. The block is assumed to be an array
 * of four uint32_t values, so copy for alignment (and byte-order
 * reversal for little endian systems might be necessary on the
 * input and output byte streams.
 * The size of the key schedule depends on the number of rounds
 * (which can be computed from the size of the key), i.e. 4*(Nr + 1).
 *
 * Parameters:
 * rk           Key schedule, of aes_ks_t (60 32-bit integers)
 * Nr           Number of rounds
 * pt           Input block (plain text)
 * ct           Output block (crypto text).  Can overlap with pt
 * flags        Indicates whether we're on Intel AES-NI-capable hardware
 */
static void
rijndael_encrypt(const uint32_t rk[], int Nr, const uint32_t pt[4],
    uint32_t ct[4], int flags) {
        if (flags & INTEL_AES_NI_CAPABLE) {
                KPREEMPT_DISABLE;
                aes_encrypt_intel(rk, Nr, pt, ct);
                KPREEMPT_ENABLE;
        } else {
                aes_encrypt_amd64(rk, Nr, pt, ct);
        }
}
```



```
/*
 * Decrypt one block of data. The block is assumed to be an array
 * of four uint32_t values, so copy for alignment (and byte-order
 * reversal for little endian systems might be necessary on the
 * input and output byte streams.
 * The size of the key schedule depends on the number of rounds
 * (which can be computed from the size of the key), i.e. 4*(Nr + 1).
 *
 * Parameters:
 * rk           Key schedule, of aes_ks_t (60 32-bit integers)
 * Nr           Number of rounds
 * ct           Input block (crypto text)
 * pt           Output block (plain text). Can overlap with pt
 * flags        Indicates whether we're on Intel AES-NI-capable hardware
 */
static void
rijndael_decrypt(const uint32_t rk[], int Nr, const uint32_t ct[4],
    uint32_t pt[4], int flags) {
        if (flags & INTEL_AES_NI_CAPABLE) {
                KPREEMPT_DISABLE;
                aes_decrypt_intel(rk, Nr, ct, pt);
                KPREEMPT_ENABLE;
        } else {
                aes_decrypt_amd64(rk, Nr, ct, pt);
        }
}

#else /* generic C implementation */

/*
 *  Expand the cipher key into the decryption key schedule.
 *  Return the number of rounds for the given cipher key size.
 *  The size of the key schedule depends on the number of rounds
 *  (which can be computed from the size of the key), i.e. 4*(Nr + 1).
 *
 * Parameters:
 * rk           AES key schedule 32-bit array to be initialized
 * cipherKey    User key
 * keyBits      AES key size (128, 192, or 256 bits)
 */
static int
rijndael_key_setup_dec(uint32_t rk[], const uint32_t cipherKey[], int keyBits)
{
        int      Nr, i, j;
        uint32_t temp;

        /* expand the cipher key: */
        Nr = rijndael_key_setup_enc_raw(rk, cipherKey, keyBits);

        /* invert the order of the round keys: */
        for (i = 0, j = 4 * Nr; i < j; i += 4, j -= 4) {
                temp = rk[i];
                rk[i] = rk[j];
                rk[j] = temp;
                temp = rk[i + 1];
                rk[i + 1] = rk[j + 1];
                rk[j + 1] = temp;
                temp = rk[i + 2];
                rk[i + 2] = rk[j + 2];
                rk[j + 2] = temp;
                temp = rk[i + 3];
                rk[i + 3] = rk[j + 3];
                rk[j + 3] = temp;
        }
```

```
1222         /*
1223          * apply the inverse MixColumn transform to all
1224          * round keys but the first and the last:
1225          */
1226         for (i = 1; i < Nr; i++) {
1227                 rk += 4;
1228                 rk[0] = Td0[Te4[rk[0] >> 24] & 0xff] ^
1229                     Td1[Te4[(rk[0] >> 16) & 0xff] & 0xff] ^
1230                     Td2[Te4[(rk[0] >> 8) & 0xff] & 0xff] ^
1231                     Td3[Te4[rk[0] & 0xff] & 0xff];
1232                 rk[1] = Td0[Te4[rk[1] >> 24] & 0xff] ^
1233                     Td1[Te4[(rk[1] >> 16) & 0xff] & 0xff] ^
1234                     Td2[Te4[(rk[1] >> 8) & 0xff] & 0xff] ^
1235                     Td3[Te4[rk[1] & 0xff] & 0xff];
1236                 rk[2] = Td0[Te4[rk[2] >> 24] & 0xff] ^
1237                     Td1[Te4[(rk[2] >> 16) & 0xff] & 0xff] ^
1238                     Td2[Te4[(rk[2] >> 8) & 0xff] & 0xff] ^
1239                     Td3[Te4[rk[2] & 0xff] & 0xff];
1240                 rk[3] = Td0[Te4[rk[3] >> 24] & 0xff] ^
1241                     Td1[Te4[(rk[3] >> 16) & 0xff] & 0xff] ^
1242                     Td2[Te4[(rk[3] >> 8) & 0xff] & 0xff] ^
1243                     Td3[Te4[rk[3] & 0xff] & 0xff];
1244         }

1246         return (Nr);
1247 }
```

unchanged_portion_omitted

```
1592 #if     defined(__amd64) && defined(_KERNEL)
1593 void
1594 aes_accel_enter(void *savestate)
1595 {
1596         KPREEMPT_DISABLE;
1597         aes_accel_save(savestate);
1598 }

1600 void
1601 aes_accel_exit(void *savestate)
1602 {
1603         aes_accel_restore(savestate);
1604         KPREEMPT_ENABLE;
1605 }
1606 #endif  /* defined(__amd64) && defined(_KERNEL) */

1608 /*
1609  * Encrypt one block using AES.
1610  * Align if needed and (for x86 32-bit only) byte-swap.
1611  *
1612  * Parameters:
1613  * ks   Key schedule, of type aes_key_t
1614  * pt   Input block (plain text)
1615  * ct   Output block (crypto text).  Can overlap with pt
1616  */
1617 int
1618 aes_encrypt_block(const void *ks, const uint8_t *pt, uint8_t *ct)
1619 {
1620         aes_key_t       *ksch = (aes_key_t *)ks;

1622 #ifdef  __amd64
1623         if (intel_aes_instructions_present())
1624                 aes_encrypt_intel(&ksch->encr_ks.ks32[0], ksch->nr,
1625                     /* LINTED:  pointer alignment */
1626                     (uint32_t *)pt, (uint32_t *)ct);
1627         else
1628                 aes_encrypt_amd64(&ksch->encr_ks.ks32[0], ksch->nr,
1629                     /* LINTED:  pointer alignment */
```



```
1630                     (uint32_t *)pt, (uint32_t *)ct);
1631 #else   /* !__amd64 */
1632 #ifndef AES_BYTE_SWAP
1633         if (IS_P2ALIGNED2(pt, ct, sizeof (uint32_t))) {
1642                 /* LINTED:  pointer alignment */
1634                 AES_ENCRYPT_IMPL(&ksch->encr_ks.ks32[0], ksch->nr,
1635                     /* LINTED:  pointer alignment */
1636                     (uint32_t *)pt, (uint32_t *)ct);
1645                 (uint32_t *)pt, (uint32_t *)ct, ksch->flags);
1637         } else {
1638 #endif
1639                 uint32_t buffer[AES_BLOCK_LEN / sizeof (uint32_t)];

1641                 /* Copy input block into buffer */
1642 #ifndef AES_BYTE_SWAP
1643                 bcopy(pt, &buffer, AES_BLOCK_LEN);

1645 #else   /* byte swap */
1646                 buffer[0] = htonl(*(uint32_t *)(void *)&pt[0]);
1647                 buffer[1] = htonl(*(uint32_t *)(void *)&pt[4]);
1648                 buffer[2] = htonl(*(uint32_t *)(void *)&pt[8]);
1649                 buffer[3] = htonl(*(uint32_t *)(void *)&pt[12]);
1650 #endif  /* byte swap */
1659 #endif

1652                 AES_ENCRYPT_IMPL(&ksch->encr_ks.ks32[0], ksch->nr,
1653                     buffer, buffer);
1662                 buffer, buffer, ksch->flags);

1655                 /* Copy result from buffer to output block */
1656 #ifndef AES_BYTE_SWAP
1657                 bcopy(&buffer, ct, AES_BLOCK_LEN);
1658         }

1660 #else   /* byte swap */
1661                 *(uint32_t *)(void *)&ct[0] = htonl(buffer[0]);
1662                 *(uint32_t *)(void *)&ct[4] = htonl(buffer[1]);
1663                 *(uint32_t *)(void *)&ct[8] = htonl(buffer[2]);
1664                 *(uint32_t *)(void *)&ct[12] = htonl(buffer[3]);
1665 #endif  /* byte swap */
1666 #endif  /* !__amd64 */

1674 #endif
1668         return (CRYPTO_SUCCESS);
1669 }


1672 /*
1673  * Decrypt one block using AES.
1674  * Align and byte-swap if needed.
1675  *
1676  * Parameters:
1677  * ks   Key schedule, of type aes_key_t
1678  * ct   Input block (crypto text)
1679  * pt   Output block (plain text). Can overlap with pt
1680  */
1681 int
1682 aes_decrypt_block(const void *ks, const uint8_t *ct, uint8_t *pt)
1683 {
1684         aes_key_t       *ksch = (aes_key_t *)ks;

1686 #ifdef  __amd64
1687         if (intel_aes_instructions_present())
1688                 aes_decrypt_intel(&ksch->decr_ks.ks32[0], ksch->nr,
1689                     /* LINTED:  pointer alignment */
1690                     (uint32_t *)ct, (uint32_t *)pt);
```

```
1691         else
1692                 aes_decrypt_amd64(&ksch->decr_ks.ks32[0], ksch->nr,
1693                     /* LINTED:  pointer alignment */
1694                     (uint32_t *)ct, (uint32_t *)pt);
1695 #else   /* !__amd64 */
1696 #ifndef AES_BYTE_SWAP
1697         if (IS_P2ALIGNED2(ct, pt, sizeof (uint32_t))) {
1695                 /* LINTED:  pointer alignment */
1698                 AES_DECRYPT_IMPL(&ksch->decr_ks.ks32[0], ksch->nr,
1699                     /* LINTED:  pointer alignment */
1700                     (uint32_t *)ct, (uint32_t *)pt);
1698                     (uint32_t *)ct, (uint32_t *)pt, ksch->flags);
1701         } else {
1702 #endif
1703                 uint32_t buffer[AES_BLOCK_LEN / sizeof (uint32_t)];

1705                 /* Copy input block into buffer */
1706 #ifndef AES_BYTE_SWAP
1707                 bcopy(ct, &buffer, AES_BLOCK_LEN);

1709 #else   /* byte swap */
1710                 buffer[0] = htonl(*(uint32_t *)(void *)&ct[0]);
1711                 buffer[1] = htonl(*(uint32_t *)(void *)&ct[4]);
1712                 buffer[2] = htonl(*(uint32_t *)(void *)&ct[8]);
1713                 buffer[3] = htonl(*(uint32_t *)(void *)&ct[12]);
1714 #endif  /* byte swap */
1712 #endif

1716                 AES_DECRYPT_IMPL(&ksch->decr_ks.ks32[0], ksch->nr,
1717                     buffer, buffer);
1715                     buffer, buffer, ksch->flags);

1719                 /* Copy result from buffer to output block */
1720 #ifndef AES_BYTE_SWAP
1721                 bcopy(&buffer, pt, AES_BLOCK_LEN);
1722         }

1724 #else   /* byte swap */
1725         *(uint32_t *)(void *)&pt[0] = htonl(buffer[0]);
1726         *(uint32_t *)(void *)&pt[4] = htonl(buffer[1]);
1727         *(uint32_t *)(void *)&pt[8] = htonl(buffer[2]);
1728         *(uint32_t *)(void *)&pt[12] = htonl(buffer[3]);
1729 #endif  /* byte swap */
1730 #endif  /* !__amd64 */
1727 #endif

1732         return (CRYPTO_SUCCESS);
1733 }

1735 #define ECB_LOOP(ciph_func)                                             \
1736         do {                                                            \
1737                 for (; i < length; i += AES_BLOCK_LEN)                  \
1738                         ciph_func;                                      \
1739                 _NOTE(CONSTCOND)                                        \
1740         } while (0)
1741 #define ECB_LOOP_4P(ciph_func, enc_or_dec, in, out)                     \
1742         ECB_LOOP(ciph_func(&ksch->enc_or_dec ## r_ks.ks32[0],           \
1743             ksch->nr, (void *)&in[i], (void *)&out[i]))
1744 #define ECB_LOOP_3P(ciph_func, in, out)                                 \
1745         ECB_LOOP(ciph_func(ksch, (void *)&in[i], (void *)&out[i]))

1747 #ifdef  __amd64
1748 #define ECB_INTEL_IMPL(enc_or_dec, in, out)                             \
1749         do {                                                            \
1750                 if (intel_aes_instructions_present()) {                 \
1751                         /* first use the accelerated function */        \
```



```
1752                         for (; i + 8 * AES_BLOCK_LEN <= length;         \
1753                             i += 8 * AES_BLOCK_LEN)                     \
1754                                 aes_ ## enc_or_dec ## rypt_intel8(      \
1755                                     &ksch->enc_or_dec ## r_ks.ks32[0],  \
1756                                     ksch->nr, &in[i], &out[i]);         \
1757                         /* finish off the remainder per-block */        \
1758                         ECB_LOOP_4P(aes_ ## enc_or_dec ## rypt_intel,   \
1759                             enc_or_dec, in, out);                       \
1760                 } else {                                                \
1761                         ECB_LOOP_4P(aes_ ## enc_or_dec ## rypt_amd64,   \
1762                             enc_or_dec, in, out);                       \
1763                 }                                                       \
1764                 _NOTE(CONSTCOND)                                        \
1765         } while (0)
1766 #endif  /* __amd64 */

1768 /*
1769  * Perform AES ECB encryption on a sequence of blocks. On x86-64 CPUs with
1770  * the AES-NI extension, this performs the encryption in increments of 8
1771  * blocks at a time, exploiting instruction parallelism more efficiently.
1772  * On other platforms, this simply encrypts the blocks in sequence.
1773  */
1774 int
1775 aes_encrypt_ecb(const void *ks, const uint8_t *pt, uint8_t *ct, uint64_t length)
1776 {
1777         aes_key_t *ksch = (aes_key_t *)ks;
1778         uint64_t i = 0;

1780 #ifdef  __amd64
1781         ECB_INTEL_IMPL(enc, pt, ct);
1782 #elif   defined(sun4u)
1783         ECB_LOOP_4P(aes_encrypt_impl, enc, pt, ct);
1784 #else   /* Generic C implementation */
1785         ECB_LOOP_3P((void) aes_encrypt_block, pt, ct);
1786 #endif  /* Generic C implementation */

1788         return (CRYPTO_SUCCESS);
1789 }

1791 /*
1792  * Same as aes_encrypt_ecb, but performs decryption.
1793  */
1794 int
1795 aes_decrypt_ecb(const void *ks, const uint8_t *ct, uint8_t *pt, uint64_t length)
1796 {
1797         aes_key_t *ksch = (aes_key_t *)ks;
1798         uint64_t i = 0;

1800 #ifdef  __amd64
1801         ECB_INTEL_IMPL(dec, ct, pt);
1802 #elif   defined(sun4u)
1803         ECB_LOOP_4P(aes_decrypt_impl, dec, ct, pt);
1804 #else   /* Generic C implementation */
1805         ECB_LOOP_3P((void) aes_decrypt_block, ct, pt);
1806 #endif  /* Generic C implementation */

1808         return (CRYPTO_SUCCESS);
1809 }
1810 #ifdef  __amd64
1811 #undef  ECB_INTEL_IMPL
1812 #endif  /* __amd64 */

1814 #undef  ECB_LOOP
1815 #undef  ECB_LOOP_4P
1816 #undef  ECB_LOOP_3P
```

```
#define CBC_LOOP(enc_func, xor_func)                                    \
        do {                                                            \
                for (; i < length; i += AES_BLOCK_LEN) {                \
                        /* copy IV to ciphertext */                     \
                        bcopy(iv, &ct[i], AES_BLOCK_LEN);               \
                        /* XOR IV with plaintext with input */          \
                        xor_func(&pt[i], &ct[i]);                       \
                        /* encrypt counter in output region */          \
                        enc_func;                                       \
                        iv = &ct[i];                                    \
                }                                                       \
                _NOTE(CONSTCOND)                                        \
        } while (0)
#define CBC_LOOP_4P(enc_func, xor_func)                                 \
        CBC_LOOP(enc_func(&ksch->encr_ks.ks32[0],                       \
            ksch->nr, (void *)&ct[i], (void *)&ct[i]), xor_func)
#define CBC_LOOP_3P(enc_func, xor_func)                                 \
        CBC_LOOP(enc_func(ksch, (void *)&ct[i], (void *)&ct[i]), xor_func)

/*
 * Encrypts a sequence of consecutive AES blocks in CBC mode. On x86-64
 * with the AES-NI extension, the encryption is performed on 8 blocks at
 * a time using an optimized assembly implementation, giving a speed boost
 * of around 75%. On other platforms, this simply performs CBC encryption
 * in sequence on the blocks.
 *
 * Decryption acceleration is implemented in the kernel kcf block cipher
 * modes code (cbc.c), because that doesn't require a complete hand-tuned
 * CBC implementation in assembly.
 */
int
aes_encrypt_cbc(const void *ks, const uint8_t *pt, uint8_t *ct,
    const uint8_t *iv, uint64_t length)
{
        aes_key_t *ksch = (aes_key_t *)ks;
        size_t i = 0;

#ifdef  __amd64
        if (intel_aes_instructions_present()) {
                for (; i + 8 * AES_BLOCK_LEN <= length;
                    i += 8 * AES_BLOCK_LEN) {
                        aes_encrypt_cbc_intel8(&ksch->encr_ks.ks32[0],
                            ksch->nr, &ct[i], &ct[i], iv);
                        iv = &ct[7 * AES_BLOCK_LEN];
                }
                CBC_LOOP_4P(aes_encrypt_intel, aes_xor_intel);
        } else {
                CBC_LOOP_4P(aes_encrypt_amd64, aes_xor_intel);
        }
#elif   defined(sun4u)
        CBC_LOOP_4P(aes_encrypt_impl, aes_xor_block);
#else   /* Generic C implementation */
        CBC_LOOP_3P((void) aes_encrypt_block, aes_xor_block);
#endif  /* Generic C implementation */

        return (CRYPTO_SUCCESS);
}
#undef  CBC_LOOP
#undef  CBC_LOOP_4P
#undef  CBC_LOOP_3P

#define CTR_LOOP(enc_func, xor_func)                                    \
        do {                                                            \
                for (; i < length; i += AES_BLOCK_LEN) {                \
                        /* set up counter in output region */           \
                        *(uint64_t *)(void *)&output[i] = counter[0];   \
```



```
                        *(uint64_t *)(void *)&output[i + 8] =           \
                            htonll(counter[1]++);                       \
                        /* encrypt counter in output region */          \
                        enc_func;                                       \
                        /* XOR encrypted counter with input */          \
                        xor_func(&input[i], &output[i]);                \
                }                                                       \
                _NOTE(CONSTCOND)                                        \
        } while (0)
#define CTR_LOOP_4P(enc_func, xor_func)                                 \
        CTR_LOOP(enc_func(&ksch->encr_ks.ks32[0], ksch->nr,             \
            (void *)&output[i], (void *)&output[i]), xor_func)
#define CTR_LOOP_3P(enc_func, xor_func)                                 \
        CTR_LOOP(enc_func(ksch, (void *)&output[i], (void *)&output[i]),\
            xor_func)
/*
 * Performs high-performance counter mode encryption and decryption on
 * a sequence of blocks. In CTR mode, encryption and decryption are the
 * same operation, just with the plaintext and ciphertext reversed:
 * plaintext = CTR(CTR(plaintext, K), K)
 * Blocks also do not interdepend on each other, so it is an excellent
 * mode when high performance is required and data authentication/integrity
 * checking is provided via some other means, or isn't necessary.
 *
 * On x86-64 with the AES-NI extension, this code performs CTR mode
 * encryption in parallel on 8 blocks at a time and can provide in
 * excess of 3GB/s/core of encryption/decryption performance (<1 CPB).
 */
int
aes_ctr_mode(const void *ks, const uint8_t *input, uint8_t *output,
    uint64_t length, uint64_t counter[2])
{
        aes_key_t *ksch = (aes_key_t *)ks;
        uint64_t i = 0;

        // swap lower part to host order for computations
        counter[1] = ntohll(counter[1]);

#ifdef  __amd64
        if (intel_aes_instructions_present()) {
                /* first use the wide-register accelerated function */
                for (; i + 8 * AES_BLOCK_LEN <= length;
                    i += 8 * AES_BLOCK_LEN) {
                        aes_ctr_intel8(&ksch->encr_ks.ks32[0], ksch->nr,
                            &input[i], &output[i], counter[0], counter[1]);
                        counter[1] += 8;
                }
                /* finish off the remainder using the slow per-block method */
                CTR_LOOP_4P(aes_encrypt_intel, aes_xor_intel);
        } else {
                CTR_LOOP_4P(aes_encrypt_amd64, aes_xor_intel);
        }
#elif   defined(sun4u)
        CTR_LOOP_4P(aes_encrypt_impl, aes_xor_block);
#else   /* Generic C implementation */
        CTR_LOOP_3P((void) aes_encrypt_block, aes_xor_block);
#endif  /* Generic C implementation */

        // swap lower part back to big endian
        counter[1] = htonll(counter[1]);

        return (CRYPTO_SUCCESS);
}
#undef  CTR_LOOP

/*
```

```
 * Allocate key schedule for AES.
 *
 * Return the pointer and set size to the number of bytes allocated.
 * Memory allocated must be freed by the caller when done.
 *
 * Parameters:
 * size         Size of key schedule allocated, in bytes
 * kmflag       Flag passed to kmem_alloc(9F); ignored in userland.
 */
/* ARGSUSED */
void *
aes_alloc_keysched(size_t *size, int kmflag)
{
        aes_key_t *keysched;

#ifdef  _KERNEL
        keysched = (aes_key_t *)kmem_alloc(sizeof (aes_key_t), kmflag);
#else   /* !_KERNEL */
        keysched = (aes_key_t *)malloc(sizeof (aes_key_t));
#endif  /* _KERNEL */

        if (keysched != NULL) {
                *size = sizeof (aes_key_t);
                return (keysched);
        }
        return (NULL);
}


#ifdef __amd64
/*
 * Return 1 if executing on x86-64 with AES-NI instructions, otherwise 0.
 * Return 1 if executing on Intel with AES-NI instructions,
 * otherwise 0 (i.e., Intel without AES-NI or AMD64).
 * Cache the result, as the CPU can't change.
 *
 * Note: the userland version uses getisax().  The kernel version uses
 * global variable x86_featureset.
 */
static inline int
static int
intel_aes_instructions_present(void)
{
        static int      cached_result = -1;

        if (cached_result == -1) { /* first time */
#ifdef _KERNEL
                cached_result = is_x86_feature(x86_featureset, X86FSET_AES);
#else
                uint_t          ui = 0;

                (void) getisax(&ui, 1);
                cached_result = (ui & AV_386_AES) != 0;
#endif  /* _KERNEL */
        }

        return (cached_result);
}
```

unchanged_portion_omitted_

```
**********************************************************
    8103 Thu Apr 30 20:52:29 2015
new/usr/src/common/crypto/aes/aes_impl.h
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#ifndef _AES_IMPL_H
#define _AES_IMPL_H

/*
 * Common definitions used by AES.
 */

#ifdef  __cplusplus
extern "C" {
#endif

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

/* Similar to sysmacros.h IS_P2ALIGNED, but checks two pointers: */
#define IS_P2ALIGNED2(v, w, a) \
        ((((uintptr_t)(v) | (uintptr_t)(w)) & ((uintptr_t)(a) - 1)) == 0)

#define AES_BLOCK_LEN   16      /* bytes */
/* Round constant length, in number of 32-bit elements: */
#define RC_LENGTH       (5 * ((AES_BLOCK_LEN) / 4 - 2))

#define AES_COPY_BLOCK_UNALIGNED(src, dst) \
#define AES_COPY_BLOCK(src, dst) \
        (dst)[0] = (src)[0]; \
        (dst)[1] = (src)[1]; \
        (dst)[2] = (src)[2]; \
        (dst)[3] = (src)[3]; \
        (dst)[4] = (src)[4]; \
        (dst)[5] = (src)[5]; \
        (dst)[6] = (src)[6]; \
        (dst)[7] = (src)[7]; \
        (dst)[8] = (src)[8]; \
```



```
        (dst)[9] = (src)[9]; \
        (dst)[10] = (src)[10]; \
        (dst)[11] = (src)[11]; \
        (dst)[12] = (src)[12]; \
        (dst)[13] = (src)[13]; \
        (dst)[14] = (src)[14]; \
        (dst)[15] = (src)[15]

#define AES_XOR_BLOCK_UNALIGNED(src, dst) \
#define AES_XOR_BLOCK(src, dst) \
        (dst)[0] ^= (src)[0]; \
        (dst)[1] ^= (src)[1]; \
        (dst)[2] ^= (src)[2]; \
        (dst)[3] ^= (src)[3]; \
        (dst)[4] ^= (src)[4]; \
        (dst)[5] ^= (src)[5]; \
        (dst)[6] ^= (src)[6]; \
        (dst)[7] ^= (src)[7]; \
        (dst)[8] ^= (src)[8]; \
        (dst)[9] ^= (src)[9]; \
        (dst)[10] ^= (src)[10]; \
        (dst)[11] ^= (src)[11]; \
        (dst)[12] ^= (src)[12]; \
        (dst)[13] ^= (src)[13]; \
        (dst)[14] ^= (src)[14]; \
        (dst)[15] ^= (src)[15]

#define AES_COPY_BLOCK_ALIGNED(src, dst) \
        ((uint64_t *)(void *)(dst))[0] = ((uint64_t *)(void *)(src))[0]; \
        ((uint64_t *)(void *)(dst))[1] = ((uint64_t *)(void *)(src))[1]

#define AES_XOR_BLOCK_ALIGNED(src, dst) \
        ((uint64_t *)(void *)(dst))[0] ^= ((uint64_t *)(void *)(src))[0]; \
        ((uint64_t *)(void *)(dst))[1] ^= ((uint64_t *)(void *)(src))[1]

/* AES key size definitions */
#define AES_MINBITS             128
#define AES_MINBYTES            ((AES_MINBITS) >> 3)
#define AES_MAXBITS             256
#define AES_MAXBYTES            ((AES_MAXBITS) >> 3)

#define AES_MIN_KEY_BYTES       ((AES_MINBITS) >> 3)
#define AES_MAX_KEY_BYTES       ((AES_MAXBITS) >> 3)
#define AES_192_KEY_BYTES       24
#define AES_IV_LEN              16

/* AES key schedule may be implemented with 32- or 64-bit elements: */
#define AES_32BIT_KS            32
#define AES_64BIT_KS            64

#define MAX_AES_NR              14 /* Maximum number of rounds */
#define MAX_AES_NB              4  /* Number of columns comprising a state */

/*
 * Architecture-specific acceleration support autodetection.
 * Some architectures provide hardware-assisted acceleration using floating
 * point registers, which need special handling inside of the kernel, so the
 * macros below define the auxiliary functions needed to utilize them.
 */
#if     defined(__amd64) && defined(_KERNEL)
/*
 * Using floating point registers requires temporarily disabling kernel
 * thread preemption, so we need to operate on small-enough chunks to
 * prevent scheduling latency bubbles.
 * A typical 64-bit CPU can sustain around 300-400MB/s/core even in the
 * slowest encryption modes (CBC), which with 32k per run works out to ~100us
```

```
 * per run. CPUs with AES-NI in fast modes (ECB, CTR, CBC decryption) can
 * easily sustain 3GB/s/core, so the latency potential essentially vanishes.
 */
#define AES_OPSZ        32768

#if     defined(lint) || defined(__lint)
#define AES_ACCEL_SAVESTATE(name)       uint8_t name[16 * 16 + 8]
#else   /* lint || __lint */
#define AES_ACCEL_SAVESTATE(name) \
        /* stack space for xmm0--xmm15 and cr0 (16 x 128 bits + 64 bits) */ \
        uint8_t name[16 * 16 + 8] __attribute__((aligned(16)))
#endif  /* lint || __lint */

#else   /* !defined(__amd64) || !defined(_KERNEL) */
/*
 * All other accel support
 */
#define AES_OPSZ        ((size_t)-1)
/* On other architectures or outside of the kernel these get stubbed out */
#define AES_ACCEL_SAVESTATE(name)
#define aes_accel_enter(savestate)
#define aes_accel_exit(savestate)
#endif  /* !defined(__amd64) || !defined(_KERNEL) */

typedef union {
#ifdef  sun4u
        uint64_t        ks64[((MAX_AES_NR) + 1) * (MAX_AES_NB)];
#endif
        uint32_t        ks32[((MAX_AES_NR) + 1) * (MAX_AES_NB)];
} aes_ks_t;

/* aes_key.flags value: */
#define INTEL_AES_NI_CAPABLE    0x1     /* AES-NI instructions present */

typedef struct aes_key aes_key_t;
struct aes_key {
        aes_ks_t        encr_ks;  /* encryption key schedule */
        aes_ks_t        decr_ks;  /* decryption key schedule */
#ifdef __amd64
        long double     align128; /* Align fields above for Intel AES-NI */
        int             flags;    /* implementation-dependent flags */
#endif  /* __amd64 */
        int             nr;       /* number of rounds (10, 12, or 14) */
        int             type;     /* key schedule size (32 or 64 bits) */
};

/*
 * Core AES functions.
 * ks and keysched are pointers to aes_key_t.
 * They are declared void* as they are intended to be opaque types.
 * Use function aes_alloc_keysched() to allocate memory for ks and keysched.
 */
extern void *aes_alloc_keysched(size_t *size, int kmflag);
extern void aes_init_keysched(const uint8_t *cipherKey, uint_t keyBits,
        void *keysched);
extern int aes_encrypt_block(const void *ks, const uint8_t *pt, uint8_t *ct);
extern int aes_decrypt_block(const void *ks, const uint8_t *ct, uint8_t *pt);
extern int aes_encrypt_ecb(const void *ks, const uint8_t *pt, uint8_t *ct,
    uint64_t length);
extern int aes_decrypt_ecb(const void *ks, const uint8_t *pt, uint8_t *ct,
    uint64_t length);
extern int aes_encrypt_cbc(const void *ks, const uint8_t *pt, uint8_t *ct,
    const uint8_t *iv, uint64_t length);
extern int aes_ctr_mode(const void *ks, const uint8_t *pt, uint8_t *ct,
    uint64_t length, uint64_t counter[2]);
```



```
/*
 * AES mode functions.
 * The first 2 functions operate on 16-byte AES blocks.
 */
#ifdef  __amd64
#define AES_COPY_BLOCK  aes_copy_intel
#define AES_XOR_BLOCK   aes_xor_intel
extern void aes_copy_intel(const uint8_t *src, uint8_t *dst);
extern void aes_xor_intel(const uint8_t *src, uint8_t *dst);
extern void aes_xor_intel8(const uint8_t *src, uint8_t *dst);
#else   /* !__amd64 */
#define AES_COPY_BLOCK  aes_copy_block
#define AES_XOR_BLOCK   aes_xor_block
#endif  /* !__amd64 */
extern void aes_copy_block(uint8_t *in, uint8_t *out);
extern void aes_xor_block(uint8_t *data, uint8_t *dst);

extern void aes_copy_block(const uint8_t *src, uint8_t *dst);
extern void aes_xor_block(const uint8_t *src, uint8_t *dst);

/* Note: ctx is a pointer to aes_ctx_t defined in modes.h */
extern int aes_encrypt_contiguous_blocks(void *ctx, char *data, size_t length,
    crypto_data_t *out);
extern int aes_decrypt_contiguous_blocks(void *ctx, char *data, size_t length,
    crypto_data_t *out);

#if     defined(__amd64) && defined(_KERNEL)
/*
 * When AES floating-point acceleration is available, these will be called
 * by the worker functions to clear and restore floating point state and
 * control kernel thread preemption.
 */
extern void aes_accel_enter(void *savestate);
extern void aes_accel_exit(void *savestate);
#endif  /* __amd64 && _KERNEL */

/*
 * The following definitions and declarations are only used by AES FIPS POST
 */
#ifdef _AES_IMPL

#ifdef _KERNEL
typedef enum aes_mech_type {
        AES_ECB_MECH_INFO_TYPE,         /* SUN_CKM_AES_ECB */
        AES_CBC_MECH_INFO_TYPE,         /* SUN_CKM_AES_CBC */
        AES_CBC_PAD_MECH_INFO_TYPE,     /* SUN_CKM_AES_CBC_PAD */
        AES_CTR_MECH_INFO_TYPE,         /* SUN_CKM_AES_CTR */
        AES_CCM_MECH_INFO_TYPE,         /* SUN_CKM_AES_CCM */
        AES_GCM_MECH_INFO_TYPE,         /* SUN_CKM_AES_GCM */
        AES_GMAC_MECH_INFO_TYPE         /* SUN_CKM_AES_GMAC */
} aes_mech_type_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
    6272 Thu Apr 30 20:52:29 2015
new/usr/src/common/crypto/aes/aes_modes.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#include <sys/types.h>
#include <sys/sysmacros.h>
#include <modes/modes.h>
#include "aes_impl.h"
#ifndef _KERNEL
#include <stdlib.h>
#endif  /* !_KERNEL */

#if defined(__amd64)

/*
 * XORs a range of contiguous AES blocks in 'data' with blocks in 'dst'
 * and places the result in 'dst'. On x86-64 this exploits the 128-bit
 * floating point registers (xmm) to maximize performance.
 */
static void
aes_xor_range(const uint8_t *data, uint8_t *dst, uint64_t length)
{
        uint64_t i = 0;

        /* First use the unrolled version. */
        for (; i + 8 * AES_BLOCK_LEN <= length; i += 8 * AES_BLOCK_LEN)
                aes_xor_intel8(&data[i], &dst[i]);
        /* Finish the rest in single blocks. */
        for (; i < length; i += AES_BLOCK_LEN)
                aes_xor_intel(&data[i], &dst[i]);
}

#else   /* !__amd64 */

/*
 * XORs a range of contiguous AES blocks in 'data' with blocks in 'dst'
 * and places the result in 'dst'.
```



```
 */
static void
aes_xor_range(const uint8_t *data, uint8_t *dst, uint64_t length)
{
        uint64_t i = 0;

        if (IS_P2ALIGNED2(dst, data, sizeof (uint64_t))) {
                /* Unroll the loop to enable efficiency. */
                for (; i + 8 * AES_BLOCK_LEN < length; i += 8 * AES_BLOCK_LEN) {
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x00], &dst[i + 0x00]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x10], &dst[i + 0x10]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x20], &dst[i + 0x20]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x30], &dst[i + 0x30]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x40], &dst[i + 0x40]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x50], &dst[i + 0x50]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x60], &dst[i + 0x60]);
                        AES_XOR_BLOCK_ALIGNED(&data[i + 0x70], &dst[i + 0x70]);
                }
        }
        /* Finish the rest in single blocks. */
        for (; i < length; i += AES_BLOCK_LEN)
                AES_XOR_BLOCK(&data[i], &dst[i]);
}

#endif  /* !__amd64 */

/* Copy a 16-byte AES block from "in" to "out" */
void
aes_copy_block(const uint8_t *in, uint8_t *out)
aes_copy_block(uint8_t *in, uint8_t *out)
{
        if (IS_P2ALIGNED2(in, out, sizeof (uint32_t))) {
                AES_COPY_BLOCK_ALIGNED(in, out);
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[0] = *(uint32_t *)&in[0];
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[4] = *(uint32_t *)&in[4];
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[8] = *(uint32_t *)&in[8];
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[12] = *(uint32_t *)&in[12];
        } else {
                AES_COPY_BLOCK_UNALIGNED(in, out);
                AES_COPY_BLOCK(in, out);
        }
}


/* XOR a 16-byte AES block of data into dst */
void
aes_xor_block(const uint8_t *data, uint8_t *dst)
aes_xor_block(uint8_t *data, uint8_t *dst)
{
        if (IS_P2ALIGNED2(dst, data, sizeof (uint32_t))) {
                AES_XOR_BLOCK_ALIGNED(data, dst);
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[0] ^= *(uint32_t *)&data[0];
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[4] ^= *(uint32_t *)&data[4];
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[8] ^= *(uint32_t *)&data[8];
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[12] ^= *(uint32_t *)&data[12];
        } else {
                AES_XOR_BLOCK_UNALIGNED(data, dst);
                AES_XOR_BLOCK(data, dst);
```

```
107         }
108 }

110 /*
111  * Encrypt multiple blocks of data according to mode.
112  */
113 int
114 aes_encrypt_contiguous_blocks(void *ctx, char *data, size_t length,
115     crypto_data_t *out)
116 {
117         aes_ctx_t *aes_ctx = ctx;
118         int rv = CRYPTO_SUCCESS;
 81         int rv;

120         for (size_t i = 0; i < length; i += AES_OPSZ) {
121                 size_t opsz = MIN(length - i, AES_OPSZ);
122                 AES_ACCEL_SAVESTATE(savestate);
123                 aes_accel_enter(savestate);

125                 if (aes_ctx->ac_flags & CTR_MODE) {
126                         rv = ctr_mode_contiguous_blocks(ctx, &data[i], opsz,
127                             out, AES_BLOCK_LEN, aes_encrypt_block,
128                             AES_XOR_BLOCK, aes_ctr_mode);
 84                 rv = ctr_mode_contiguous_blocks(ctx, data, length, out,
 85                     AES_BLOCK_LEN, aes_encrypt_block, aes_xor_block);
129 #ifdef _KERNEL
130                 } else if (aes_ctx->ac_flags & CCM_MODE) {
131                         rv = ccm_mode_encrypt_contiguous_blocks(ctx, &data[i],
132                             opsz, out, AES_BLOCK_LEN, aes_encrypt_block,
133                             AES_COPY_BLOCK, AES_XOR_BLOCK);
 88                 rv = ccm_mode_encrypt_contiguous_blocks(ctx, data, length,
 89                     out, AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
 90                     aes_xor_block);
134                 } else if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE)) {
135                         rv = gcm_mode_encrypt_contiguous_blocks(ctx, &data[i],
136                             opsz, out, AES_BLOCK_LEN, aes_encrypt_block,
137                             AES_COPY_BLOCK, AES_XOR_BLOCK, aes_ctr_mode);
 92                 rv = gcm_mode_encrypt_contiguous_blocks(ctx, data, length,
 93                     out, AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
 94                     aes_xor_block);
138 #endif
139                 } else if (aes_ctx->ac_flags & CBC_MODE) {
140                         rv = cbc_encrypt_contiguous_blocks(ctx, &data[i], opsz,
141                             out, AES_BLOCK_LEN, aes_encrypt_block,
142                             AES_COPY_BLOCK, AES_XOR_BLOCK, aes_encrypt_cbc);
 97                 rv = cbc_encrypt_contiguous_blocks(ctx,
 98                     data, length, out, AES_BLOCK_LEN, aes_encrypt_block,
 99                     aes_copy_block, aes_xor_block);
143                 } else {
144                         rv = ecb_cipher_contiguous_blocks(ctx, &data[i], opsz,
145                             out, AES_BLOCK_LEN, aes_encrypt_block,
146                             aes_encrypt_ecb);
101                 rv = ecb_cipher_contiguous_blocks(ctx, data, length, out,
102                     AES_BLOCK_LEN, aes_encrypt_block);
147                 }

149                 aes_accel_exit(savestate);

151                 if (rv != CRYPTO_SUCCESS)
152                                 break;
153         }

155         return (rv);
156 }
```



```
158 /*
159  * Decrypt multiple blocks of data according to mode.
160  */
161 int
162 aes_decrypt_contiguous_blocks(void *ctx, char *data, size_t length,
163     crypto_data_t *out)
164 {
165         aes_ctx_t *aes_ctx = ctx;
166         int rv = CRYPTO_SUCCESS;
116         int rv;

169         for (size_t i = 0; i < length; i += AES_OPSZ) {
170                 size_t opsz = MIN(length - i, AES_OPSZ);
171                 AES_ACCEL_SAVESTATE(savestate);
172                 aes_accel_enter(savestate);

174                 if (aes_ctx->ac_flags & CTR_MODE) {
175                         rv = ctr_mode_contiguous_blocks(ctx, &data[i], opsz,
176                             out, AES_BLOCK_LEN, aes_encrypt_block,
177                             AES_XOR_BLOCK, aes_ctr_mode);
119                 rv = ctr_mode_contiguous_blocks(ctx, data, length, out,
120                     AES_BLOCK_LEN, aes_encrypt_block, aes_xor_block);
178                         if (rv == CRYPTO_DATA_LEN_RANGE)
179                                 rv = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
180 #ifdef _KERNEL
181                 } else if (aes_ctx->ac_flags & CCM_MODE) {
182                         rv = ccm_mode_decrypt_contiguous_blocks(ctx, &data[i],
183                             opsz, out, AES_BLOCK_LEN, aes_encrypt_block,
184                             AES_COPY_BLOCK, AES_XOR_BLOCK);
125                 rv = ccm_mode_decrypt_contiguous_blocks(ctx, data, length,
126                     out, AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
127                     aes_xor_block);
185                 } else if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE)) {
186                         rv = gcm_mode_decrypt_contiguous_blocks(ctx, &data[i],
187                             opsz, out, AES_BLOCK_LEN, aes_encrypt_block,
188                             AES_COPY_BLOCK, AES_XOR_BLOCK, aes_ctr_mode);
129                 rv = gcm_mode_decrypt_contiguous_blocks(ctx, data, length,
130                     out, AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
131                     aes_xor_block);
189 #endif
190                 } else if (aes_ctx->ac_flags & CBC_MODE) {
191                         rv = cbc_decrypt_contiguous_blocks(ctx, &data[i],
192                             opsz, out, AES_BLOCK_LEN, aes_decrypt_block,
193                             AES_COPY_BLOCK, AES_XOR_BLOCK, aes_decrypt_ecb,
194                             aes_xor_range);
134                 rv = cbc_decrypt_contiguous_blocks(ctx, data, length, out,
135                     AES_BLOCK_LEN, aes_decrypt_block, aes_copy_block,
136                     aes_xor_block);
195                 } else {
196                         rv = ecb_cipher_contiguous_blocks(ctx, &data[i],
197                             opsz, out, AES_BLOCK_LEN, aes_decrypt_block,
198                             aes_decrypt_ecb);
138                 rv = ecb_cipher_contiguous_blocks(ctx, data, length, out,
139                     AES_BLOCK_LEN, aes_decrypt_block);
199                         if (rv == CRYPTO_DATA_LEN_RANGE)
200                                 rv = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
201                 }

203                 aes_accel_exit(savestate);

205                 if (rv != CRYPTO_SUCCESS)
206                                 break;
207         }
```

```
 209         return (rv);
 210 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   43134 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/aes/amd64/aes_intel.s
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * ====================================================================
 * Written by Intel Corporation for the OpenSSL project to add support
 * for Intel AES-NI instructions. Rights for redistribution and usage
 * in source and binary forms are granted according to the OpenSSL
 * license.
 *
 *   Author: Huang Ying <ying.huang at intel dot com>
 *           Vinodh Gopal <vinodh.gopal at intel dot com>
 *           Kahraman Akdemir
 *
 * Intel AES-NI is a new set of Single Instruction Multiple Data (SIMD)
 * instructions that are going to be introduced in the next generation
 * of Intel processor, as of 2009. These instructions enable fast and
 * secure data encryption and decryption, using the Advanced Encryption
 * Standard (AES), defined by FIPS Publication number 197. The
 * architecture introduces six instructions that offer full hardware
 * support for AES. Four of them support high performance data
 * encryption and decryption, and the other two instructions support
 * the AES key expansion procedure.
 * ====================================================================
 */

/*
 * ====================================================================
 * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
```



```
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 * ====================================================================
 */

/*
 * ====================================================================
 * OpenSolaris OS modifications
 *
 * This source originates as files aes-intel.S and eng_aesni_asm.pl, in
 * patches sent sent Dec. 9, 2008 and Dec. 24, 2008, respectively, by
 * Huang Ying of Intel to the openssl-dev mailing list under the subject
 * of "Add support to Intel AES-NI instruction set for x86_64 platform".
 *
 * This OpenSolaris version has these major changes from the original source:
 *
 * 1. Added OpenSolaris ENTRY_NP/SET_SIZE macros from
 * /usr/include/sys/asm_linkage.h, lint(1B) guards, and dummy C function
 * definitions for lint.
 *
 * 2. Formatted code, added comments, and added #includes and #defines.
 *
 * 3. If bit CR0.TS is set, clear and set the TS bit, after and before
 * calling kpreempt_disable() and kpreempt_enable().
 * If the TS bit is not set, Save and restore %xmm registers at the beginning
 * and end of function calls (%xmm* registers are not saved and restored by
 * during kernel thread preemption).
 *
 * 4. Renamed functions, reordered parameters, and changed return value
 * to match OpenSolaris:
 *
 * OpenSSL interface:
 *      int intel_AES_set_encrypt_key(const unsigned char *userKey,
 *              const int bits, AES_KEY *key);
 *      int intel_AES_set_decrypt_key(const unsigned char *userKey,
 *              const int bits, AES_KEY *key);
 *      Return values for above are non-zero on error, 0 on success.
 *
 *      void intel_AES_encrypt(const unsigned char *in, unsigned char *out,
 *              const AES_KEY *key);
 *      void intel_AES_decrypt(const unsigned char *in, unsigned char *out,
 *              const AES_KEY *key);
 *      typedef struct aes_key_st {
 *              unsigned int    rd_key[4 *(AES_MAXNR + 1)];
 *              int             rounds;
 *              unsigned int    pad[3];
 *      } AES_KEY;
 * Note: AES_LONG is undefined (that is, Intel uses 32-bit key schedules
 * (ks32) instead of 64-bit (ks64).
 * Number of rounds (aka round count) is at offset 240 of AES_KEY.
 *
 * OpenSolaris OS interface (#ifdefs removed for readability):
 *      int rijndael_key_setup_dec_intel(uint32_t rk[],
 *              const uint32_t cipherKey[], uint64_t keyBits);
 *      int rijndael_key_setup_enc_intel(uint32_t rk[],
 *              const uint32_t cipherKey[], uint64_t keyBits);
 *      Return values for above are 0 on error, number of rounds on success.
 *
 *      void aes_encrypt_intel(const aes_ks_t *ks, int Nr,
```

```
 128  *              const uint32_t pt[4], uint32_t ct[4]);
 129  *      void aes_decrypt_intel(const aes_ks_t *ks, int Nr,
 130  *              const uint32_t pt[4], uint32_t ct[4]);
 131  *      typedef union {uint64_t ks64[(MAX_AES_NR + 1) * 4];
 132  *               uint32_t ks32[(MAX_AES_NR + 1) * 4]; } aes_ks_t;
 133  *
 134  *      typedef union {
 135  *              uint32_t        ks32[((MAX_AES_NR) + 1) * (MAX_AES_NB)];
 136  *      } aes_ks_t;
 137  *      typedef struct aes_key {
 138  *              aes_ks_t        encr_ks, decr_ks;
 139  *              long double     align128;
 140  *              int             flags, nr, type;
 141  *      } aes_key_t;
 142  *
 143  * Note: ks is the AES key schedule, Nr is number of rounds, pt is plain text,
 144  * ct is crypto text, and MAX_AES_NR is 14.
 145  * For the x86 64-bit architecture, OpenSolaris OS uses ks32 instead of ks64.
 146  *
 147  * Note2: aes_ks_t must be aligned on a 0 mod 128 byte boundary.
 148  *
 149  * ====================================================================
 150  */
 151 /*
 152  * Copyright 2015 by Saso Kiselkov. All rights reserved.
 153  */

 155 #if defined(lint) || defined(__lint)

 157 #include <sys/types.h>

 159 /* ARGSUSED */
 160 void
 161 aes_encrypt_intel(const uint32_t rk[], int Nr, const uint32_t pt[4],
 162     uint32_t ct[4]) {
 163 }
_____unchanged_portion_omitted_


 183 #else   /* lint */

 185 #include <sys/asm_linkage.h>
 186 #include <sys/controlregs.h>
 187 #ifdef _KERNEL
 188 #include <sys/machprivregs.h>
 189 #endif

 191 #ifdef _KERNEL
 192         /*
 193          * Note: the CLTS macro clobbers P2 (%rsi) under i86xpv.  That is,
 194          * it calls HYPERVISOR_fpu_taskswitch() which modifies %rsi when it
 195          * uses it to pass P2 to syscall.
 196          * This also occurs with the STTS macro, but we don't care if
 197          * P2 (%rsi) is modified just before function exit.
 198          * The CLTS and STTS macros push and pop P1 (%rdi) already.
 199          */
 200 #ifdef __xpv
 201 #define PROTECTED_CLTS \
 202         push    %rsi; \
 203         CLTS; \
 204         pop     %rsi
 205 #else
 206 #define PROTECTED_CLTS \
 207         CLTS
 208 #endif  /* __xpv */
```



```
 210 #define CLEAR_TS_OR_PUSH_XMM0_XMM1(tmpreg) \
 211         push    %rbp; \
 212         mov     %rsp, %rbp; \
 213         movq    %cr0, tmpreg; \
 214         testq   $CR0_TS, tmpreg; \
 215         jnz     1f; \
 216         and     $-XMM_ALIGN, %rsp; \
 217         sub     $[XMM_SIZE * 2], %rsp; \
 218         movaps  %xmm0, 16(%rsp); \
 219         movaps  %xmm1, (%rsp); \
 220         jmp     2f; \
 221 1: \
 222         PROTECTED_CLTS; \
 223 2:

 225         /*
 226          * If CR0_TS was not set above, pop %xmm0 and %xmm1 off stack,
 227          * otherwise set CR0_TS.
 228          */
 229 #define SET_TS_OR_POP_XMM0_XMM1(tmpreg) \
 230         testq   $CR0_TS, tmpreg; \
 231         jnz     1f; \
 232         movaps  (%rsp), %xmm1; \
 233         movaps  16(%rsp), %xmm0; \
 234         jmp     2f; \
 235 1: \
 236         STTS(tmpreg); \
 237 2: \
 238         mov     %rbp, %rsp; \
 239         pop     %rbp

 241         /*
 242          * If CR0_TS is not set, align stack (with push %rbp) and push
 243          * %xmm0 - %xmm6 on stack, otherwise clear CR0_TS
 244          */
 245 #define CLEAR_TS_OR_PUSH_XMM0_TO_XMM6(tmpreg) \
 246         push    %rbp; \
 247         mov     %rsp, %rbp; \
 248         movq    %cr0, tmpreg; \
 249         testq   $CR0_TS, tmpreg; \
 250         jnz     1f; \
 251         and     $-XMM_ALIGN, %rsp; \
 252         sub     $[XMM_SIZE * 7], %rsp; \
 253         movaps  %xmm0, 96(%rsp); \
 254         movaps  %xmm1, 80(%rsp); \
 255         movaps  %xmm2, 64(%rsp); \
 256         movaps  %xmm3, 48(%rsp); \
 257         movaps  %xmm4, 32(%rsp); \
 258         movaps  %xmm5, 16(%rsp); \
 259         movaps  %xmm6, (%rsp); \
 260         jmp     2f; \
 261 1: \
 262         PROTECTED_CLTS; \
 263 2:


 266         /*
 267          * If CR0_TS was not set above, pop %xmm0 - %xmm6 off stack,
 268          * otherwise set CR0_TS.
 269          */
 270 #define SET_TS_OR_POP_XMM0_TO_XMM6(tmpreg) \
 271         testq   $CR0_TS, tmpreg; \
 272         jnz     1f; \
 273         movaps  (%rsp), %xmm6; \
 274         movaps  16(%rsp), %xmm5; \
 275         movaps  32(%rsp), %xmm4; \
```

```
276         movaps  48(%rsp), %xmm3; \
277         movaps  64(%rsp), %xmm2; \
278         movaps  80(%rsp), %xmm1; \
279         movaps  96(%rsp), %xmm0; \
280         jmp     2f; \
281 1: \
282         STTS(tmpreg); \
283 2: \
284         mov     %rbp, %rsp; \
285         pop     %rbp

287 /*
288  * void aes_accel_save(void *savestate);
289  *
290  * Saves all 16 XMM registers and CR0 to a temporary location pointed to
291  * in the first argument and clears TS in CR0. This must be invoked before
292  * executing any floating point operations inside the kernel (and kernel
293  * thread preemption must be disabled as well). The memory region to which
294  * all state is saved must be at least 16x 128-bit + 64-bit long and must
295  * be 128-bit aligned.
296  */
297 ENTRY_NP(aes_accel_save)
298         movq    %cr0, %rax
299         movq    %rax, 0x100(%rdi)
300         testq   $CR0_TS, %rax
301         jnz     1f
302         movaps  %xmm0, 0x00(%rdi)
303         movaps  %xmm1, 0x10(%rdi)
304         movaps  %xmm2, 0x20(%rdi)
305         movaps  %xmm3, 0x30(%rdi)
306         movaps  %xmm4, 0x40(%rdi)
307         movaps  %xmm5, 0x50(%rdi)
308         movaps  %xmm6, 0x60(%rdi)
309         movaps  %xmm7, 0x70(%rdi)
310         movaps  %xmm8, 0x80(%rdi)
311         movaps  %xmm9, 0x90(%rdi)
312         movaps  %xmm10, 0xa0(%rdi)
313         movaps  %xmm11, 0xb0(%rdi)
314         movaps  %xmm12, 0xc0(%rdi)
315         movaps  %xmm13, 0xd0(%rdi)
316         movaps  %xmm14, 0xe0(%rdi)
317         movaps  %xmm15, 0xf0(%rdi)
318         ret
319 1:
320         PROTECTED_CLTS
321         ret
322         SET_SIZE(aes_accel_save)

324 /*
325  * void aes_accel_restore(void *savestate);
326  *
327  * Restores the saved XMM and CR0.TS state from aes_accel_save.
328  */
329 ENTRY_NP(aes_accel_restore)
330         mov     0x100(%rdi), %rax
331         testq   $CR0_TS, %rax
332         jnz     1f
333         movaps  0x00(%rdi), %xmm0
334         movaps  0x10(%rdi), %xmm1
335         movaps  0x20(%rdi), %xmm2
336         movaps  0x30(%rdi), %xmm3
337         movaps  0x40(%rdi), %xmm4
338         movaps  0x50(%rdi), %xmm5
339         movaps  0x60(%rdi), %xmm6
340         movaps  0x70(%rdi), %xmm7
341         movaps  0x80(%rdi), %xmm8
```



```
342         movaps  0x90(%rdi), %xmm9
343         movaps  0xa0(%rdi), %xmm10
344         movaps  0xb0(%rdi), %xmm11
345         movaps  0xc0(%rdi), %xmm12
346         movaps  0xd0(%rdi), %xmm13
347         movaps  0xe0(%rdi), %xmm14
348         movaps  0xf0(%rdi), %xmm15
349         ret
350 1:
351         STTS(%rax)
352         ret
353         SET_SIZE(aes_accel_restore)

355 #else
356 #define PROTECTED_CLTS
357 #define CLEAR_TS_OR_PUSH_XMM0_XMM1(tmpreg)
358 #define SET_TS_OR_POP_XMM0_XMM1(tmpreg)
359 #define CLEAR_TS_OR_PUSH_XMM0_TO_XMM6(tmpreg)
360 #define SET_TS_OR_POP_XMM0_TO_XMM6(tmpreg)
361 #endif  /* _KERNEL */


364 /*
365  * _key_expansion_128(), * _key_expansion_192a(), _key_expansion_192b(),
366  * _key_expansion_256a(), _key_expansion_256b()
367  *
368  * Helper functions called by rijndael_key_setup_inc_intel().
369  * Also used indirectly by rijndael_key_setup_dec_intel().
370  *
371  * Input:
372  * %xmm0        User-provided cipher key
373  * %xmm1        Round constant
374  * Output:
375  * (%rcx)       AES key
376  */

378 .align  16
379 _key_expansion_128:
380 _key_expansion_256a:
381         pshufd  $0b11111111, %xmm1, %xmm1
382         shufps  $0b00010000, %xmm0, %xmm4
383         pxor    %xmm4, %xmm0
384         shufps  $0b10001100, %xmm0, %xmm4
385         pxor    %xmm4, %xmm0
386         pxor    %xmm1, %xmm0
387         movaps  %xmm0, (%rcx)
388         add     $0x10, %rcx
389         ret
390         SET_SIZE(_key_expansion_128)
_____unchanged_portion_omitted_

451 /*
452  * void aes_copy_intel(const uint8_t *src, uint8_t *dst);
453  *
454  * Copies one unaligned 128-bit block from 'src' to 'dst'. The copy is
455  * performed using FPU registers, so make sure FPU state is saved when
456  * running this in the kernel.
457  */
458 ENTRY_NP(aes_copy_intel)
459         movdqu  (%rdi), %xmm0
460         movdqu  %xmm0, (%rsi)
461         ret
462         SET_SIZE(aes_copy_intel)

464 /*
465  * void aes_xor_intel(const uint8_t *src, uint8_t *dst);
```

```
 *
 * XORs one pair of unaligned 128-bit blocks from ‘src’ and ‘dst’ and
 * stores the result at ‘dst’. The XOR is performed using FPU registers,
 * so make sure FPU state is saved when running this in the kernel.
 */
ENTRY_NP(aes_xor_intel)
        movdqu  (%rdi), %xmm0
        movdqu  (%rsi), %xmm1
        pxor    %xmm1, %xmm0
        movdqu  %xmm0, (%rsi)
        ret
        SET_SIZE(aes_xor_intel)

/*
 * void aes_xor_intel8(const uint8_t *src, uint8_t *dst);
 *
 * XORs eight pairs of consecutive unaligned 128-bit blocks from ‘src’ and
 * ’dst’ and stores the results at ‘dst’. The XOR is performed using FPU
 * registers, so make sure FPU state is saved when running this in the kernel.
 */
ENTRY_NP(aes_xor_intel8)
        movdqu  0x00(%rdi), %xmm0
        movdqu  0x00(%rsi), %xmm1
        movdqu  0x10(%rdi), %xmm2
        movdqu  0x10(%rsi), %xmm3
        movdqu  0x20(%rdi), %xmm4
        movdqu  0x20(%rsi), %xmm5
        movdqu  0x30(%rdi), %xmm6
        movdqu  0x30(%rsi), %xmm7
        movdqu  0x40(%rdi), %xmm8
        movdqu  0x40(%rsi), %xmm9
        movdqu  0x50(%rdi), %xmm10
        movdqu  0x50(%rsi), %xmm11
        movdqu  0x60(%rdi), %xmm12
        movdqu  0x60(%rsi), %xmm13
        movdqu  0x70(%rdi), %xmm14
        movdqu  0x70(%rsi), %xmm15
        pxor    %xmm1, %xmm0
        pxor    %xmm3, %xmm2
        pxor    %xmm5, %xmm4
        pxor    %xmm7, %xmm6
        pxor    %xmm9, %xmm8
        pxor    %xmm11, %xmm10
        pxor    %xmm13, %xmm12
        pxor    %xmm15, %xmm14
        movdqu  %xmm0, 0x00(%rsi)
        movdqu  %xmm2, 0x10(%rsi)
        movdqu  %xmm4, 0x20(%rsi)
        movdqu  %xmm6, 0x30(%rsi)
        movdqu  %xmm8, 0x40(%rsi)
        movdqu  %xmm10, 0x50(%rsi)
        movdqu  %xmm12, 0x60(%rsi)
        movdqu  %xmm14, 0x70(%rsi)
        ret
        SET_SIZE(aes_xor_intel8)

/*
 * rijndael_key_setup_enc_intel()
 * Expand the cipher key into the encryption key schedule.
 *
 * For kernel code, caller is responsible for ensuring kpreempt_disable()
 * has been called.  This is because %xmm registers are not saved/restored.
 * Clear and set the CR0.TS bit on entry and exit, respectively,  if TS is set
 * on entry.  Otherwise, if TS is not set, save and restore %xmm registers
 * on the stack.
 *
```



```
 * OpenSolaris interface:
 * int rijndael_key_setup_enc_intel(uint32_t rk[], const uint32_t cipherKey[],
 *      uint64_t keyBits);
 * Return value is 0 on error, number of rounds on success.
 *
 * Original Intel OpenSSL interface:
 * int intel_AES_set_encrypt_key(const unsigned char *userKey,
 *      const int bits, AES_KEY *key);
 * Return value is non-zero on error, 0 on success.
 */

#ifdef  OPENSSL_INTERFACE
#define rijndael_key_setup_enc_intel    intel_AES_set_encrypt_key
#define rijndael_key_setup_dec_intel    intel_AES_set_decrypt_key

#define USERCIPHERKEY           rdi     /* P1, 64 bits */
#define KEYSIZE32               esi     /* P2, 32 bits */
#define KEYSIZE64               rsi     /* P2, 64 bits */
#define AESKEY                  rdx     /* P3, 64 bits */

#else   /* OpenSolaris Interface */
#define AESKEY                  rdi     /* P1, 64 bits */
#define USERCIPHERKEY           rsi     /* P2, 64 bits */
#define KEYSIZE32               edx     /* P3, 32 bits */
#define KEYSIZE64               rdx     /* P3, 64 bits */
#endif  /* OPENSSL_INTERFACE */

#define ROUNDS32                KEYSIZE32       /* temp */
#define ROUNDS64                KEYSIZE64       /* temp */
#define ENDAESKEY               USERCIPHERKEY   /* temp */


ENTRY_NP(rijndael_key_setup_enc_intel)
        CLEAR_TS_OR_PUSH_XMM0_TO_XMM6(%r10)

        / NULL pointer sanity check
        test    %USERCIPHERKEY, %USERCIPHERKEY
        jz      .Lenc_key_invalid_param
        test    %AESKEY, %AESKEY
        jz      .Lenc_key_invalid_param

        movups  (%USERCIPHERKEY), %xmm0 / user key (first 16 bytes)
        movaps  %xmm0, (%AESKEY)
        lea     0x10(%AESKEY), %rcx     / key addr
        pxor    %xmm4, %xmm4            / xmm4 is assumed 0 in _key_expansion_x

        cmp     $256, %KEYSIZE32
        jnz     .Lenc_key192

        / AES 256: 14 rounds in encryption key schedule
#ifdef OPENSSL_INTERFACE
        mov     $14, %ROUNDS32
        movl    %ROUNDS32, 240(%AESKEY)         / key.rounds = 14
#endif  /* OPENSSL_INTERFACE */

        movups  0x10(%USERCIPHERKEY), %xmm2     / other user key (2nd 16 bytes)
        movaps  %xmm2, (%rcx)
        add     $0x10, %rcx

        aeskeygenassist $0x1, %xmm2, %xmm1      / expand the key
        call    _key_expansion_256a
        aeskeygenassist $0x1, %xmm0, %xmm1
        call    _key_expansion_256b
        aeskeygenassist $0x2, %xmm2, %xmm1      / expand the key
        call    _key_expansion_256a
        aeskeygenassist $0x2, %xmm0, %xmm1
```

```
598         call    _key_expansion_256b
599         aeskeygenassist $0x4, %xmm2, %xmm1      / expand the key
600         call    _key_expansion_256a
601         aeskeygenassist $0x4, %xmm0, %xmm1
602         call    _key_expansion_256b
603         aeskeygenassist $0x8, %xmm2, %xmm1      / expand the key
604         call    _key_expansion_256a
605         aeskeygenassist $0x8, %xmm0, %xmm1
606         call    _key_expansion_256b
607         aeskeygenassist $0x10, %xmm2, %xmm1     / expand the key
608         call    _key_expansion_256a
609         aeskeygenassist $0x10, %xmm0, %xmm1
610         call    _key_expansion_256b
611         aeskeygenassist $0x20, %xmm2, %xmm1     / expand the key
612         call    _key_expansion_256a
613         aeskeygenassist $0x20, %xmm0, %xmm1
614         call    _key_expansion_256b
615         aeskeygenassist $0x40, %xmm2, %xmm1     / expand the key
616         call    _key_expansion_256a

618         SET_TS_OR_POP_XMM0_TO_XMM6(%r10)
619 #ifdef  OPENSSL_INTERFACE
620         xor     %rax, %rax                      / return 0 (OK)
621 #else   /* Open Solaris Interface */
622         mov     $14, %rax                       / return # rounds = 14
623 #endif
624         ret

626 .align 4
627 .Lenc_key192:
628         cmp     $192, %KEYSIZE32
629         jnz     .Lenc_key128

631         / AES 192: 12 rounds in encryption key schedule
632 #ifdef OPENSSL_INTERFACE
633         mov     $12, %ROUNDS32
634         movl    %ROUNDS32, 240(%AESKEY) / key.rounds = 12
635 #endif  /* OPENSSL_INTERFACE */

637         movq    0x10(%USERCIPHERKEY), %xmm2     / other user key
638         aeskeygenassist $0x1, %xmm2, %xmm1      / expand the key
639         call    _key_expansion_192a
640         aeskeygenassist $0x2, %xmm2, %xmm1      / expand the key
641         call    _key_expansion_192b
642         aeskeygenassist $0x4, %xmm2, %xmm1      / expand the key
643         call    _key_expansion_192a
644         aeskeygenassist $0x8, %xmm2, %xmm1      / expand the key
645         call    _key_expansion_192b
646         aeskeygenassist $0x10, %xmm2, %xmm1     / expand the key
647         call    _key_expansion_192a
648         aeskeygenassist $0x20, %xmm2, %xmm1     / expand the key
649         call    _key_expansion_192b
650         aeskeygenassist $0x40, %xmm2, %xmm1     / expand the key
651         call    _key_expansion_192a
652         aeskeygenassist $0x80, %xmm2, %xmm1     / expand the key
653         call    _key_expansion_192b

655         SET_TS_OR_POP_XMM0_TO_XMM6(%r10)
656 #ifdef  OPENSSL_INTERFACE
657         xor     %rax, %rax                      / return 0 (OK)
658 #else   /* OpenSolaris Interface */
659         mov     $12, %rax                       / return # rounds = 12
660 #endif
661         ret

663 .align 4
```



```
664 .Lenc_key128:
665         cmp $128, %KEYSIZE32
666         jnz .Lenc_key_invalid_key_bits

668         / AES 128: 10 rounds in encryption key schedule
669 #ifdef OPENSSL_INTERFACE
670         mov     $10, %ROUNDS32
671         movl    %ROUNDS32, 240(%AESKEY)         / key.rounds = 10
672 #endif  /* OPENSSL_INTERFACE */

674         aeskeygenassist $0x1, %xmm0, %xmm1      / expand the key
675         call    _key_expansion_128
676         aeskeygenassist $0x2, %xmm0, %xmm1      / expand the key
677         call    _key_expansion_128
678         aeskeygenassist $0x4, %xmm0, %xmm1      / expand the key
679         call    _key_expansion_128
680         aeskeygenassist $0x8, %xmm0, %xmm1      / expand the key
681         call    _key_expansion_128
682         aeskeygenassist $0x10, %xmm0, %xmm1     / expand the key
683         call    _key_expansion_128
684         aeskeygenassist $0x20, %xmm0, %xmm1     / expand the key
685         call    _key_expansion_128
686         aeskeygenassist $0x40, %xmm0, %xmm1     / expand the key
687         call    _key_expansion_128
688         aeskeygenassist $0x80, %xmm0, %xmm1     / expand the key
689         call    _key_expansion_128
690         aeskeygenassist $0x1b, %xmm0, %xmm1     / expand the key
691         call    _key_expansion_128
692         aeskeygenassist $0x36, %xmm0, %xmm1     / expand the key
693         call    _key_expansion_128

695         SET_TS_OR_POP_XMM0_TO_XMM6(%r10)
696 #ifdef  OPENSSL_INTERFACE
697         xor     %rax, %rax                      / return 0 (OK)
698 #else   /* OpenSolaris Interface */
699         mov     $10, %rax                       / return # rounds = 10
700 #endif
701         ret

703 .Lenc_key_invalid_param:
704 #ifdef  OPENSSL_INTERFACE
705         SET_TS_OR_POP_XMM0_TO_XMM6(%r10)
706         mov     $-1, %rax       / user key or AES key pointer is NULL
707         ret
708 #else
709         /* FALLTHROUGH */
710 #endif  /* OPENSSL_INTERFACE */

712 .Lenc_key_invalid_key_bits:
713         SET_TS_OR_POP_XMM0_TO_XMM6(%r10)
714 #ifdef  OPENSSL_INTERFACE
715         mov     $-2, %rax       / keysize is invalid
716 #else   /* Open Solaris Interface */
717         xor     %rax, %rax      / a key pointer is NULL or invalid keysize
718 #endif  /* OPENSSL_INTERFACE */

720         ret
721         SET_SIZE(rijndael_key_setup_enc_intel)
______unchanged_portion_omitted_

662 /*
663  * aes_encrypt_intel()
664  * Encrypt a single block (in and out can overlap).
665  *
666  * For kernel code, caller is responsible for ensuring kpreempt_disable()
```

```
 * has been called.  This is because %xmm registers are not saved/restored.
 * Clear and set the CR0.TS bit on entry and exit, respectively,  if TS is set
 * on entry.  Otherwise, if TS is not set, save and restore %xmm registers
 * on the stack.
 *
 * Temporary register usage:
 * %xmm0        State
 * %xmm1        Key
 *
 * Original OpenSolaris Interface:
 * void aes_encrypt_intel(const aes_ks_t *ks, int Nr,
 *      const uint32_t pt[4], uint32_t ct[4])
 *
 * Original Intel OpenSSL Interface:
 * void intel_AES_encrypt(const unsigned char *in, unsigned char *out,
 *      const AES_KEY *key)
 */

#ifdef  OPENSSL_INTERFACE
#define aes_encrypt_intel       intel_AES_encrypt
#define aes_decrypt_intel       intel_AES_decrypt

#define INP             rdi     /* P1, 64 bits */
#define OUTP            rsi     /* P2, 64 bits */
#define KEYP            rdx     /* P3, 64 bits */

/* No NROUNDS parameter--offset 240 from KEYP saved in %ecx:  */
#define NROUNDS32       ecx     /* temporary, 32 bits */
#define NROUNDS         cl      /* temporary,  8 bits */

#else   /* OpenSolaris Interface */
#define KEYP            rdi     /* P1, 64 bits */
#define NROUNDS         esi     /* P2, 32 bits */
#define INP             rdx     /* P3, 64 bits */
#define OUTP            rcx     /* P4, 64 bits */
#define LENGTH          r8      /* P5, 64 bits */
#endif  /* OPENSSL_INTERFACE */

#define KEY             xmm0    /* temporary, 128 bits */
#define STATE0          xmm8    /* temporary, 128 bits */
#define STATE1          xmm9    /* temporary, 128 bits */
#define STATE2          xmm10   /* temporary, 128 bits */
#define STATE3          xmm11   /* temporary, 128 bits */
#define STATE4          xmm12   /* temporary, 128 bits */
#define STATE5          xmm13   /* temporary, 128 bits */
#define STATE6          xmm14   /* temporary, 128 bits */
#define STATE7          xmm15   /* temporary, 128 bits */
#define STATE           xmm0    /* temporary, 128 bits */
#define KEY             xmm1    /* temporary, 128 bits */

/*
 * Runs the first two rounds of AES256 on a state register. 'op' should be
 * aesenc or aesdec.
 */
#define AES256_ROUNDS(op, statereg)     \
        movaps  -0x60(%KEYP), %KEY;     \
        op      %KEY, %statereg;        \
        movaps  -0x50(%KEYP), %KEY;     \
        op      %KEY, %statereg

/*
 * Runs the first two rounds of AES192, or the 3rd & 4th round of AES256 on
 * a state register. 'op' should be aesenc or aesdec.
 */
#define AES192_ROUNDS(op, statereg)     \
        movaps  -0x40(%KEYP), %KEY;     \
```



```
        op      %KEY, %statereg;        \
        movaps  -0x30(%KEYP), %KEY;     \
        op      %KEY, %statereg

/*
 * Runs the full 10 rounds of AES128, or the last 10 rounds of AES192/AES256
 * on a state register. 'op' should be aesenc or aesdec and 'lastop' should
 * be aesenclast or aesdeclast.
 */
#define AES128_ROUNDS(op, lastop, statereg) \
        movaps  -0x20(%KEYP), %KEY;     \
        op      %KEY, %statereg;        \
        movaps  -0x10(%KEYP), %KEY;     \
        op      %KEY, %statereg;        \
        movaps  (%KEYP), %KEY;          \
        op      %KEY, %statereg;        \
        movaps  0x10(%KEYP), %KEY;      \
        op      %KEY, %statereg;        \
        movaps  0x20(%KEYP), %KEY;      \
        op      %KEY, %statereg;        \
        movaps  0x30(%KEYP), %KEY;      \
        op      %KEY, %statereg;        \
        movaps  0x40(%KEYP), %KEY;      \
        op      %KEY, %statereg;        \
        movaps  0x50(%KEYP), %KEY;      \
        op      %KEY, %statereg;        \
        movaps  0x60(%KEYP), %KEY;      \
        op      %KEY, %statereg;        \
        movaps  0x70(%KEYP), %KEY;      \
        lastop  %KEY, %statereg

/*
 * Macros to run AES encryption rounds. Input must be prefilled in state
 * register - output will be left there as well.
 * To run AES256, invoke all of these macros in sequence. To run AES192,
 * invoke only the -192 and -128 variants. To run AES128, invoke only the
 * -128 variant.
 */
#define AES256_ENC_ROUNDS(statereg) \
        AES256_ROUNDS(aesenc, statereg)
#define AES192_ENC_ROUNDS(statereg) \
        AES192_ROUNDS(aesenc, statereg)
#define AES128_ENC_ROUNDS(statereg) \
        AES128_ROUNDS(aesenc, aesenclast, statereg)

/* Same as the AES*_ENC_ROUNDS macros, but for decryption. */
#define AES256_DEC_ROUNDS(statereg) \
        AES256_ROUNDS(aesdec, statereg)
#define AES192_DEC_ROUNDS(statereg) \
        AES192_ROUNDS(aesdec, statereg)
#define AES128_DEC_ROUNDS(statereg) \
        AES128_ROUNDS(aesdec, aesdeclast, statereg)


/*
 * aes_encrypt_intel()
 * Encrypt a single block (in and out can overlap).
 *
 * For kernel code, caller is responsible for bracketing this call with
 * disabling kernel thread preemption and calling aes_accel_save/restore().
 *
 * Temporary register usage:
 * %xmm0        Key
 * %xmm8        State
 *
 * Original OpenSolaris Interface:
```

```
 * void aes_encrypt_intel(const aes_ks_t *ks, int Nr,
 *      const uint32_t pt[4], uint32_t ct[4])
 *
 * Original Intel OpenSSL Interface:
 * void intel_AES_encrypt(const unsigned char *in, unsigned char *out,
 *      const AES_KEY *key)
 */
ENTRY_NP(aes_encrypt_intel)
        movups  (%INP), %STATE0                 / input
        movaps  (%KEYP), %KEY                   / key
        CLEAR_TS_OR_PUSH_XMM0_XMM1(%r10)

        movups  (%INP), %STATE                  / input
        movaps  (%KEYP), %KEY                   / key
#ifdef  OPENSSL_INTERFACE
        mov     240(%KEYP), %NROUNDS32          / round count
#else   /* OpenSolaris Interface */
        /* Round count is already present as P2 in %rsi/%esi */
#endif  /* OPENSSL_INTERFACE */

        pxor    %KEY, %STATE0                   / round 0
        pxor    %KEY, %STATE                    / round 0
        lea     0x30(%KEYP), %KEYP
        cmp     $12, %NROUNDS
        jb      .Lenc128
        lea     0x20(%KEYP), %KEYP
        je      .Lenc192

        / AES 256
        lea     0x20(%KEYP), %KEYP
        AES256_ENC_ROUNDS(STATE0)
        movaps  -0x60(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  -0x50(%KEYP), %KEY
        aesenc  %KEY, %STATE

.align 4
.Lenc192:
        / AES 192 and 256
        AES192_ENC_ROUNDS(STATE0)
        movaps  -0x40(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  -0x30(%KEYP), %KEY
        aesenc  %KEY, %STATE

.align 4
.Lenc128:
        / AES 128, 192, and 256
        AES128_ENC_ROUNDS(STATE0)
        movups  %STATE0, (%OUTP)                / output
        movaps  -0x20(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  -0x10(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  (%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  0x10(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  0x20(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  0x30(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  0x40(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  0x50(%KEYP), %KEY
        aesenc  %KEY, %STATE
```



```
        movaps  0x60(%KEYP), %KEY
        aesenc  %KEY, %STATE
        movaps  0x70(%KEYP), %KEY
        aesenclast      %KEY, %STATE            / last round
        movups  %STATE, (%OUTP)                 / output

        SET_TS_OR_POP_XMM0_XMM1(%r10)
        ret
        SET_SIZE(aes_encrypt_intel)


/*
 * aes_decrypt_intel()
 * Decrypt a single block (in and out can overlap).
 *
 * For kernel code, caller is responsible for bracketing this call with
 * disabling kernel thread preemption and calling aes_accel_save/restore().
 * For kernel code, caller is responsible for ensuring kpreempt_disable()
 * has been called.  This is because %xmm registers are not saved/restored.
 * Clear and set the CR0.TS bit on entry and exit, respectively,  if TS is set
 * on entry.  Otherwise, if TS is not set, save and restore %xmm registers
 * on the stack.
 *
 * Temporary register usage:
 * %xmm0        State
 * %xmm1        Key
 *
 * Original OpenSolaris Interface:
 * void aes_decrypt_intel(const aes_ks_t *ks, int Nr,
 *      const uint32_t pt[4], uint32_t ct[4])/
 *
 * Original Intel OpenSSL Interface:
 * void intel_AES_decrypt(const unsigned char *in, unsigned char *out,
 *      const AES_KEY *key);
 */
ENTRY_NP(aes_decrypt_intel)
        movups  (%INP), %STATE0                 / input
        movaps  (%KEYP), %KEY                   / key
        CLEAR_TS_OR_PUSH_XMM0_XMM1(%r10)

        movups  (%INP), %STATE                  / input
        movaps  (%KEYP), %KEY                   / key
#ifdef  OPENSSL_INTERFACE
        mov     240(%KEYP), %NROUNDS32          / round count
#else   /* OpenSolaris Interface */
        /* Round count is already present as P2 in %rsi/%esi */
#endif  /* OPENSSL_INTERFACE */

        pxor    %KEY, %STATE0                   / round 0
        pxor    %KEY, %STATE                    / round 0
        lea     0x30(%KEYP), %KEYP
        cmp     $12, %NROUNDS
        jb      .Ldec128
        lea     0x20(%KEYP), %KEYP
        je      .Ldec192

        / AES 256
        lea     0x20(%KEYP), %KEYP
        AES256_DEC_ROUNDS(STATE0)
        movaps  -0x60(%KEYP), %KEY
        aesdec  %KEY, %STATE
        movaps  -0x50(%KEYP), %KEY
        aesdec  %KEY, %STATE

.align 4
```

```
 997 .Ldec192:
 998         / AES 192 and 256
 999         AES192_DEC_ROUNDS(STATE0)
 820         movaps  -0x40(%KEYP), %KEY
 821         aesdec  %KEY, %STATE
 822         movaps  -0x30(%KEYP), %KEY
 823         aesdec  %KEY, %STATE

1001 .align 4
1002 .Ldec128:
1003         / AES 128, 192, and 256
1004         AES128_DEC_ROUNDS(STATE0)
1005         movups  %STATE0, (%OUTP)                / output
 828         movaps  -0x20(%KEYP), %KEY
 829         aesdec  %KEY, %STATE
 830         movaps  -0x10(%KEYP), %KEY
 831         aesdec  %KEY, %STATE
 832         movaps  (%KEYP), %KEY
 833         aesdec  %KEY, %STATE
 834         movaps  0x10(%KEYP), %KEY
 835         aesdec  %KEY, %STATE
 836         movaps  0x20(%KEYP), %KEY
 837         aesdec  %KEY, %STATE
 838         movaps  0x30(%KEYP), %KEY
 839         aesdec  %KEY, %STATE
 840         movaps  0x40(%KEYP), %KEY
 841         aesdec  %KEY, %STATE
 842         movaps  0x50(%KEYP), %KEY
 843         aesdec  %KEY, %STATE
 844         movaps  0x60(%KEYP), %KEY
 845         aesdec  %KEY, %STATE
 846         movaps  0x70(%KEYP), %KEY
 847         aesdeclast      %KEY, %STATE            / last round
 848         movups  %STATE, (%OUTP)                 / output

 850         SET_TS_OR_POP_XMM0_XMM1(%r10)
1007         ret
1008         SET_SIZE(aes_decrypt_intel)

1010 /* Does a pipelined load of eight input blocks into our AES state registers. */
1011 #define AES_LOAD_INPUT_8BLOCKS          \
1012         movups  0x00(%INP), %STATE0;    \
1013         movups  0x10(%INP), %STATE1;    \
1014         movups  0x20(%INP), %STATE2;    \
1015         movups  0x30(%INP), %STATE3;    \
1016         movups  0x40(%INP), %STATE4;    \
1017         movups  0x50(%INP), %STATE5;    \
1018         movups  0x60(%INP), %STATE6;    \
1019         movups  0x70(%INP), %STATE7;

1021 /* Does a pipelined store of eight AES state registers to the output. */
1022 #define AES_STORE_OUTPUT_8BLOCKS        \
1023         movups  %STATE0, 0x00(%OUTP);   \
1024         movups  %STATE1, 0x10(%OUTP);   \
1025         movups  %STATE2, 0x20(%OUTP);   \
1026         movups  %STATE3, 0x30(%OUTP);   \
1027         movups  %STATE4, 0x40(%OUTP);   \
1028         movups  %STATE5, 0x50(%OUTP);   \
1029         movups  %STATE6, 0x60(%OUTP);   \
1030         movups  %STATE7, 0x70(%OUTP);

1032 /* Performs a pipelined AES instruction with the key on all state registers. */
1033 #define AES_KEY_STATE_OP_8BLOCKS(op)    \
1034         op      %KEY, %STATE0;          \
1035         op      %KEY, %STATE1;          \
1036         op      %KEY, %STATE2;          \
```



```
1037         op      %KEY, %STATE3;          \
1038         op      %KEY, %STATE4;          \
1039         op      %KEY, %STATE5;          \
1040         op      %KEY, %STATE6;          \
1041         op      %KEY, %STATE7

1043 /* XOR all AES state regs with key to initiate encryption/decryption. */
1044 #define AES_XOR_STATE_8BLOCKS           \
1045         AES_KEY_STATE_OP_8BLOCKS(pxor)

1047 /*
1048  * Loads a round key from the key schedule offset ‘off’ into the KEY
1049  * register and performs ‘op’ using the KEY on all 8 STATE registers.
1050  */
1051 #define AES_RND_8BLOCKS(op, off)        \
1052         movaps  off(%KEYP), %KEY;       \
1053         AES_KEY_STATE_OP_8BLOCKS(op)

1055 /*
1056  * void aes_encrypt_intel8(const uint32_t roundkeys[], int numrounds,
1057  *      const void *plaintext, void *ciphertext)
1058  *
1059  * Same as aes_encrypt_intel, but performs the encryption operation on
1060  * 8 independent blocks in sequence, exploiting instruction pipelining.
1061  * This function doesn’t support the OpenSSL interface, it’s only meant
1062  * for kernel use.
1063  */
1064 ENTRY_NP(aes_encrypt_intel8)
1065         AES_LOAD_INPUT_8BLOCKS          / load input
1066         movaps  (%KEYP), %KEY           / key
1067         AES_XOR_STATE_8BLOCKS           / round 0

1069         lea     0x30(%KEYP), %KEYP      / point to key schedule
1070         cmp     $12, %NROUNDS           / determine AES variant
1071         jb      .Lenc8_128
1072         lea     0x20(%KEYP), %KEYP      / AES192 has larger key schedule
1073         je      .Lenc8_192

1075         lea     0x20(%KEYP), %KEYP      / AES256 has even larger key schedule
1076         AES_RND_8BLOCKS(aesenc, -0x60)  / AES256 R.1
1077         AES_RND_8BLOCKS(aesenc, -0x50)  / AES256 R.2

1079 .align 4
1080 .Lenc8_192:
1081         AES_RND_8BLOCKS(aesenc, -0x40)  / AES192 R.1; AES256 R.3
1082         AES_RND_8BLOCKS(aesenc, -0x30)  / AES192 R.2; AES256 R.4

1084 .align 4
1085 .Lenc8_128:
1086         AES_RND_8BLOCKS(aesenc, -0x20)  / AES128 R.1; AES192 R.3; AES256 R.5
1087         AES_RND_8BLOCKS(aesenc, -0x10)  / AES128 R.2; AES192 R.4; AES256 R.6
1088         AES_RND_8BLOCKS(aesenc, 0x00)   / AES128 R.3; AES192 R.5; AES256 R.7
1089         AES_RND_8BLOCKS(aesenc, 0x10)   / AES128 R.4; AES192 R.6; AES256 R.8
1090         AES_RND_8BLOCKS(aesenc, 0x20)   / AES128 R.5; AES192 R.7; AES256 R.9
1091         AES_RND_8BLOCKS(aesenc, 0x30)   / AES128 R.6; AES192 R.8; AES256 R.10
1092         AES_RND_8BLOCKS(aesenc, 0x40)   / AES128 R.7; AES192 R.9; AES256 R.11
1093         AES_RND_8BLOCKS(aesenc, 0x50)   / AES128 R.8; AES192 R.10; AES256 R.12
1094         AES_RND_8BLOCKS(aesenc, 0x60)   / AES128 R.9; AES192 R.11; AES256 R.13
1095         AES_RND_8BLOCKS(aesenclast, 0x70)/ AES128 R.10; AES192 R.12; AES256 R.14

1097         AES_STORE_OUTPUT_8BLOCKS        / store output
1098         ret
1099         SET_SIZE(aes_encrypt_intel8)


1102 /*
```

```
 * void aes_decrypt_intel8(const uint32_t roundkeys[], int numrounds,
 *      const void *ciphertext, void *plaintext)
 *
 * Same as aes_decrypt_intel, but performs the decryption operation on
 * 8 independent blocks in sequence, exploiting instruction pipelining.
 * This function doesn't support the OpenSSL interface, it's only meant
 * for kernel use.
 */
ENTRY_NP(aes_decrypt_intel8)
        AES_LOAD_INPUT_8BLOCKS          / load input
        movaps  (%KEYP), %KEY           / key
        AES_XOR_STATE_8BLOCKS           / round 0

        lea     0x30(%KEYP), %KEYP      / point to key schedule
        cmp     $12, %NROUNDS           / determine AES variant
        jb      .Ldec8_128
        lea     0x20(%KEYP), %KEYP      / AES192 has larger key schedule
        je      .Ldec8_192

        lea     0x20(%KEYP), %KEYP      / AES256 has even larger key schedule
        AES_RND_8BLOCKS(aesdec, -0x60)  / AES256 R.1
        AES_RND_8BLOCKS(aesdec, -0x50)  / AES256 R.2

.align 4
.Ldec8_192:
        AES_RND_8BLOCKS(aesdec, -0x40)  / AES192 R.1; AES256 R.3
        AES_RND_8BLOCKS(aesdec, -0x30)  / AES192 R.2; AES256 R.4

.align 4
.Ldec8_128:
        AES_RND_8BLOCKS(aesdec, -0x20)  / AES128 R.1; AES192 R.3; AES256 R.5
        AES_RND_8BLOCKS(aesdec, -0x10)  / AES128 R.2; AES192 R.4; AES256 R.6
        AES_RND_8BLOCKS(aesdec, 0x00)   / AES128 R.3; AES192 R.5; AES256 R.7
        AES_RND_8BLOCKS(aesdec, 0x10)   / AES128 R.4; AES192 R.6; AES256 R.8
        AES_RND_8BLOCKS(aesdec, 0x20)   / AES128 R.5; AES192 R.7; AES256 R.9
        AES_RND_8BLOCKS(aesdec, 0x30)   / AES128 R.6; AES192 R.8; AES256 R.10
        AES_RND_8BLOCKS(aesdec, 0x40)   / AES128 R.7; AES192 R.9; AES256 R.11
        AES_RND_8BLOCKS(aesdec, 0x50)   / AES128 R.8; AES192 R.10; AES256 R.12
        AES_RND_8BLOCKS(aesdec, 0x60)   / AES128 R.9; AES192 R.11; AES256 R.13
        AES_RND_8BLOCKS(aesdeclast, 0x70)/ AES128 R.10; AES192 R.12; AES256 R.14

        AES_STORE_OUTPUT_8BLOCKS        / store output
        ret
        SET_SIZE(aes_decrypt_intel8)


/*
 * This macro encapsulates the entire AES encryption algo for a single
 * block, which is prefilled in statereg and which will be replaced by
 * the encrypted output. The KEYP register must already point to the
 * AES128 key schedule ("lea 0x30(%KEYP), %KEYP" from encryption
 * function call) so that consecutive invocations of this macro are
 * supported (KEYP is restored after each invocation).
 */
#define AES_ENC(statereg, label_128, label_192, label_out)      \
        cmp     $12, %NROUNDS;                                  \
        jb      label_128;                                      \
        je      label_192;                                      \
        /* AES 256 only */                                      \
        lea     0x40(%KEYP), %KEYP;                             \
        AES256_ENC_ROUNDS(statereg);                            \
        AES192_ENC_ROUNDS(statereg);                            \
        AES128_ENC_ROUNDS(statereg);                            \
        lea     -0x40(%KEYP), %KEYP;                            \
        jmp     label_out;                                      \
.align 4;                                                       \
```



```
label_192:                                                      \
        lea     0x20(%KEYP), %KEYP;                             \
        /* AES 192 only */                                      \
        AES192_ENC_ROUNDS(statereg);                            \
        AES128_ENC_ROUNDS(statereg);                            \
        lea     -0x20(%KEYP), %KEYP;                            \
        jmp     label_out;                                      \
.align 4;                                                       \
label_128:                                                      \
        /* AES 128 only */                                      \
        AES128_ENC_ROUNDS(statereg);                            \
.align 4;                                                       \
label_out:


/*
 * void aes_encrypt_cbc_intel8(const uint32_t roundkeys[], int numrounds,
 *      const void *plaintext, void *ciphertext, const void *IV)
 *
 * Encrypts 8 consecutive AES blocks in the CBC mode. Input and output
 * may overlap. This provides a modest performance boost over invoking
 * the encryption and XOR in separate functions because we can avoid
 * copying the ciphertext block to and from memory between encryption
 * and XOR calls.
 */
#define CBC_IV                  r8      /* input - IV blk pointer */
#define CBC_IV_XMM              xmm1    /* tmp IV location for alignment */

ENTRY_NP(aes_encrypt_cbc_intel8)
        AES_LOAD_INPUT_8BLOCKS          / load input
        movaps  (%KEYP), %KEY           / key
        AES_XOR_STATE_8BLOCKS           / round 0

        lea     0x30(%KEYP), %KEYP      / point to key schedule
        movdqu  (%CBC_IV), %CBC_IV_XMM  / load IV from unaligned memory
        pxor    %CBC_IV_XMM, %STATE0    / XOR IV with input block and encrypt
        AES_ENC(STATE0, .Lenc_cbc_0_128, .Lenc_cbc_0_192, .Lenc_cbc_0_out)
        pxor    %STATE0, %STATE1
        AES_ENC(STATE1, .Lenc_cbc_1_128, .Lenc_cbc_1_192, .Lenc_cbc_1_out)
        pxor    %STATE1, %STATE2
        AES_ENC(STATE2, .Lenc_cbc_2_128, .Lenc_cbc_2_192, .Lenc_cbc_2_out)
        pxor    %STATE2, %STATE3
        AES_ENC(STATE3, .Lenc_cbc_3_128, .Lenc_cbc_3_192, .Lenc_cbc_3_out)
        pxor    %STATE3, %STATE4
        AES_ENC(STATE4, .Lenc_cbc_4_128, .Lenc_cbc_4_192, .Lenc_cbc_4_out)
        pxor    %STATE4, %STATE5
        AES_ENC(STATE5, .Lenc_cbc_5_128, .Lenc_cbc_5_192, .Lenc_cbc_5_out)
        pxor    %STATE5, %STATE6
        AES_ENC(STATE6, .Lenc_cbc_6_128, .Lenc_cbc_6_192, .Lenc_cbc_6_out)
        pxor    %STATE6, %STATE7
        AES_ENC(STATE7, .Lenc_cbc_7_128, .Lenc_cbc_7_192, .Lenc_cbc_7_out)

        AES_STORE_OUTPUT_8BLOCKS        / store output
        ret
        SET_SIZE(aes_encrypt_cbc_intel8)

/*
 * Prefills register state with counters suitable for the CTR encryption
 * mode. The counter is assumed to consist of two portions:
 * - A lower monotonically increasing 64-bit counter. If the caller wants
 *   a smaller counter, they are responsible for checking that it doesn't
 *   overflow between encryption calls.
 * - An upper static "nonce" portion, in big endian, preloaded into the
 *   lower portion of an XMM register.
 * This macro adds 'ctridx' to the lower_LE counter, swaps it to big
 * endian and by way of a temporary general-purpose register loads the
```

```
 * lower and upper counter portions into a target XMM result register,
 * which can then be handed off to the encryption process.
 */
#define PREP_CTR_BLOCKS(lower_LE, upper_BE_xmm, ctridx, tmpreg, resreg) \
        lea     ctridx(%lower_LE), %tmpreg;                             \
        bswap   %tmpreg;                                                \
        movq    %tmpreg, %resreg;                                       \
        movlhps %upper_BE_xmm, %resreg;                                 \
        pshufd  $0b01001110, %resreg, %resreg

#define CTR_UPPER_BE            r8      /* input - counter upper 64 bits (BE) */
#define CTR_UPPER_BE_XMM        xmm1    /* tmp for upper counter bits */
#define CTR_LOWER_LE            r9      /* input - counter lower 64 bits (LE) */
#define CTR_TMP0                rax     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP1                rbx     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP2                r10     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP3                r11     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP4                r12     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP5                r13     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP6                r14     /* tmp for lower 64 bit add & bswap */
#define CTR_TMP7                r15     /* tmp for lower 64 bit add & bswap */

/*
 * These are used in case CTR encryption input is unaligned before XORing.
 * Must not overlap with any STATE[0-7] register.
 */
#define TMP_INPUT0      xmm0
#define TMP_INPUT1      xmm1
#define TMP_INPUT2      xmm2
#define TMP_INPUT3      xmm3
#define TMP_INPUT4      xmm4
#define TMP_INPUT5      xmm5
#define TMP_INPUT6      xmm6
#define TMP_INPUT7      xmm7

/*
 * void aes_ctr_intel8(const uint32_t roundkeys[], int numrounds,
 *      const void *input, void *output, uint64_t counter_upper_BE,
 *      uint64_t counter_lower_LE)
 *
 * Runs AES on 8 consecutive blocks in counter mode (encryption and
 * decryption in counter mode are the same).
 */
ENTRY_NP(aes_ctr_intel8)
        /* save caller's regs */
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $0x38, %rsp
        / CTR_TMP0 is rax, no need to save
        movq    %CTR_TMP1, -0x38(%rbp)
        movq    %CTR_TMP2, -0x30(%rbp)
        movq    %CTR_TMP3, -0x28(%rbp)
        movq    %CTR_TMP4, -0x20(%rbp)
        movq    %CTR_TMP5, -0x18(%rbp)
        movq    %CTR_TMP6, -0x10(%rbp)
        movq    %CTR_TMP7, -0x08(%rbp)

        /*
         * CTR step 1: prepare big-endian formatted 128-bit counter values,
         * placing the result in the AES-NI input state registers.
         */
        movq    %CTR_UPPER_BE, %CTR_UPPER_BE_XMM
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 0, CTR_TMP0, STATE0)
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 1, CTR_TMP1, STATE1)
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 2, CTR_TMP2, STATE2)
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 3, CTR_TMP3, STATE3)
```



```
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 4, CTR_TMP4, STATE4)
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 5, CTR_TMP5, STATE5)
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 6, CTR_TMP6, STATE6)
        PREP_CTR_BLOCKS(CTR_LOWER_LE, CTR_UPPER_BE_XMM, 7, CTR_TMP7, STATE7)

        /*
         * CTR step 2: Encrypt the counters.
         */
        movaps  (%KEYP), %KEY           / key
        AES_XOR_STATE_8BLOCKS           / round 0

        /* Determine the AES variant we're going to compute */
        lea     0x30(%KEYP), %KEYP      / point to key schedule
        cmp     $12, %NROUNDS           / determine AES variant
        jb      .Lctr8_128
        lea     0x20(%KEYP), %KEYP      / AES192 has larger key schedule
        je      .Lctr8_192

        /* AES 256 */
        lea     0x20(%KEYP), %KEYP      / AES256 has even larger key schedule
        AES_RND_8BLOCKS(aesenc, -0x60)  / AES256 R.1
        AES_RND_8BLOCKS(aesenc, -0x50)  / AES256 R.2

.align 4
.Lctr8_192:
        /* AES 192 and 256 */
        AES_RND_8BLOCKS(aesenc, -0x40)  / AES192 R.1; AES256 R.3
        AES_RND_8BLOCKS(aesenc, -0x30)  / AES192 R.2; AES256 R.4

.align 4
.Lctr8_128:
        /* AES 128, 192, and 256 */
        AES_RND_8BLOCKS(aesenc, -0x20)  / AES128 R.1; AES192 R.3; AES256 R.5
        AES_RND_8BLOCKS(aesenc, -0x10)  / AES128 R.2; AES192 R.4; AES256 R.6
        AES_RND_8BLOCKS(aesenc, 0x00)   / AES128 R.3; AES192 R.5; AES256 R.7
        AES_RND_8BLOCKS(aesenc, 0x10)   / AES128 R.4; AES192 R.6; AES256 R.8
        AES_RND_8BLOCKS(aesenc, 0x20)   / AES128 R.5; AES192 R.7; AES256 R.9
        AES_RND_8BLOCKS(aesenc, 0x30)   / AES128 R.6; AES192 R.8; AES256 R.10
        AES_RND_8BLOCKS(aesenc, 0x40)   / AES128 R.7; AES192 R.9; AES256 R.11
        AES_RND_8BLOCKS(aesenc, 0x50)   / AES128 R.8; AES192 R.10; AES256 R.12
        AES_RND_8BLOCKS(aesenc, 0x60)   / AES128 R.9; AES192 R.11; AES256 R.13
        AES_RND_8BLOCKS(aesenclast, 0x70)/ AES128 R.10; AES192 R.12; AES256 R.14

        /*
         * CTR step 3: XOR input data blocks with encrypted counters to
         * produce result.
         */
        mov     %INP, %rax              / pxor requires alignment, so check
        andq    $0xf, %rax
        jnz     .Lctr_input_unaligned
        pxor    0x00(%INP), %STATE0
        pxor    0x10(%INP), %STATE1
        pxor    0x20(%INP), %STATE2
        pxor    0x30(%INP), %STATE3
        pxor    0x40(%INP), %STATE4
        pxor    0x50(%INP), %STATE5
        pxor    0x60(%INP), %STATE6
        pxor    0x70(%INP), %STATE7
        jmp     .Lctr_out

.align 4
.Lctr_input_unaligned:
        movdqu  0x00(%INP), %TMP_INPUT0
        movdqu  0x10(%INP), %TMP_INPUT1
        movdqu  0x20(%INP), %TMP_INPUT2
        movdqu  0x30(%INP), %TMP_INPUT3
```

```
1367         movdqu  0x40(%INP), %TMP_INPUT4
1368         movdqu  0x50(%INP), %TMP_INPUT5
1369         movdqu  0x60(%INP), %TMP_INPUT6
1370         movdqu  0x70(%INP), %TMP_INPUT7
1371         pxor    %TMP_INPUT0, %STATE0
1372         pxor    %TMP_INPUT1, %STATE1
1373         pxor    %TMP_INPUT2, %STATE2
1374         pxor    %TMP_INPUT3, %STATE3
1375         pxor    %TMP_INPUT4, %STATE4
1376         pxor    %TMP_INPUT5, %STATE5
1377         pxor    %TMP_INPUT6, %STATE6
1378         pxor    %TMP_INPUT7, %STATE7

1380 .align 4
1381 .Lctr_out:
1382         /*
1383          * Step 4: Write out processed blocks to memory.
1384          */
1385         movdqu  %STATE0, 0x00(%OUTP)
1386         movdqu  %STATE1, 0x10(%OUTP)
1387         movdqu  %STATE2, 0x20(%OUTP)
1388         movdqu  %STATE3, 0x30(%OUTP)
1389         movdqu  %STATE4, 0x40(%OUTP)
1390         movdqu  %STATE5, 0x50(%OUTP)
1391         movdqu  %STATE6, 0x60(%OUTP)
1392         movdqu  %STATE7, 0x70(%OUTP)

1394         /* restore caller's regs */
1395         / CTR_TMP0 is rax, no need to restore
1396         movq    -0x38(%rbp), %CTR_TMP1
1397         movq    -0x30(%rbp), %CTR_TMP2
1398         movq    -0x28(%rbp), %CTR_TMP3
1399         movq    -0x20(%rbp), %CTR_TMP4
1400         movq    -0x18(%rbp), %CTR_TMP5
1401         movq    -0x10(%rbp), %CTR_TMP6
1402         movq    -0x08(%rbp), %CTR_TMP7
1403         leave
1404         ret
1405         SET_SIZE(aes_ctr_intel8)

1407 #endif  /* lint || __lint */
```

```
***********************************************************
   26629 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/blowfish/blowfish_impl.c
4896 Performance improvements for KCF AES modes
***********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

/*
 * Blowfish encryption/decryption and keyschedule code.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sysmacros.h>
#include <sys/strsun.h>
#include <sys/note.h>
#include <sys/byteorder.h>
#include <sys/crypto/spi.h>
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include "blowfish_impl.h"

#ifdef _KERNEL

#define BLOWFISH_ASSERT(x)      ASSERT(x)

#else /* !_KERNEL */

#include <strings.h>
#include <stdlib.h>
#define BLOWFISH_ASSERT(x)
#endif /* _KERNEL */

#if defined(__i386) || defined(__amd64)
#include <sys/byteorder.h>
#define UNALIGNED_POINTERS_PERMITTED
#endif

/*
```



```
 * Blowfish initial P box and S boxes, derived from the hex digits of PI.
 *
 * NOTE:  S boxes are placed into one large array.
 */
static const uint32_t init_P[] = {
        0x243f6a88U, 0x85a308d3U, 0x13198a2eU,
        0x03707344U, 0xa4093822U, 0x299f31d0U,
        0x082efa98U, 0xec4e6c89U, 0x452821e6U,
        0x38d01377U, 0xbe5466cfU, 0x34e90c6cU,
        0xc0ac29b7U, 0xc97c50ddU, 0x3f84d5b5U,
        0xb5470917U, 0x9216d5d9U, 0x8979fb1bU
};
______unchanged_portion_omitted_

void
blowfish_copy_block(const uint8_t *in, uint8_t *out)
blowfish_copy_block(uint8_t *in, uint8_t *out)
{
        if (IS_P2ALIGNED(in, sizeof (uint32_t)) &&
            IS_P2ALIGNED(out, sizeof (uint32_t))) {
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[0] = *(uint32_t *)&in[0];
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[4] = *(uint32_t *)&in[4];
        } else {
                BLOWFISH_COPY_BLOCK(in, out);
        }
}

/* XOR block of data into dest */
void
blowfish_xor_block(const uint8_t *data, uint8_t *dst)
blowfish_xor_block(uint8_t *data, uint8_t *dst)
{
        if (IS_P2ALIGNED(dst, sizeof (uint32_t)) &&
            IS_P2ALIGNED(data, sizeof (uint32_t))) {
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[0] ^= *(uint32_t *)&data[0];
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[4] ^= *(uint32_t *)&data[4];
        } else {
                BLOWFISH_XOR_BLOCK(data, dst);
        }
}

/*
 * Encrypt multiple blocks of data according to mode.
 */
int
blowfish_encrypt_contiguous_blocks(void *ctx, char *data, size_t length,
    crypto_data_t *out)
{
        blowfish_ctx_t *blowfish_ctx = ctx;
        int rv;

        if (blowfish_ctx->bc_flags & CBC_MODE) {
                rv = cbc_encrypt_contiguous_blocks(ctx, data, length, out,
                    BLOWFISH_BLOCK_LEN, blowfish_encrypt_block,
                    blowfish_copy_block, blowfish_xor_block, NULL);
                    blowfish_copy_block, blowfish_xor_block);
        } else {
                rv = ecb_cipher_contiguous_blocks(ctx, data, length, out,
                    BLOWFISH_BLOCK_LEN, blowfish_encrypt_block, NULL);
                    BLOWFISH_BLOCK_LEN, blowfish_encrypt_block);
        }
        return (rv);
```

```
 752 }

 754 /*
 755  * Decrypt multiple blocks of data according to mode.
 756  */
 757 int
 758 blowfish_decrypt_contiguous_blocks(void *ctx, char *data, size_t length,
 759     crypto_data_t *out)
 760 {
 761         blowfish_ctx_t *blowfish_ctx = ctx;
 762         int rv;

 764         if (blowfish_ctx->bc_flags & CBC_MODE) {
 765                 rv = cbc_decrypt_contiguous_blocks(ctx, data, length, out,
 766                     BLOWFISH_BLOCK_LEN, blowfish_decrypt_block,
 767                     blowfish_copy_block, blowfish_xor_block, NULL, NULL);
 764                     blowfish_copy_block, blowfish_xor_block);
 768         } else {
 769                 rv = ecb_cipher_contiguous_blocks(ctx, data, length, out,
 770                     BLOWFISH_BLOCK_LEN, blowfish_decrypt_block, NULL);
 767                     BLOWFISH_BLOCK_LEN, blowfish_decrypt_block);
 771                 if (rv == CRYPTO_DATA_LEN_RANGE)
 772                         rv = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
 773         }
 774         return (rv);
 775 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    2508 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/blowfish/blowfish_impl.h
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#ifndef _BLOWFISH_IMPL_H
#define _BLOWFISH_IMPL_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * Common definitions used by Blowfish.
 */

#ifdef  __cplusplus
extern "C" {
#endif

#define BLOWFISH_COPY_BLOCK(src, dst) \
        (dst)[0] = (src)[0]; \
        (dst)[1] = (src)[1]; \
        (dst)[2] = (src)[2]; \
        (dst)[3] = (src)[3]; \
        (dst)[4] = (src)[4]; \
        (dst)[5] = (src)[5]; \
        (dst)[6] = (src)[6]; \
        (dst)[7] = (src)[7]

#define BLOWFISH_XOR_BLOCK(src, dst) \
        (dst)[0] ^= (src)[0]; \
        (dst)[1] ^= (src)[1]; \
        (dst)[2] ^= (src)[2]; \
        (dst)[3] ^= (src)[3]; \
        (dst)[4] ^= (src)[4]; \
        (dst)[5] ^= (src)[5]; \
        (dst)[6] ^= (src)[6]; \
        (dst)[7] ^= (src)[7]
```



```
#define BLOWFISH_MINBITS        32
#define BLOWFISH_MINBYTES       (BLOWFISH_MINBITS >> 3)
#define BLOWFISH_MAXBITS        448
#define BLOWFISH_MAXBYTES       (BLOWFISH_MAXBITS >> 3)

#define BLOWFISH_IV_LEN         8
#define BLOWFISH_BLOCK_LEN      8
#define BLOWFISH_KEY_INCREMENT  8
#define BLOWFISH_DEFAULT        128

extern int blowfish_encrypt_contiguous_blocks(void *, char *, size_t,
    crypto_data_t *);
extern int blowfish_decrypt_contiguous_blocks(void *, char *, size_t,
    crypto_data_t *);
extern int blowfish_encrypt_block(const void *, const uint8_t *, uint8_t *);
extern int blowfish_decrypt_block(const void *, const uint8_t *, uint8_t *);
extern void blowfish_init_keysched(uint8_t *, uint_t, void *);
extern void *blowfish_alloc_keysched(size_t *, int);
extern void blowfish_copy_block(const uint8_t *, uint8_t *);
extern void blowfish_xor_block(const uint8_t *, uint8_t *);
extern void blowfish_copy_block(uint8_t *, uint8_t *);
extern void blowfish_xor_block(uint8_t *, uint8_t *);
#ifdef  __cplusplus
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   47234 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/des/des_impl.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sysmacros.h>
#include <sys/strsun.h>
#include <sys/crypto/spi.h>
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include "des_impl.h"
#ifndef _KERNEL
#include <strings.h>
#include <stdlib.h>
#endif  /* !_KERNEL */

#if defined(__i386) || defined(__amd64)
#include <sys/byteorder.h>
#define UNALIGNED_POINTERS_PERMITTED
#endif

typedef struct keysched_s {
        uint64_t ksch_encrypt[16];
        uint64_t ksch_decrypt[16];
} keysched_t;
```

*unchanged_portion_omitted_*

```
void
des_copy_block(const uint8_t *in, uint8_t *out)
des_copy_block(uint8_t *in, uint8_t *out)
{
        if (IS_P2ALIGNED(in, sizeof (uint32_t)) &&
            IS_P2ALIGNED(out, sizeof (uint32_t))) {
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[0] = *(uint32_t *)&in[0];
```



```
                /* LINTED: pointer alignment */
                *(uint32_t *)&out[4] = *(uint32_t *)&in[4];
        } else {
                DES_COPY_BLOCK(in, out);
        }
}

/* XOR block of data into dest */
void
des_xor_block(const uint8_t *data, uint8_t *dst)
des_xor_block(uint8_t *data, uint8_t *dst)
{
        if (IS_P2ALIGNED(dst, sizeof (uint32_t)) &&
            IS_P2ALIGNED(data, sizeof (uint32_t))) {
                /* LINTED: pointer alignment */
                *(uint32_t *)&dst[0] ^=
                    /* LINTED: pointer alignment */
                    *(uint32_t *)&data[0];
                    /* LINTED: pointer alignment */
                *(uint32_t *)&dst[4] ^=
                    /* LINTED: pointer alignment */
                    *(uint32_t *)&data[4];
        } else {
                DES_XOR_BLOCK(data, dst);
        }
}
```

*unchanged_portion_omitted_*

```
/*
 * Encrypt multiple blocks of data according to mode.
 */
int
des_encrypt_contiguous_blocks(void *ctx, char *data, size_t length,
    crypto_data_t *out)
{
        des_ctx_t *des_ctx = ctx;
        int rv;

        if (des_ctx->dc_flags & DES3_STRENGTH) {
                if (des_ctx->dc_flags & CBC_MODE) {
                        rv = cbc_encrypt_contiguous_blocks(ctx, data,
                            length, out, DES_BLOCK_LEN, des3_encrypt_block,
                            des_copy_block, des_xor_block, NULL);
                            des_copy_block, des_xor_block);
                } else {
                        rv = ecb_cipher_contiguous_blocks(ctx, data, length,
                            out, DES_BLOCK_LEN, des3_encrypt_block, NULL);
                            out, DES_BLOCK_LEN, des3_encrypt_block);
                }
        } else {
                if (des_ctx->dc_flags & CBC_MODE) {
                        rv = cbc_encrypt_contiguous_blocks(ctx, data,
                            length, out, DES_BLOCK_LEN, des_encrypt_block,
                            des_copy_block, des_xor_block, NULL);
                            des_copy_block, des_xor_block);
                } else {
                        rv = ecb_cipher_contiguous_blocks(ctx, data, length,
                            out, DES_BLOCK_LEN, des_encrypt_block, NULL);
                            out, DES_BLOCK_LEN, des_encrypt_block);
                }
        }
        return (rv);
}

/*
 * Decrypt multiple blocks of data according to mode.
```

```
 */
int
des_decrypt_contiguous_blocks(void *ctx, char *data, size_t length,
    crypto_data_t *out)
{
        des_ctx_t *des_ctx = ctx;
        int rv;

        if (des_ctx->dc_flags & DES3_STRENGTH) {
                if (des_ctx->dc_flags & CBC_MODE) {
                        rv = cbc_decrypt_contiguous_blocks(ctx, data,
                            length, out, DES_BLOCK_LEN, des3_decrypt_block,
                            des_copy_block, des_xor_block, NULL, NULL);
                            des_copy_block, des_xor_block);
                } else {
                        rv = ecb_cipher_contiguous_blocks(ctx, data, length,
                            out, DES_BLOCK_LEN, des3_decrypt_block, NULL);
                            out, DES_BLOCK_LEN, des3_decrypt_block);
                        if (rv == CRYPTO_DATA_LEN_RANGE)
                                rv = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
                }
        } else {
                if (des_ctx->dc_flags & CBC_MODE) {
                        rv = cbc_decrypt_contiguous_blocks(ctx, data,
                            length, out, DES_BLOCK_LEN, des_decrypt_block,
                            des_copy_block, des_xor_block, NULL, NULL);
                            des_copy_block, des_xor_block);
                } else {
                        rv = ecb_cipher_contiguous_blocks(ctx, data, length,
                            out, DES_BLOCK_LEN, des_decrypt_block, NULL);
                            out, DES_BLOCK_LEN, des_decrypt_block);
                        if (rv == CRYPTO_DATA_LEN_RANGE)
                                rv = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
                }
        }
        return (rv);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
    3889 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/des/des_impl.h
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#ifndef _DES_IMPL_H
#define _DES_IMPL_H

/*
 * Common definitions used by DES
 */

#ifdef  __cplusplus
extern "C" {
#endif

#define DES_BLOCK_LEN   8

#define DES_COPY_BLOCK(src, dst) \
        (dst)[0] = (src)[0]; \
        (dst)[1] = (src)[1]; \
        (dst)[2] = (src)[2]; \
        (dst)[3] = (src)[4]; \
        (dst)[4] = (src)[4]; \
        (dst)[5] = (src)[5]; \
        (dst)[6] = (src)[6]; \
        (dst)[7] = (src)[7];

#define DES_XOR_BLOCK(src, dst) \
        (dst)[0] ^= (src)[0]; \
        (dst)[1] ^= (src)[1]; \
        (dst)[2] ^= (src)[2]; \
        (dst)[3] ^= (src)[3]; \
        (dst)[4] ^= (src)[4]; \
        (dst)[5] ^= (src)[5]; \
        (dst)[6] ^= (src)[6]; \
        (dst)[7] ^= (src)[7]
```



```
typedef enum des_strength {
        DES = 1,
        DES2,
        DES3
} des_strength_t;

#define DES3_STRENGTH   0x08000000

#define DES_KEYSIZE     8
#define DES_MINBITS     64
#define DES_MAXBITS     64
#define DES_MINBYTES    (DES_MINBITS / 8)
#define DES_MAXBYTES    (DES_MAXBITS / 8)
#define DES_IV_LEN      8

#define DES2_KEYSIZE    (2 * DES_KEYSIZE)
#define DES2_MINBITS    (2 * DES_MINBITS)
#define DES2_MAXBITS    (2 * DES_MAXBITS)
#define DES2_MINBYTES   (DES2_MINBITS / 8)
#define DES2_MAXBYTES   (DES2_MAXBITS / 8)

#define DES3_KEYSIZE    (3 * DES_KEYSIZE)
#define DES3_MINBITS    (2 * DES_MINBITS)       /* DES3 handles CKK_DES2 keys */
#define DES3_MAXBITS    (3 * DES_MAXBITS)
#define DES3_MINBYTES   (DES3_MINBITS / 8)
#define DES3_MAXBYTES   (DES3_MAXBITS / 8)

extern int des_encrypt_contiguous_blocks(void *, char *, size_t,
    crypto_data_t *);
extern int des_decrypt_contiguous_blocks(void *, char *, size_t,
    crypto_data_t *);
extern uint64_t des_crypt_impl(uint64_t *, uint64_t, int);
extern void des_ks(uint64_t *, uint64_t);
extern int des_crunch_block(const void *, const uint8_t *, uint8_t *,
    boolean_t);
extern int des3_crunch_block(const void *, const uint8_t *, uint8_t *,
    boolean_t);
extern void des_init_keysched(uint8_t *, des_strength_t, void *);
extern void *des_alloc_keysched(size_t *, des_strength_t, int);
extern boolean_t des_keycheck(uint8_t *, des_strength_t, uint8_t *);
extern void des_parity_fix(uint8_t *, des_strength_t, uint8_t *);
extern void des_copy_block(const uint8_t *, uint8_t *);
extern void des_xor_block(const uint8_t *, uint8_t *);
extern void des_copy_block(uint8_t *, uint8_t *);
extern void des_xor_block(uint8_t *, uint8_t *);
extern int des_encrypt_block(const void *, const uint8_t *, uint8_t *);
extern int des3_encrypt_block(const void *, const uint8_t *, uint8_t *);
extern int des_decrypt_block(const void *, const uint8_t *, uint8_t *);
extern int des3_decrypt_block(const void *, const uint8_t *, uint8_t *);

#ifdef _DES_IMPL

#ifdef _KERNEL
typedef enum des_mech_type {
        DES_ECB_MECH_INFO_TYPE,         /* SUN_CKM_DES_ECB */
        DES_CBC_MECH_INFO_TYPE,         /* SUN_CKM_DES_CBC */
        DES_CFB_MECH_INFO_TYPE,         /* SUN_CKM_DES_CFB */
        DES3_ECB_MECH_INFO_TYPE,        /* SUN_CKM_DES3_ECB */
        DES3_CBC_MECH_INFO_TYPE,        /* SUN_CKM_DES3_CBC */
        DES3_CFB_MECH_INFO_TYPE         /* SUN_CKM_DES3_CFB */
} des_mech_type_t;
______unchanged_portion_omitted_
```

```
**********************************************************
    9645 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/modes/amd64/gcm_intel.s
4896 Performance improvements for KCF AES modes
**********************************************************
/*
 * 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 (c) 2009 Intel Corporation
 * All Rights Reserved.
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

/*
 * Accelerated GHASH implementation with Intel PCLMULQDQ-NI
 * instructions.  This file contains an accelerated
 * Galois Field Multiplication implementation.
 *
 * PCLMULQDQ is used to accelerate the most time-consuming part of GHASH,
 * carry-less multiplication. More information about PCLMULQDQ can be
 * found at:
 * http://software.intel.com/en-us/articles/
 * carry-less-multiplication-and-its-usage-for-computing-the-gcm-mode/
 *
 */

/*
 * ====================================================================
 * OpenSolaris OS modifications
 *
 * This source originates as file galois_hash_asm.c from
 * Intel Corporation dated September 21, 2009.
 *
 * This OpenSolaris version has these major changes from the original source:
 *
 * 1. Added OpenSolaris ENTRY_NP/SET_SIZE macros from
 * /usr/include/sys/asm_linkage.h, lint(1B) guards, and a dummy C function
 * definition for lint.
 *
 * 2. Formatted code, added comments, and added #includes and #defines.
 *
```



```
 * 3. If bit CR0.TS is set, clear and set the TS bit, after and before
 * calling kpreempt_disable() and kpreempt_enable().
 * If the TS bit is not set, Save and restore %xmm registers at the beginning
 * and end of function calls (%xmm* registers are not saved and restored by
 * during kernel thread preemption).
 *
 * 4. Removed code to perform hashing.  This is already done with C macro
 * GHASH in gcm.c.  For better performance, this removed code should be
 * reintegrated in the future to replace the C GHASH macro.
 *
 * 5. Added code to byte swap 16-byte input and output.
 *
 * 6. Folded in comments from the original C source with embedded assembly
 * (SB_w_shift_xor.c)
 *
 * 7. Renamed function and reordered parameters to match OpenSolaris:
 * Intel interface:
 *      void galois_hash_asm(unsigned char *hk, unsigned char *s,
 *              unsigned char *d, int length)
 * OpenSolaris OS interface:
 *      void gcm_mul_pclmulqdq(uint64_t *x_in, uint64_t *y, uint64_t *res);
 * ====================================================================
 */


#if defined(lint) || defined(__lint)

#include <sys/types.h>

/* ARGSUSED */
void
gcm_mul_pclmulqdq(uint64_t *x_in, uint64_t *y, uint64_t *res) {
}

#ifdef  _KERNEL
/*ARGSUSED*/
void
gcm_intel_save(void *savestate)
{
}

/*ARGSUSED*/
void
gcm_accel_restore(void *savestate)
{
}
#endif  /* _KERNEL */

#else   /* lint */

#include <sys/asm_linkage.h>
#include <sys/controlregs.h>
#ifdef _KERNEL
#include <sys/machprivregs.h>
#endif

#ifdef _KERNEL
        /*
         * Note: the CLTS macro clobbers P2 (%rsi) under i86xpv.  That is,
         * it calls HYPERVISOR_fpu_taskswitch() which modifies %rsi when it
         * uses it to pass P2 to syscall.
         * This also occurs with the STTS macro, but we don't care if
         * P2 (%rsi) is modified just before function exit.
         * The CLTS and STTS macros push and pop P1 (%rdi) already.
         */
#ifdef __xpv
```

```
#define PROTECTED_CLTS \
        push    %rsi; \
        CLTS; \
        pop     %rsi
#else
#define PROTECTED_CLTS \
        CLTS
#endif  /* __xpv */
#endif  /* _KERNEL */

.text
.align XMM_ALIGN
/*
 * Use this mask to byte-swap a 16-byte integer with the pshufb instruction:
 * static uint8_t byte_swap16_mask[] = {
 *      15, 14, 13, 12, 11, 10, 9, 8, 7, 6 ,5, 4, 3, 2, 1, 0 };
        /*
         * If CR0_TS is not set, align stack (with push %rbp) and push
         * %xmm0 - %xmm10 on stack, otherwise clear CR0_TS
 */
.Lbyte_swap16_mask:
        .byte   15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
#define CLEAR_TS_OR_PUSH_XMM_REGISTERS(tmpreg) \
        push    %rbp; \
        mov     %rsp, %rbp; \
        movq    %cr0, tmpreg; \
        testq   $CR0_TS, tmpreg; \
        jnz     1f; \
        and     $-XMM_ALIGN, %rsp; \
        sub     $[XMM_SIZE * 11], %rsp; \
        movaps  %xmm0, 160(%rsp); \
        movaps  %xmm1, 144(%rsp); \
        movaps  %xmm2, 128(%rsp); \
        movaps  %xmm3, 112(%rsp); \
        movaps  %xmm4, 96(%rsp); \
        movaps  %xmm5, 80(%rsp); \
        movaps  %xmm6, 64(%rsp); \
        movaps  %xmm7, 48(%rsp); \
        movaps  %xmm8, 32(%rsp); \
        movaps  %xmm9, 16(%rsp); \
        movaps  %xmm10, (%rsp); \
        jmp     2f; \
1: \
        PROTECTED_CLTS; \
2:

#ifdef  _KERNEL
/*
 * void gcm_intel_save(void *savestate)
 *
 * Saves the XMM0--XMM14 registers and CR0 to a temporary location pointed
 * to in the first argument and clears TS in CR0. This must be invoked before
 * executing accelerated GCM computations inside the kernel (and kernel
 * thread preemption must be disabled as well). The memory region to which
 * all state is saved must be at least 16x 128-bit + 64-bit long and must
 * be 128-bit aligned.

        /*
         * If CR0_TS was not set above, pop %xmm0 - %xmm10 off stack,
         * otherwise set CR0_TS.
 */
ENTRY_NP(gcm_accel_save)
        movq    %cr0, %rax
        movq    %rax, 0x100(%rdi)
        testq   $CR0_TS, %rax
        jnz     1f
```



```
        /* FPU is in use, save registers */
        movaps  %xmm0, 0x00(%rdi)
        movaps  %xmm1, 0x10(%rdi)
        movaps  %xmm2, 0x20(%rdi)
        movaps  %xmm3, 0x30(%rdi)
        movaps  %xmm4, 0x40(%rdi)
        movaps  %xmm5, 0x50(%rdi)
        movaps  %xmm6, 0x60(%rdi)
        movaps  %xmm7, 0x70(%rdi)
        movaps  %xmm8, 0x80(%rdi)
        movaps  %xmm9, 0x90(%rdi)
        movaps  %xmm10, 0xa0(%rdi)
        movaps  %xmm11, 0xb0(%rdi)
        movaps  %xmm12, 0xc0(%rdi)
        movaps  %xmm13, 0xd0(%rdi)
        movaps  %xmm14, 0xe0(%rdi)
        movaps  %xmm15, 0xf0(%rdi)
        ret
1:
        PROTECTED_CLTS
        ret
        SET_SIZE(gcm_accel_save)
#define SET_TS_OR_POP_XMM_REGISTERS(tmpreg) \
        testq   $CR0_TS, tmpreg; \
        jnz     1f; \
        movaps  (%rsp), %xmm10; \
        movaps  16(%rsp), %xmm9; \
        movaps  32(%rsp), %xmm8; \
        movaps  48(%rsp), %xmm7; \
        movaps  64(%rsp), %xmm6; \
        movaps  80(%rsp), %xmm5; \
        movaps  96(%rsp), %xmm4; \
        movaps  112(%rsp), %xmm3; \
        movaps  128(%rsp), %xmm2; \
        movaps  144(%rsp), %xmm1; \
        movaps  160(%rsp), %xmm0; \
        jmp     2f; \
1: \
        STTS(tmpreg); \
2: \
        mov     %rbp, %rsp; \
        pop     %rbp

#else
#define PROTECTED_CLTS
#define CLEAR_TS_OR_PUSH_XMM_REGISTERS(tmpreg)
#define SET_TS_OR_POP_XMM_REGISTERS(tmpreg)
#endif  /* _KERNEL */

/*
 * void gcm_accel_restore(void *savestate)
 *
 * Restores the saved XMM and CR0.TS state from aes_accel_save.
 * Use this mask to byte-swap a 16-byte integer with the pshufb instruction
 */
ENTRY_NP(gcm_accel_restore)
        movq    0x100(%rdi), %rax
        testq   $CR0_TS, %rax
        jnz     1f
        movaps  0x00(%rdi), %xmm0
        movaps  0x10(%rdi), %xmm1
        movaps  0x20(%rdi), %xmm2
        movaps  0x30(%rdi), %xmm3
        movaps  0x40(%rdi), %xmm4
        movaps  0x50(%rdi), %xmm5
```

```
        movaps  0x60(%rdi), %xmm6
        movaps  0x70(%rdi), %xmm7
        movaps  0x80(%rdi), %xmm8
        movaps  0x90(%rdi), %xmm9
        movaps  0xa0(%rdi), %xmm10
        movaps  0xb0(%rdi), %xmm11
        movaps  0xc0(%rdi), %xmm12
        movaps  0xd0(%rdi), %xmm13
        movaps  0xe0(%rdi), %xmm14
        movaps  0xf0(%rdi), %xmm15
        ret
1:
        STTS(%rax)
        ret
        SET_SIZE(gcm_accel_restore)
```

```
#endif  /* _KERNEL */
// static uint8_t byte_swap16_mask[] = {
//       15, 14, 13, 12, 11, 10, 9, 8, 7, 6 ,5, 4, 3, 2, 1, 0 };
.text
.align XMM_ALIGN
.Lbyte_swap16_mask:
        .byte   15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

```
/*
 * void gcm_mul_pclmulqdq(uint64_t *x_in, uint64_t *y, uint64_t *res);
 *
 * Perform a carry-less multiplication (that is, use XOR instead of the
 * multiply operator) on P1 and P2 and place the result in P3.
 *
 * Byte swap the input and the output.
 *
 * Note: x_in, y, and res all point to a block of 16-byte numbers
 * Note: x_in, y, and res all point to a block of 20-byte numbers
 * (an array of two 64-bit integers).
 *
 * Note2: For kernel code, caller is responsible for bracketing this call with
 * disabling kernel thread preemption and calling gcm_accel_save/restore().
 * Note2: For kernel code, caller is responsible for ensuring
 * kpreempt_disable() has been called.  This is because %xmm registers are
 * not saved/restored.  Clear and set the CR0.TS bit on entry and exit,
 * respectively, if TS is set on entry.  Otherwise, if TS is not set,
 * save and restore %xmm registers on the stack.
 *
 * Note3: Original Intel definition:
 * void galois_hash_asm(unsigned char *hk, unsigned char *s,
 *      unsigned char *d, int length)
 *
 * Note4: Register/parameter mapping:
 * Intel:
 *      Parameter 1: %rcx (copied to %xmm0)     hk or x_in
 *      Parameter 2: %rdx (copied to %xmm1)     s or y
 *      Parameter 3: %rdi (result)              d or res
 * OpenSolaris:
 *      Parameter 1: %rdi (copied to %xmm0)     x_in
 *      Parameter 2: %rsi (copied to %xmm1)     y
 *      Parameter 3: %rdx (result)              res
 */

ENTRY_NP(gcm_mul_pclmulqdq)
        CLEAR_TS_OR_PUSH_XMM_REGISTERS(%r10)

        //
        // Copy Parameters
```



```
        //
        movdqu  (%rdi), %xmm0   // P1
        movdqu  (%rsi), %xmm1   // P2

        //
        // Byte swap 16-byte input
        //
        lea     .Lbyte_swap16_mask(%rip), %rax
        movaps  (%rax), %xmm10
        pshufb  %xmm10, %xmm0
        pshufb  %xmm10, %xmm1


        //
        // Multiply with the hash key
        //
        movdqu  %xmm0, %xmm3
        pclmulqdq $0, %xmm1, %xmm3      // xmm3 holds a0*b0

        movdqu  %xmm0, %xmm4
        pclmulqdq $16, %xmm1, %xmm4     // xmm4 holds a0*b1

        movdqu  %xmm0, %xmm5
        pclmulqdq $1, %xmm1, %xmm5      // xmm5 holds a1*b0
        movdqu  %xmm0, %xmm6
        pclmulqdq $17, %xmm1, %xmm6     // xmm6 holds a1*b1

        pxor    %xmm5, %xmm4    // xmm4 holds a0*b1 + a1*b0

        movdqu  %xmm4, %xmm5    // move the contents of xmm4 to xmm5
        psrldq  $8, %xmm4       // shift by xmm4 64 bits to the right
        pslldq  $8, %xmm5       // shift by xmm5 64 bits to the left
        pxor    %xmm5, %xmm3
        pxor    %xmm4, %xmm6    // Register pair <xmm6:xmm3> holds the result
                                // of the carry-less multiplication of
                                // xmm0 by xmm1.

        // We shift the result of the multiplication by one bit position
        // to the left to cope for the fact that the bits are reversed.
        movdqu  %xmm3, %xmm7
        movdqu  %xmm6, %xmm8
        pslld   $1, %xmm3
        pslld   $1, %xmm6
        psrld   $31, %xmm7
        psrld   $31, %xmm8
        movdqu  %xmm7, %xmm9
        pslldq  $4, %xmm8
        pslldq  $4, %xmm7
        psrldq  $12, %xmm9
        por     %xmm7, %xmm3
        por     %xmm8, %xmm6
        por     %xmm9, %xmm6

        //
        // First phase of the reduction
        //
        // Move xmm3 into xmm7, xmm8, xmm9 in order to perform the shifts
        // independently.
        movdqu  %xmm3, %xmm7
        movdqu  %xmm3, %xmm8
        movdqu  %xmm3, %xmm9
        pslld   $31, %xmm7      // packed right shift shifting << 31
        pslld   $30, %xmm8      // packed right shift shifting << 30
        pslld   $25, %xmm9      // packed right shift shifting << 25
        pxor    %xmm8, %xmm7    // xor the shifted versions
        pxor    %xmm9, %xmm7
```

```
 318         movdqu  %xmm7, %xmm8
 319         pslldq  $12, %xmm7
 320         psrldq  $4, %xmm8
 321         pxor    %xmm7, %xmm3    // first phase of the reduction complete

 323         //
 324         // Second phase of the reduction
 325         //
 326         // Make 3 copies of xmm3 in xmm2, xmm4, xmm5 for doing these
 327         // shift operations.
 328         movdqu  %xmm3, %xmm2
 329         movdqu  %xmm3, %xmm4    // packed left shifting >> 1
 330         movdqu  %xmm3, %xmm5
 331         psrld   $1, %xmm2
 332         psrld   $2, %xmm4       // packed left shifting >> 2
 333         psrld   $7, %xmm5       // packed left shifting >> 7
 334         pxor    %xmm4, %xmm2    // xor the shifted versions
 335         pxor    %xmm5, %xmm2
 336         pxor    %xmm8, %xmm2
 337         pxor    %xmm2, %xmm3
 338         pxor    %xmm3, %xmm6    // the result is in xmm6

 340         //
 341         // Byte swap 16-byte result
 342         //
 343         pshufb  %xmm10, %xmm6   // %xmm10 has the swap mask

 345         //
 346         // Store the result
 347         //
 348         movdqu  %xmm6, (%rdx)   // P3


 351         //
 352         // Cleanup and Return
 353         //
 333         SET_TS_OR_POP_XMM_REGISTERS(%r10)
 354         ret
 355         SET_SIZE(gcm_mul_pclmulqdq)
```

_____unchanged_portion_omitted_

```
**********************************************************
   11419 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/modes/amd64/gcm_intel_cryptogams.s
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * Copyright (c) 2013, CRYPTOGAMS by <appro@openssl.org>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *    Redistributions of source code must retain copyright notices,
 *      this list of conditions and the following disclaimer.
 *
 * *    Redistributions in binary form must reproduce the above
 *      copyright notice, this list of conditions and the following
 *      disclaimer in the documentation and/or other materials
 *      provided with the distribution.
 *
 * *    Neither the name of the CRYPTOGAMS nor the names of its
 *      copyright holder and contributors may be used to endorse or
 *      promote products derived from this software without specific
 *      prior written permission.
 *
 * ALTERNATIVELY, provided that this notice is retained in full, this
 * product may be distributed under the terms of the GNU General Public
 * License (GPL), in which case the provisions of the GPL apply INSTEAD OF
 * those given above.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
/*
 * Copyright 2015 by Saso Kiselkov on Illumos port sections.
 */

#if defined(lint) || defined(__lint)

#include <sys/types.h>

/*ARGSUSED*/
void
gcm_ghash_clmul(uint64_t ghash[2], const uint8_t Htable[256],
    const uint8_t *inp, size_t len)
{
}

/*ARGSUSED*/
void
gcm_init_clmul(const uint64_t hash_init[2], uint8_t Htable[256])
{
}

#else   /* lint */
```



```
#include <sys/asm_linkage.h>
#include <sys/controlregs.h>
#ifdef _KERNEL
#include <sys/machprivregs.h>
#endif

.text
.align XMM_ALIGN
.Lbyte_swap16_mask:
        .byte   15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
.L0x1c2_polynomial:
        .byte   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xc2
.L7_mask:
        .long   7, 0, 7, 0

#define Xi      xmm0    /* hash value */
#define Xhi     xmm1    /* hash value high order 64 bits */
#define Hkey    xmm2    /* hash key */
#define T1      xmm3    /* temp1 */
#define T2      xmm4    /* temp2 */
#define T3      xmm5    /* temp3 */
#define Xb0     xmm6    /* cipher block #0 */
#define Xb1     xmm7    /* cipher block #1 */
#define Xb2     xmm8    /* cipher block #2 */
#define Xb3     xmm9    /* cipher block #3 */
#define Xb4     xmm10   /* cipher block #4 */
#define Xb5     xmm11   /* cipher block #5 */
#define Xb6     xmm12   /* cipher block #6 */
#define Xb7     xmm13   /* cipher block #7 */

#define clmul64x64_T2(tmpreg)                           \
        movdqa          %Xi, %Xhi;                      \
        pshufd          $0b01001110, %Xi, %T1;          \
        pxor            %Xi, %T1;                       \
                                                        \
        pclmulqdq       $0x00, %Hkey, %Xi;              \
        pclmulqdq       $0x11, %Hkey, %Xhi;             \
        pclmulqdq       $0x00, %tmpreg, %T1;            \
        pxor            %Xi, %T1;                       \
        pxor            %Xhi, %T1;                      \
                                                        \
        movdqa          %T1, %T2;                       \
        psrldq          $8, %T1;                        \
        pslldq          $8, %T2;                        \
        pxor            %T1, %Xhi;                      \
        pxor            %T2, %Xi

#define reduction_alg9                                  \
        /* 1st phase */                                 \
        movdqa          %Xi, %T2;                       \
        movdqa          %Xi, %T1;                       \
        psllq           $5, %Xi;                        \
        pxor            %Xi, %T1;                       \
        psllq           $1, %Xi;                        \
        pxor            %T1, %Xi;                       \
        psllq           $57, %Xi;                       \
        movdqa          %Xi, %T1;                       \
        pslldq          $8, %Xi;                        \
        psrldq          $8, %T1;                        \
        pxor            %T2, %Xi;                       \
        pxor            %T1, %Xhi;                      \
        /* 2nd phase */                                 \
        movdqa          %Xi, %T2;                       \
        psrlq           $1, %Xi;                        \
        pxor            %T2, %Xhi;                      \
        pxor            %Xi, %T2;                       \
```

```
128         psrlq           $5, %Xi;                        \
129         pxor            %T2, %Xi;                       \
130         psrlq           $1, %Xi;                        \
131         pxor            %Xhi, %Xi

133 #define Xip     rdi
134 #define Htbl    rsi
135 #define inp     rdx
136 #define len     rcx

138 #define Xln     xmm6
139 #define Xmn     xmm7
140 #define Xhn     xmm8
141 #define Hkey2   xmm9
142 #define HK      xmm10
143 #define Xl      xmm11
144 #define Xm      xmm12
145 #define Xh      xmm13
146 #define Hkey3   xmm14
147 #define Hkey4   xmm15

149 /*
150  * void gcm_ghash_clmul(uint64_t Xi[2], const uint64_t Htable[32],
151  *      const uint8_t *inp, size_t len)
152  */
153 ENTRY_NP(gcm_ghash_clmul)
154         movdqa          .Lbyte_swap16_mask(%rip), %T3
155         mov             $0xA040608020C0E000, %rax       / ((7..0) ? 0xE0) & 0xff

157         movdqu          (%Xip), %Xi
158         movdqu          (%Htbl), %Hkey
159         movdqu          0x20(%Htbl), %HK
160         pshufb          %T3, %Xi

162         sub             $0x10, %len
163         jz              .Lodd_tail

165         movdqu          0x10(%Htbl), %Hkey2

167         cmp             $0x30, %len
168         jb              .Lskip4x

170         sub             $0x30, %len
171         movdqu          0x30(%Htbl),%Hkey3
172         movdqu          0x40(%Htbl),%Hkey4

174         /* Xi+4 =[(H*Ii+3) + (H^2*Ii+2) + (H^3*Ii+1) + H^4*(Ii+Xi)] mod P */
175         movdqu          0x30(%inp), %Xln
176         movdqu          0x20(%inp), %Xl
177         pshufb          %T3, %Xln
178         pshufb          %T3, %Xl
179         movdqa          %Xln, %Xhn
180         pshufd          $0b01001110, %Xln, %Xmn
181         pxor            %Xln, %Xmn
182         pclmulqdq       $0x00, %Hkey, %Xln
183         pclmulqdq       $0x11, %Hkey, %Xhn
184         pclmulqdq       $0x00, %HK, %Xmn

186         movdqa          %Xl, %Xh
187         pshufd          $0b01001110, %Xl, %Xm
188         pxor            %Xl, %Xm
189         pclmulqdq       $0x00, %Hkey2, %Xl
190         pclmulqdq       $0x11, %Hkey2, %Xh
191         xorps           %Xl, %Xln
192         pclmulqdq       $0x10, %HK, %Xm
193         xorps           %Xh, %Xhn
```



```
194         movups          0x50(%Htbl), %HK
195         xorps           %Xm, %Xmn

197         movdqu          0x10(%inp), %Xl
198         movdqu          0x00(%inp), %T1
199         pshufb          %T3, %Xl
200         pshufb          %T3, %T1
201         movdqa          %Xl, %Xh
202         pshufd          $0b01001110, %Xl, %Xm
203         pxor            %T1, %Xi
204         pxor            %Xl, %Xm
205         pclmulqdq       $0x00, %Hkey3, %Xl
206         movdqa          %Xi, %Xhi
207         pshufd          $0b01001110, %Xi, %T1
208         pxor            %Xi, %T1
209         pclmulqdq       $0x11, %Hkey3, %Xh
210         xorps           %Xl, %Xln
211         pclmulqdq       $0x00, %HK, %Xm
212         xorps           %Xh, %Xhn

214         lea             0x40(%inp), %inp
215         sub             $0x40, %len
216         jc              .Ltail4x

218         jmp             .Lmod4_loop

220 .align  32
221 .Lmod4_loop:
222         pclmulqdq       $0x00, %Hkey4, %Xi
223         xorps           %Xm, %Xmn
224         movdqu          0x30(%inp), %Xl
225         pshufb          %T3, %Xl
226         pclmulqdq       $0x11, %Hkey4, %Xhi
227         xorps           %Xln, %Xi
228         movdqu          0x20(%inp), %Xln
229         movdqa          %Xl, %Xh
230         pshufd          $0b01001110, %Xl, %Xm
231         pclmulqdq       $0x10, %HK, %T1
232         xorps           %Xhn, %Xhi
233         pxor            %Xl, %Xm
234         pshufb          %T3, %Xln
235         movups          0x20(%Htbl), %HK
236         pclmulqdq       $0x00, %Hkey, %Xl
237         xorps           %Xmn, %T1
238         movdqa          %Xln, %Xhn
239         pshufd          $0b01001110, %Xln, %Xmn

241         pxor            %Xi, %T1        / aggregated Karatsuba post-processing
242         pxor            %Xln, %Xmn
243         pxor            %Xhi, %T1
244         movdqa          %T1, %T2
245         pslldq          $8, %T1
246         pclmulqdq       $0x11, %Hkey, %Xh
247         psrldq          $8, %T2
248         pxor            %T1, %Xi
249         movdqa          .L7_mask(%rip), %T1
250         pxor            %T2, %Xhi
251         movq            %rax, %T2

253         pand            %Xi, %T1        / 1st phase
254         pshufb          %T1, %T2
255         pclmulqdq       $0x00, %HK, %Xm
256         pxor            %Xi, %T2
257         psllq           $57, %T2
258         movdqa          %T2, %T1
259         pslldq          $8, %T2
```

```
260         pclmulqdq       $0x00, %Hkey2, %Xln
261         psrldq          $8, %T1
262         pxor            %T2, %Xi
263         pxor            %T1, %Xhi
264         movdqu          0(%inp), %T1

266         movdqa          %Xi, %T2        / 2nd phase
267         psrlq           $1, %Xi
268         pclmulqdq       $0x11, %Hkey2, %Xhn
269         xorps           %Xl, %Xln
270         movdqu          0x10(%inp), %Xl
271         pshufb          %T3, %Xl
272         pclmulqdq       $0x10, %HK, %Xmn
273         xorps           %Xh, %Xhn
274         movups          0x50(%Htbl), %HK
275         pshufb          %T3, %T1
276         pxor            %T2, %Xhi
277         pxor            %Xi, %T2
278         psrlq           $5, %Xi

280         movdqa          %Xl, %Xh
281         pxor            %Xm, %Xmn
282         pshufd          $0b01001110, %Xl, %Xm
283         pxor            %Xl, %Xm
284         pclmulqdq       $0x00, %Hkey3, %Xl
285         pxor            %T2, %Xi
286         pxor            %T1, %Xhi
287         psrlq           $1, %Xi
288         pclmulqdq       $0x11, %Hkey3, %Xh
289         xorps           %Xl, %Xln
290         pxor            %Xhi, %Xi

292         pclmulqdq       $0x00, %HK, %Xm
293         xorps           %Xh, %Xhn

295         movdqa          %Xi, %Xhi
296         pshufd          $0b01001110, %Xi, %T1
297         pxor            %Xi, %T1

299         lea             0x40(%inp), %inp
300         sub             $0x40, %len
301         jnc             .Lmod4_loop

303 .Ltail4x:
304         pclmulqdq       $0x00, %Hkey4, %Xi
305         xorps           %Xm, %Xmn
306         pclmulqdq       $0x11, %Hkey4, %Xhi
307         xorps           %Xln, %Xi
308         pclmulqdq       $0x10, %HK, %T1
309         xorps           %Xhn, %Xhi
310         pxor            %Xi, %Xhi       / aggregated Karatsuba post-processing
311         pxor            %Xmn, %T1

313         pxor            %Xhi, %T1
314         pxor            %Xi, %Xhi

316         movdqa          %T1, %T2
317         psrldq          $8, %T1
318         pslldq          $8, %T2
319         pxor            %T1, %Xhi
320         pxor            %T2, %Xi

322         reduction_alg9

324         add     $0x40, %len
325         jz      .Ldone
```



```
326         movdqu  0x20(%Htbl), %HK
327         sub     $0x10, %len
328         jz      .Lodd_tail
329 .Lskip4x:

331         /*
332          * Xi+2 =[H*(Ii+1 + Xi+1)] mod P =
333          *      [(H*Ii+1) + (H*Xi+1)] mod P =
334          *      [(H*Ii+1) + H^2*(Ii+Xi)] mod P
335          */
336         movdqu          (%inp), %T1             / Ii
337         movdqu          16(%inp), %Xln          / Ii+1
338         pshufb          %T3, %T1
339         pshufb          %T3, %Xln
340         pxor            %T1, %Xi                / Ii+Xi

342         movdqa          %Xln, %Xhn
343         pshufd          $0b01001110, %Xln, %T1
344         pxor            %Xln, %T1
345         pclmulqdq       $0x00, %Hkey, %Xln
346         pclmulqdq       $0x11, %Hkey, %Xhn
347         pclmulqdq       $0x00, %HK, %T1

349         lea             32(%inp), %inp          / i+=2
350         sub             $0x20, %len
351         jbe             .Leven_tail
352         jmp             .Lmod_loop

354 .align  32
355 .Lmod_loop:
356         movdqa          %Xi, %Xhi
357         pshufd          $0b01001110, %Xi, %T2
358         pxor            %Xi, %T2

360         pclmulqdq       $0x00, %Hkey2, %Xi
361         pclmulqdq       $0x11, %Hkey2, %Xhi
362         pclmulqdq       $0x10, %HK, %T2

364         pxor            %Xln, %Xi               / (H*Ii+1) + H^2*(Ii+Xi)
365         pxor            %Xhn, %Xhi
366         movdqu          (%inp), %Xhn            / Ii
367         pshufb          %T3, %Xhn
368         movdqu          16(%inp), %Xln          / Ii+1

370         pxor            %Xi, %T1                / aggregated Karatsuba post-proc
371         pxor            %Xhi, %T1
372         pxor            %Xhn, %Xhi              / "Ii+Xi",  consume early
373         pxor            %T1, %T2
374         pshufb          %T3, %Xln
375         movdqa          %T2, %T1
376         psrldq          $8, %T1
377         pslldq          $8, %T2
378         pxor            %T1, %Xhi
379         pxor            %T2, %Xi

381         movdqa          %Xln, %Xhn

383         movdqa          %Xi, %T2                / 1st phase
384         movdqa          %Xi, %T1
385         psllq           $5, %Xi
386         pclmulqdq       $0x00, %Hkey, %Xln
387         pxor            %Xi, %T1
388         psllq           $1, %Xi
389         pxor            %T1, %Xi
390         psllq           $57, %Xi
391         movdqa          %Xi, %T1
```

```
392         pslldq          $8, %Xi
393         psrldq          $8, %T1
394         pxor            %T2, %Xi
395         pxor            %T1, %Xhi
396         pshufd          $0b01001110, %Xhn, %T1
397         pxor            %Xhn, %T1

399         pclmulqdq       $0x11, %Hkey, %Xhn
400         movdqa          %Xi, %T2                / 2nd phase
401         psrlq           $1, %Xi
402         pxor            %T2, %Xhi
403         pxor            %Xi, %T2
404         psrlq           $5, %Xi
405         pxor            %T2, %Xi
406         psrlq           $1, %Xi
407         pclmulqdq       $0x00, %HK, %T1
408         pxor            %Xhi, %Xi

410         lea             32(%inp), %inp
411         sub             $0x20, %len
412         ja              .Lmod_loop

414 .Leven_tail:
415         movdqa          %Xi, %Xhi
416         pshufd          $0b01001110, %Xi, %T2
417         pxor            %Xi, %T2

419         pclmulqdq       $0x00, %Hkey2, %Xi
420         pclmulqdq       $0x11, %Hkey2, %Xhi
421         pclmulqdq       $0x10, %HK, %T2

423         pxor            %Xln, %Xi               /* (H*Ii+1) + H^2*(Ii+Xi) */
424         pxor            %Xhn, %Xhi
425         pxor            %Xi, %T1
426         pxor            %Xhi, %T1
427         pxor            %T1, %T2
428         movdqa          %T2, %T1
429         psrldq          $8, %T1
430         pslldq          $8, %T2
431         pxor            %T1, %Xhi
432         pxor            %T2, %Xi

434         reduction_alg9

436         test            %len, %len
437         jnz             .Ldone

439 .Lodd_tail:
440         movdqu          (%inp), %T1                     / Ii
441         pshufb          %T3, %T1
442         pxor            %T1, %Xi                        / Ii+Xi

444         clmul64x64_T2(HK)                               / H*(Ii+Xi)
445         reduction_alg9

447 .Ldone:
448         pshufb          %T3, %Xi
449         movdqu          %Xi, (%Xip)

451         ret
452         SET_SIZE(gcm_ghash_clmul)

454 /*
455  * void gcm_init_clmul(const void *Xi, void *Htable)
456  */
457 ENTRY_NP(gcm_init_clmul)
```



```
458         movdqu          (%Xip), %Hkey
459         pshufd          $0b01001110, %Hkey, %Hkey       / dword swap

461         / <<1 twist
462         pshufd          $0b11111111, %Hkey, %T2 / broadcast uppermost dword
463         movdqa          %Hkey, %T1
464         psllq           $1, %Hkey
465         pxor            %T3, %T3
466         psrlq           $63, %T1
467         pcmpgtd         %T2, %T3                / broadcast carry bit
468         pslldq          $8, %T1
469         por             %T1, %Hkey              / H<<=1

471         / magic reduction
472         pand            .L0x1c2_polynomial(%rip), %T3
473         pxor            %T3, %Hkey              / if(carry) H^=0x1c2_polynomial

475         / calculate H^2
476         pshufd          $0b01001110, %Hkey, %HK
477         movdqa          %Hkey, %Xi
478         pxor            %Hkey, %HK

480         clmul64x64_T2(HK)
481         reduction_alg9

483         pshufd          $0b01001110, %Hkey, %T1
484         pshufd          $0b01001110, %Xi, %T2
485         pxor            %Hkey, %T1              / Karatsuba pre-processing
486         movdqu          %Hkey, 0x00(%Htbl)      / save H
487         pxor            %Xi, %T2                / Karatsuba pre-processing
488         movdqu          %Xi, 0x10(%Htbl)        / save H^2
489         palignr         $8, %T1, %T2            / low part is H.lo^H.hi...
490         movdqu          %T2, 0x20(%Htbl)        / save Karatsuba "salt"

492         clmul64x64_T2(HK)                       / H^3
493         reduction_alg9

495         movdqa          %Xi, %T3

497         clmul64x64_T2(HK)                       / H^4
498         reduction_alg9

500         pshufd          $0b01001110, %T3, %T1
501         pshufd          $0b01001110, %Xi, %T2
502         pxor            %T3, %T1                / Karatsuba pre-processing
503         movdqu          %T3, 0x30(%Htbl)        / save H^3
504         pxor            %Xi, %T2                / Karatsuba pre-processing
505         movdqu          %Xi, 0x40(%Htbl)        / save H^4
506         palignr         $8, %T1, %T2            / low part is H^3.lo^H^3.hi...
507         movdqu          %T2, 0x50(%Htbl)        / save Karatsuba "salt"

509         ret
510         SET_SIZE(gcm_init_clmul)

512 #endif  /* lint || __lint */
```

```
**********************************************************
   11127 Thu Apr 30 20:52:30 2015
new/usr/src/common/crypto/modes/cbc.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

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

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

boolean_t cbc_fastpath_enabled = B_TRUE;

static void
cbc_decrypt_fastpath(cbc_ctx_t *ctx, const uint8_t *data, size_t length,
    uint8_t *out, size_t block_size,
    int (*decrypt)(const void *, const uint8_t *, uint8_t *),
    int (*decrypt_ecb)(const void *, const uint8_t *, uint8_t *, uint64_t),
    void (*xor_block)(const uint8_t *, uint8_t *),
    void (*xor_block_range)(const uint8_t *, uint8_t *, uint64_t))
{
        const uint8_t *iv = (uint8_t *)ctx->cbc_iv;

        /* Use bulk decryption when available. */
        if (decrypt_ecb != NULL) {
                decrypt_ecb(ctx->cbc_keysched, data, out, length);
        } else {
                for (size_t i = 0; i < length; i += block_size)
                        decrypt(ctx->cbc_keysched, &data[i], &out[i]);
        }
```



```
        /* Use bulk XOR when available. */
        if (xor_block_range != NULL && length >= 2 * block_size) {
                xor_block(iv, out);
                xor_block_range(data, &out[block_size], length - block_size);
        } else {
                for (size_t i = 0; i < length; i += block_size) {
                        xor_block(iv, &out[i]);
                        iv = &data[i];
                }
        }
}

/*
 * Algorithm independent CBC functions.
 */
int
cbc_encrypt_contiguous_blocks(cbc_ctx_t *ctx, char *data, size_t length,
    crypto_data_t *out, size_t block_size,
    int (*encrypt)(const void *, const uint8_t *, uint8_t *),
    void (*copy_block)(const uint8_t *, uint8_t *),
    void (*xor_block)(const uint8_t *, uint8_t *),
    int (*encrypt_cbc)(const void *, const uint8_t *, uint8_t *,
    const uint8_t *, uint64_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 *lastp;
        void *iov_or_mp;
        offset_t offset;
        uint8_t *out_data_1;
        uint8_t *out_data_2;
        size_t out_data_1_len;

        /*
         * CBC encryption fastpath requirements:
         * - fastpath is enabled
         * - algorithm-specific acceleration function is available
         * - input is block-aligned
         * - output is a single contiguous region or the user requested that
         *   we overwrite their input buffer (input/output aliasing allowed)
         */
        if (cbc_fastpath_enabled && encrypt_cbc != NULL && length != 0 &&
            ctx->cbc_remainder_len == 0 && (length & (block_size - 1)) == 0 &&
            (out == NULL || CRYPTO_DATA_IS_SINGLE_BLOCK(out))) {
                if (out == NULL) {
                        encrypt_cbc(ctx->cbc_keysched, (uint8_t *)data,
                            (uint8_t *)data, (uint8_t *)ctx->cbc_iv, length);
                        ctx->cbc_lastp = (uint8_t *)&data[length - block_size];
                } else {
                        uint8_t *outp = CRYPTO_DATA_FIRST_BLOCK(out);
                        encrypt_cbc(ctx->cbc_keysched, (uint8_t *)data, outp,
                            (uint8_t *)ctx->cbc_iv, length);
                        out->cd_offset += length;
                        ctx->cbc_lastp = &outp[length - block_size];
                }
                goto out;
        }

        if (length + ctx->cbc_remainder_len < block_size) {
                /* accumulate bytes here and return */
                bcopy(datap,
                    (uint8_t *)ctx->cbc_remainder + ctx->cbc_remainder_len,
```

```
126                     length);
127                 ctx->cbc_remainder_len += length;
128                 ctx->cbc_copy_to = datap;
129                 return (CRYPTO_SUCCESS);
130         }

132         lastp = (uint8_t *)ctx->cbc_iv;
133         if (out != NULL)
134                 crypto_init_ptrs(out, &iov_or_mp, &offset);

136         do {
137                 /* Unprocessed data from last call. */
138                 if (ctx->cbc_remainder_len > 0) {
139                         need = block_size - ctx->cbc_remainder_len;

141                         if (need > remainder)
142                                 return (CRYPTO_DATA_LEN_RANGE);

144                         bcopy(datap, &((uint8_t *)ctx->cbc_remainder)
145                             [ctx->cbc_remainder_len], need);

147                         blockp = (uint8_t *)ctx->cbc_remainder;
148                 } else {
149                         blockp = datap;
150                 }

152                 if (out == NULL) {
153                         /*
154                          * XOR the previous cipher block or IV with the
155                          * current clear block.
156                          */
157                         xor_block(lastp, blockp);
158                         encrypt(ctx->cbc_keysched, blockp, blockp);

160                         ctx->cbc_lastp = blockp;
161                         lastp = blockp;

163                         if (ctx->cbc_remainder_len > 0) {
164                                 bcopy(blockp, ctx->cbc_copy_to,
165                                     ctx->cbc_remainder_len);
166                                 bcopy(blockp + ctx->cbc_remainder_len, datap,
167                                     need);
168                         }
169                 } else {
170                         /*
171                          * XOR the previous cipher block or IV with the
172                          * current clear block.
173                          */
174                         xor_block(blockp, lastp);
175                         encrypt(ctx->cbc_keysched, lastp, lastp);
176                         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
177                             &out_data_1_len, &out_data_2, block_size);

179                         /* copy block to where it belongs */
180                         if (out_data_1_len == block_size) {
181                                 copy_block(lastp, out_data_1);
182                         } else {
183                                 bcopy(lastp, out_data_1, out_data_1_len);
184                                 if (out_data_2 != NULL) {
185                                         bcopy(lastp + out_data_1_len,
186                                             out_data_2,
187                                             block_size - out_data_1_len);
188                                 }
189                         }
190                         /* update offset */
191                         out->cd_offset += block_size;
```



```
192                 }

194                 /* Update pointer to next block of data to be processed. */
195                 if (ctx->cbc_remainder_len != 0) {
196                         datap += need;
197                         ctx->cbc_remainder_len = 0;
198                 } else {
199                         datap += block_size;
200                 }

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

204                 /* Incomplete last block. */
205                 if (remainder > 0 && remainder < block_size) {
206                         bcopy(datap, ctx->cbc_remainder, remainder);
207                         ctx->cbc_remainder_len = remainder;
208                         ctx->cbc_copy_to = datap;
209                         goto out;
210                 }
211                 ctx->cbc_copy_to = NULL;

213         } while (remainder > 0);

215 out:
216         /*
217          * Save the last encrypted block in the context.
218          */
219         if (ctx->cbc_lastp != NULL) {
220                 copy_block((uint8_t *)ctx->cbc_lastp, (uint8_t *)ctx->cbc_iv);
221                 ctx->cbc_lastp = (uint8_t *)ctx->cbc_iv;
222         }

224         return (CRYPTO_SUCCESS);
225 }

227 #define OTHER(a, ctx) \
228         (((a) == (ctx)->cbc_lastblock) ? (ctx)->cbc_iv : (ctx)->cbc_lastblock)

230 /* ARGSUSED */
231 int
232 cbc_decrypt_contiguous_blocks(cbc_ctx_t *ctx, char *data, size_t length,
233     crypto_data_t *out, size_t block_size,
234     int (*decrypt)(const void *, const uint8_t *, uint8_t *),
235     void (*copy_block)(const uint8_t *, uint8_t *),
236     void (*xor_block)(const uint8_t *, uint8_t *),
237     int (*decrypt_ecb)(const void *, const uint8_t *, uint8_t *, uint64_t),
238     void (*xor_block_range)(const uint8_t *, uint8_t *, uint64_t))
172     void (*copy_block)(uint8_t *, uint8_t *),
173     void (*xor_block)(uint8_t *, uint8_t *))
239 {
240         size_t remainder = length;
241         size_t need;
242         uint8_t *datap = (uint8_t *)data;
243         uint8_t *blockp;
244         uint8_t *lastp;
245         void *iov_or_mp;
246         offset_t offset;
247         uint8_t *out_data_1;
248         uint8_t *out_data_2;
249         size_t out_data_1_len;

251         /*
252          * CBC decryption fastpath requirements:
253          * - fastpath is enabled
254          * - input is block-aligned
255          * - output is a single contiguous region and doesn't alias input
```

```
256          */
257         if (cbc_fastpath_enabled && ctx->cbc_remainder_len == 0 &&
258             length != 0 && (length & (block_size - 1)) == 0 &&
259             CRYPTO_DATA_IS_SINGLE_BLOCK(out)) {
260                 uint8_t *outp = CRYPTO_DATA_FIRST_BLOCK(out);
261
262                 cbc_decrypt_fastpath(ctx, (uint8_t *)data, length, outp,
263                     block_size, decrypt, decrypt_ecb, xor_block,
264                     xor_block_range);
265                 out->cd_offset += length;
266                 bcopy(&data[length - block_size], ctx->cbc_iv, block_size);
267                 ctx->cbc_lastp = (uint8_t *)ctx->cbc_iv;
268                 return (CRYPTO_SUCCESS);
269         }
270
271         if (length + ctx->cbc_remainder_len < block_size) {
272                 /* accumulate bytes here and return */
273                 bcopy(datap,
274                     (uint8_t *)ctx->cbc_remainder + ctx->cbc_remainder_len,
275                     length);
276                 ctx->cbc_remainder_len += length;
277                 ctx->cbc_copy_to = datap;
278                 return (CRYPTO_SUCCESS);
279         }
280
281         lastp = ctx->cbc_lastp;
282         if (out != NULL)
283                 crypto_init_ptrs(out, &iov_or_mp, &offset);
284
285         do {
286                 /* Unprocessed data from last call. */
287                 if (ctx->cbc_remainder_len > 0) {
288                         need = block_size - ctx->cbc_remainder_len;
289
290                         if (need > remainder)
291                                 return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
292
293                         bcopy(datap, &((uint8_t *)ctx->cbc_remainder)
294                             [ctx->cbc_remainder_len], need);
295
296                         blockp = (uint8_t *)ctx->cbc_remainder;
297                 } else {
298                         blockp = datap;
299                 }
300
301                 /* LINTED: pointer alignment */
302                 copy_block(blockp, (uint8_t *)OTHER((uint64_t *)lastp, ctx));
303
304                 if (out != NULL) {
305                         decrypt(ctx->cbc_keysched, blockp,
306                             (uint8_t *)ctx->cbc_remainder);
307                         blockp = (uint8_t *)ctx->cbc_remainder;
308                 } else {
309                         decrypt(ctx->cbc_keysched, blockp, blockp);
310                 }
311
312                 /*
313                  * XOR the previous cipher block or IV with the
314                  * currently decrypted block.
315                  */
316                 xor_block(lastp, blockp);
317
318                 /* LINTED: pointer alignment */
319                 lastp = (uint8_t *)OTHER((uint64_t *)lastp, ctx);
320
321                 if (out != NULL) {
```



```
322                         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
323                             &out_data_1_len, &out_data_2, block_size);
324
325                         bcopy(blockp, out_data_1, out_data_1_len);
326                         if (out_data_2 != NULL) {
327                                 bcopy(blockp + out_data_1_len, out_data_2,
328                                     block_size - out_data_1_len);
329                         }
330
331                         /* update offset */
332                         out->cd_offset += block_size;
333
334                 } else if (ctx->cbc_remainder_len > 0) {
335                         /* copy temporary block to where it belongs */
336                         bcopy(blockp, ctx->cbc_copy_to, ctx->cbc_remainder_len);
337                         bcopy(blockp + ctx->cbc_remainder_len, datap, need);
338                 }
339
340                 /* Update pointer to next block of data to be processed. */
341                 if (ctx->cbc_remainder_len != 0) {
342                         datap += need;
343                         ctx->cbc_remainder_len = 0;
344                 } else {
345                         datap += block_size;
346                 }
347
348                 remainder = (size_t)&data[length] - (size_t)datap;
349
350                 /* Incomplete last block. */
351                 if (remainder > 0 && remainder < block_size) {
352                         bcopy(datap, ctx->cbc_remainder, remainder);
353                         ctx->cbc_remainder_len = remainder;
354                         ctx->cbc_lastp = lastp;
355                         ctx->cbc_copy_to = datap;
356                         return (CRYPTO_SUCCESS);
357                 }
358                 ctx->cbc_copy_to = NULL;
359
360         } while (remainder > 0);
361
362         ctx->cbc_lastp = lastp;
363         return (CRYPTO_SUCCESS);
364 }
365
366 int
367 cbc_init_ctx(cbc_ctx_t *cbc_ctx, char *param, size_t param_len,
368     size_t block_size, void (*copy_block)(const uint8_t *, uint64_t *))
283     size_t block_size, void (*copy_block)(uint8_t *, uint64_t *))
369 {
370         /*
371          * Copy IV into context.
372          *
373          * If cm_param == NULL then the IV comes from the
374          * cd_miscdata field in the crypto_data structure.
375          */
376         if (param != NULL) {
377 #ifdef _KERNEL
378                 ASSERT(param_len == block_size);
379 #else
380                 assert(param_len == block_size);
381 #endif
382                 copy_block((uchar_t *)param, cbc_ctx->cbc_iv);
383         }
384
385         cbc_ctx->cbc_lastp = (uint8_t *)&cbc_ctx->cbc_iv[0];
386         cbc_ctx->cbc_flags |= CBC_MODE;
```

```
 387         return (CRYPTO_SUCCESS);
 388 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   24828 Thu Apr 30 20:52:31 2015
new/usr/src/common/crypto/modes/ccm.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

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

#include <sys/types.h>
#include <sys/kmem.h>
#define INLINE_CRYPTO_GET_PTRS
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>
#include <sys/byteorder.h>

#if defined(__i386) || defined(__amd64)
#define UNALIGNED_POINTERS_PERMITTED
#endif

/*
 * Encrypt multiple blocks of data in CCM mode.  Decrypt for CCM mode
 * is done in another function.
 */
int
ccm_mode_encrypt_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)(const uint8_t *, uint8_t *),
    void (*xor_block)(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 *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;
        uint8_t *mac_buf;

        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);
        }

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

        mac_buf = (uint8_t *)ctx->ccm_mac_buf;

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

                        if (need > remainder)
                                return (CRYPTO_DATA_LEN_RANGE);

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

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

                /*
                 * do CBC MAC
                 *
                 * XOR the previous cipher block current clear block.
                 * mac_buf always contain previous cipher block.
                 */
                xor_block(blockp, mac_buf);
                encrypt_block(ctx->ccm_keysched, mac_buf, mac_buf);

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

                lastp = (uint8_t *)ctx->ccm_tmp;

                /*
                 * Increment counter. Counter bits are confined
                 * to the bottom 64 bits of the counter block.
                 */
                counter = ntohll(ctx->ccm_cb[1] & ctx->ccm_counter_mask);
                counter = htonll(counter + 1);
                counter &= ctx->ccm_counter_mask;
                ctx->ccm_cb[1] =
                    (ctx->ccm_cb[1] & ~(ctx->ccm_counter_mask)) | counter;
```

```
126                 /*
127                  * XOR encrypted counter block with the current clear block.
128                  */
129                 xor_block(blockp, lastp);

131                 ctx->ccm_processed_data_len += block_size;

133                 if (out == NULL) {
134                         if (ctx->ccm_remainder_len > 0) {
135                                 bcopy(blockp, ctx->ccm_copy_to,
136                                     ctx->ccm_remainder_len);
137                                 bcopy(blockp + ctx->ccm_remainder_len, datap,
138                                     need);
139                         }
140                 } else {
141                         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
142                             &out_data_1_len, &out_data_2, block_size);

144                         /* copy block to where it belongs */
145                         if (out_data_1_len == block_size) {
146                                 copy_block(lastp, out_data_1);
147                         } else {
148                                 bcopy(lastp, out_data_1, out_data_1_len);
149                                 if (out_data_2 != NULL) {
150                                         bcopy(lastp + out_data_1_len,
151                                             out_data_2,
152                                             block_size - out_data_1_len);
153                                 }
154                         }
155                         /* update offset */
156                         out->cd_offset += block_size;
157                 }

159                 /* Update pointer to next block of data to be processed. */
160                 if (ctx->ccm_remainder_len != 0) {
161                         datap += need;
162                         ctx->ccm_remainder_len = 0;
163                 } else {
164                         datap += block_size;
165                 }

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

169                 /* Incomplete last block. */
170                 if (remainder > 0 && remainder < block_size) {
171                         bcopy(datap, ctx->ccm_remainder, remainder);
172                         ctx->ccm_remainder_len = remainder;
173                         ctx->ccm_copy_to = datap;
174                         goto out;
175                 }
176                 ctx->ccm_copy_to = NULL;

178         } while (remainder > 0);

180 out:
181         return (CRYPTO_SUCCESS);
182 }
```

*unchanged_portion_omitted_*

```
207 /* ARGSUSED */
208 int
209 ccm_encrypt_final(ccm_ctx_t *ctx, crypto_data_t *out, size_t block_size,
210     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
211     void (*xor_block)(const uint8_t *, uint8_t *))
209     void (*xor_block)(uint8_t *, uint8_t *))
212 {
```



```
213         uint8_t *lastp, *mac_buf, *ccm_mac_p, *macp;
214         void *iov_or_mp;
215         offset_t offset;
216         uint8_t *out_data_1;
217         uint8_t *out_data_2;
218         size_t out_data_1_len;
219         int i;

221         if (out->cd_length < (ctx->ccm_remainder_len + ctx->ccm_mac_len)) {
222                 return (CRYPTO_DATA_LEN_RANGE);
223         }

225         /*
226          * When we get here, the number of bytes of payload processed
227          * plus whatever data remains, if any,
228          * should be the same as the number of bytes that's being
229          * passed in the argument during init time.
230          */
231         if ((ctx->ccm_processed_data_len + ctx->ccm_remainder_len)
232             != (ctx->ccm_data_len)) {
233                 return (CRYPTO_DATA_LEN_RANGE);
234         }

236         mac_buf = (uint8_t *)ctx->ccm_mac_buf;

238         if (ctx->ccm_remainder_len > 0) {

240                 /* ccm_mac_input_buf is not used for encryption */
241                 macp = (uint8_t *)ctx->ccm_mac_input_buf;
242                 bzero(macp, block_size);

244                 /* copy remainder to temporary buffer */
245                 bcopy(ctx->ccm_remainder, macp, ctx->ccm_remainder_len);

247                 /* calculate the CBC MAC */
248                 xor_block(macp, mac_buf);
249                 encrypt_block(ctx->ccm_keysched, mac_buf, mac_buf);

251                 /* calculate the counter mode */
252                 lastp = (uint8_t *)ctx->ccm_tmp;
253                 encrypt_block(ctx->ccm_keysched, (uint8_t *)ctx->ccm_cb, lastp);

255                 /* XOR with counter block */
256                 for (i = 0; i < ctx->ccm_remainder_len; i++) {
257                         macp[i] ^= lastp[i];
258                 }
259                 ctx->ccm_processed_data_len += ctx->ccm_remainder_len;
260         }

262         /* Calculate the CCM MAC */
263         ccm_mac_p = (uint8_t *)ctx->ccm_tmp;
264         calculate_ccm_mac(ctx, ccm_mac_p, encrypt_block);

266         crypto_init_ptrs(out, &iov_or_mp, &offset);
267         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
268             &out_data_1_len, &out_data_2,
269             ctx->ccm_remainder_len + ctx->ccm_mac_len);

271         if (ctx->ccm_remainder_len > 0) {

273                 /* copy temporary block to where it belongs */
274                 if (out_data_2 == NULL) {
275                         /* everything will fit in out_data_1 */
276                         bcopy(macp, out_data_1, ctx->ccm_remainder_len);
277                         bcopy(ccm_mac_p, out_data_1 + ctx->ccm_remainder_len,
278                             ctx->ccm_mac_len);
```

```
279                 } else {
281                         if (out_data_1_len < ctx->ccm_remainder_len) {
283                                 size_t data_2_len_used;
285                                 bcopy(macp, out_data_1, out_data_1_len);
287                                 data_2_len_used = ctx->ccm_remainder_len
288                                     - out_data_1_len;
290                                 bcopy((uint8_t *)macp + out_data_1_len,
291                                     out_data_2, data_2_len_used);
292                                 bcopy(ccm_mac_p, out_data_2 + data_2_len_used,
293                                     ctx->ccm_mac_len);
294                         } else {
295                                 bcopy(macp, out_data_1, out_data_1_len);
296                                 if (out_data_1_len == ctx->ccm_remainder_len) {
297                                         /* mac will be in out_data_2 */
298                                         bcopy(ccm_mac_p, out_data_2,
299                                             ctx->ccm_mac_len);
300                                 } else {
301                                         size_t len_not_used = out_data_1_len -
302                                             ctx->ccm_remainder_len;
303                                         /*
304                                          * part of mac in will be in
305                                          * out_data_1, part of the mac will be
306                                          * in out_data_2
307                                          */
308                                         bcopy(ccm_mac_p,
309                                             out_data_1 + ctx->ccm_remainder_len,
310                                             len_not_used);
311                                         bcopy(ccm_mac_p + len_not_used,
312                                             out_data_2,
313                                             ctx->ccm_mac_len - len_not_used);
315                                 }
316                         }
317                 }
318         } else {
319                 /* copy block to where it belongs */
320                 bcopy(ccm_mac_p, out_data_1, out_data_1_len);
321                 if (out_data_2 != NULL) {
322                         bcopy(ccm_mac_p + out_data_1_len, out_data_2,
323                             block_size - out_data_1_len);
324                 }
325         }
326         out->cd_offset += ctx->ccm_remainder_len + ctx->ccm_mac_len;
327         ctx->ccm_remainder_len = 0;
328         return (CRYPTO_SUCCESS);
329 }
```

*unchanged_portion_omitted_*

```
354 /*
355  * This will decrypt the cipher text.  However, the plaintext won't be
356  * returned to the caller.  It will be returned when decrypt_final() is
357  * called if the MAC matches
358  */
359 /* ARGSUSED */
360 int
361 ccm_mode_decrypt_contiguous_blocks(ccm_ctx_t *ctx, char *data, size_t length,
362     crypto_data_t *out, size_t block_size,
363     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
364     void (*copy_block)(const uint8_t *, uint8_t *),
365     void (*xor_block)(const uint8_t *, uint8_t *))
362     void (*copy_block)(uint8_t *, uint8_t *),
```



```
363     void (*xor_block)(uint8_t *, uint8_t *))
366 {
367         size_t remainder = length;
368         size_t need;
369         uint8_t *datap = (uint8_t *)data;
370         uint8_t *blockp;
371         uint8_t *cbp;
372         uint64_t counter;
373         size_t pt_len, total_decrypted_len, mac_len, pm_len, pd_len;
374         uint8_t *resultp;
377         pm_len = ctx->ccm_processed_mac_len;
379         if (pm_len > 0) {
380                 uint8_t *tmp;
381                 /*
382                  * all ciphertext has been processed, just waiting for
383                  * part of the value of the mac
384                  */
385                 if ((pm_len + length) > ctx->ccm_mac_len) {
386                         return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
387                 }
388                 tmp = (uint8_t *)ctx->ccm_mac_input_buf;
390                 bcopy(datap, tmp + pm_len, length);
392                 ctx->ccm_processed_mac_len += length;
393                 return (CRYPTO_SUCCESS);
394         }
396         /*
397          * If we decrypt the given data, what total amount of data would
398          * have been decrypted?
399          */
400         pd_len = ctx->ccm_processed_data_len;
401         total_decrypted_len = pd_len + length + ctx->ccm_remainder_len;
403         if (total_decrypted_len >
404             (ctx->ccm_data_len + ctx->ccm_mac_len)) {
405                 return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
406         }
408         pt_len = ctx->ccm_data_len;
410         if (total_decrypted_len > pt_len) {
411                 /*
412                  * part of the input will be the MAC, need to isolate that
413                  * to be dealt with later.  The left-over data in
414                  * ccm_remainder_len from last time will not be part of the
415                  * MAC.  Otherwise, it would have already been taken out
416                  * when this call is made last time.
417                  */
418                 size_t pt_part = pt_len - pd_len - ctx->ccm_remainder_len;
420                 mac_len = length - pt_part;
422                 ctx->ccm_processed_mac_len = mac_len;
423                 bcopy(data + pt_part, ctx->ccm_mac_input_buf, mac_len);
425                 if (pt_part + ctx->ccm_remainder_len < block_size) {
426                         /*
427                          * since this is last of the ciphertext, will
428                          * just decrypt with it here
429                          */
430                         bcopy(datap, &((uint8_t *)ctx->ccm_remainder)
```

```
431                             [ctx->ccm_remainder_len], pt_part);
432                     ctx->ccm_remainder_len += pt_part;
433                     ccm_decrypt_incomplete_block(ctx, encrypt_block);
434                     ctx->ccm_processed_data_len += ctx->ccm_remainder_len;
435                     ctx->ccm_remainder_len = 0;
436                     return (CRYPTO_SUCCESS);
437             } else {
438                     /* let rest of the code handle this */
439                     length = pt_part;
440             }
441     } else if (length + ctx->ccm_remainder_len < block_size) {
442                     /* accumulate bytes here and return */
443             bcopy(datap,
444                 (uint8_t *)ctx->ccm_remainder + ctx->ccm_remainder_len,
445                 length);
446             ctx->ccm_remainder_len += length;
447             ctx->ccm_copy_to = datap;
448             return (CRYPTO_SUCCESS);
449     }

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

456                     if (need > remainder)
457                             return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);

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

462                     blockp = (uint8_t *)ctx->ccm_remainder;
463             } else {
464                     blockp = datap;
465             }

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

471             /*
472              * Increment counter.
473              * Counter bits are confined to the bottom 64 bits
474              */
475             counter = ntohll(ctx->ccm_cb[1] & ctx->ccm_counter_mask);
476             counter = htonll(counter + 1);
477             counter &= ctx->ccm_counter_mask;
478             ctx->ccm_cb[1] =
479                 (ctx->ccm_cb[1] & ~(ctx->ccm_counter_mask)) | counter;

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

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

489             ctx->ccm_processed_data_len += block_size;

491             ctx->ccm_lastp = blockp;

493             /* Update pointer to next block of data to be processed. */
494             if (ctx->ccm_remainder_len != 0) {
495                     datap += need;
496                     ctx->ccm_remainder_len = 0;
```



```
497             } else {
498                     datap += block_size;
499             }

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

503             /* Incomplete last block */
504             if (remainder > 0 && remainder < block_size) {
505                     bcopy(datap, ctx->ccm_remainder, remainder);
506                     ctx->ccm_remainder_len = remainder;
507                     ctx->ccm_copy_to = datap;
508                     if (ctx->ccm_processed_mac_len > 0) {
509                             /*
510                              * not expecting anymore ciphertext, just
511                              * compute plaintext for the remaining input
512                              */
513                             ccm_decrypt_incomplete_block(ctx,
514                                 encrypt_block);
515                             ctx->ccm_processed_data_len += remainder;
516                             ctx->ccm_remainder_len = 0;
517                     }
518                     goto out;
519             }
520             ctx->ccm_copy_to = NULL;

522     } while (remainder > 0);

524 out:
525     return (CRYPTO_SUCCESS);
526 }

528 int
529 ccm_decrypt_final(ccm_ctx_t *ctx, crypto_data_t *out, size_t block_size,
530     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
531     void (*copy_block)(const uint8_t *, uint8_t *),
532     void (*xor_block)(const uint8_t *, uint8_t *))
529     void (*copy_block)(uint8_t *, uint8_t *),
530     void (*xor_block)(uint8_t *, uint8_t *))
533 {
534     size_t mac_remain, pt_len;
535     uint8_t *pt, *mac_buf, *macp, *ccm_mac_p;
536     int rv;

538     pt_len = ctx->ccm_data_len;

540     /* Make sure output buffer can fit all of the plaintext */
541     if (out->cd_length < pt_len) {
542             return (CRYPTO_DATA_LEN_RANGE);
543     }

545     pt = ctx->ccm_pt_buf;
546     mac_remain = ctx->ccm_processed_data_len;
547     mac_buf = (uint8_t *)ctx->ccm_mac_buf;

549     macp = (uint8_t *)ctx->ccm_tmp;

551     while (mac_remain > 0) {

553             if (mac_remain < block_size) {
554                     bzero(macp, block_size);
555                     bcopy(pt, macp, mac_remain);
556                     mac_remain = 0;
557             } else {
558                     copy_block(pt, macp);
559                     mac_remain -= block_size;
560                     pt += block_size;
```

```
561                 }
563                 /* calculate the CBC MAC */
564                 xor_block(macp, mac_buf);
565                 encrypt_block(ctx->ccm_keysched, mac_buf, mac_buf);
566         }
568         /* Calculate the CCM MAC */
569         ccm_mac_p = (uint8_t *)ctx->ccm_tmp;
570         calculate_ccm_mac((ccm_ctx_t *)ctx, ccm_mac_p, encrypt_block);
572         /* compare the input CCM MAC value with what we calculated */
573         if (bcmp(ctx->ccm_mac_input_buf, ccm_mac_p, ctx->ccm_mac_len)) {
574                 /* They don't match */
575                 return (CRYPTO_INVALID_MAC);
576         } else {
577                 rv = crypto_put_output_data(ctx->ccm_pt_buf, out, pt_len);
578                 if (rv != CRYPTO_SUCCESS)
579                         return (rv);
580                 out->cd_offset += pt_len;
581         }
582         return (CRYPTO_SUCCESS);
583 }
```

unchanged_portion_omitted_

```
767 /*
768  * The following function should be call at encrypt or decrypt init time
769  * for AES CCM mode.
770  */
771 int
772 ccm_init(ccm_ctx_t *ctx, unsigned char *nonce, size_t nonce_len,
773     unsigned char *auth_data, size_t auth_data_len, size_t block_size,
774     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
775     void (*xor_block)(const uint8_t *, uint8_t *))
773     void (*xor_block)(uint8_t *, uint8_t *))
776 {
777         uint8_t *mac_buf, *datap, *ivp, *authp;
778         size_t remainder, processed;
779         uint8_t encoded_a[10]; /* max encoded auth data length is 10 octets */
780         size_t encoded_a_len = 0;
782         mac_buf = (uint8_t *)&(ctx->ccm_mac_buf);
784         /*
785          * Format the 1st block for CBC-MAC and construct the
786          * 1st counter block.
787          *
788          * aes_ctx->ccm_iv is used for storing the counter block
789          * mac_buf will store b0 at this time.
790          */
791         ccm_format_initial_blocks(nonce, nonce_len,
792             auth_data_len, mac_buf, ctx);
794         /* The IV for CBC MAC for AES CCM mode is always zero */
795         ivp = (uint8_t *)ctx->ccm_tmp;
796         bzero(ivp, block_size);
798         xor_block(ivp, mac_buf);
800         /* encrypt the nonce */
801         encrypt_block(ctx->ccm_keysched, mac_buf, mac_buf);
803         /* take care of the associated data, if any */
804         if (auth_data_len == 0) {
805                 return (CRYPTO_SUCCESS);
806         }
```



```
808         encode_adata_len(auth_data_len, encoded_a, &encoded_a_len);
810         remainder = auth_data_len;
812         /* 1st block: it contains encoded associated data, and some data */
813         authp = (uint8_t *)ctx->ccm_tmp;
814         bzero(authp, block_size);
815         bcopy(encoded_a, authp, encoded_a_len);
816         processed = block_size - encoded_a_len;
817         if (processed > auth_data_len) {
818                 /* in case auth_data is very small */
819                 processed = auth_data_len;
820         }
821         bcopy(auth_data, authp+encoded_a_len, processed);
822         /* xor with previous buffer */
823         xor_block(authp, mac_buf);
824         encrypt_block(ctx->ccm_keysched, mac_buf, mac_buf);
825         remainder -= processed;
826         if (remainder == 0) {
827                 /* a small amount of associated data, it's all done now */
828                 return (CRYPTO_SUCCESS);
829         }
831         do {
832                 if (remainder < block_size) {
833                         /*
834                          * There's not a block full of data, pad rest of
835                          * buffer with zero
836                          */
837                         bzero(authp, block_size);
838                         bcopy(&(auth_data[processed]), authp, remainder);
839                         datap = (uint8_t *)authp;
840                         remainder = 0;
841                 } else {
842                         datap = (uint8_t *)(&(auth_data[processed]));
843                         processed += block_size;
844                         remainder -= block_size;
845                 }
847                 xor_block(datap, mac_buf);
848                 encrypt_block(ctx->ccm_keysched, mac_buf, mac_buf);
850         } while (remainder > 0);
852         return (CRYPTO_SUCCESS);
853 }
855 int
856 ccm_init_ctx(ccm_ctx_t *ccm_ctx, char *param, int kmflag,
857     boolean_t is_encrypt_init, size_t block_size,
858     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
859     void (*xor_block)(const uint8_t *, uint8_t *))
857     void (*xor_block)(uint8_t *, uint8_t *))
860 {
861         int rv;
862         CK_AES_CCM_PARAMS *ccm_param;
864         if (param != NULL) {
865                 ccm_param = (CK_AES_CCM_PARAMS *)(void *)param;
867                 if ((rv = ccm_validate_args(ccm_param,
868                     is_encrypt_init)) != 0) {
869                         return (rv);
870                 }
```

```
872                 ccm_ctx->ccm_mac_len = ccm_param->ulMACSize;
873                 if (is_encrypt_init) {
874                         ccm_ctx->ccm_data_len = ccm_param->ulDataSize;
875                 } else {
876                         ccm_ctx->ccm_data_len =
877                             ccm_param->ulDataSize - ccm_ctx->ccm_mac_len;
878                         ccm_ctx->ccm_processed_mac_len = 0;
879                 }
880                 ccm_ctx->ccm_processed_data_len = 0;

882                 ccm_ctx->ccm_flags |= CCM_MODE;
883         } else {
884                 rv = CRYPTO_MECHANISM_PARAM_INVALID;
885                 goto out;
886         }

888         if (ccm_init(ccm_ctx, ccm_param->nonce, ccm_param->ulNonceSize,
889             ccm_param->authData, ccm_param->ulAuthDataSize, block_size,
890             encrypt_block, xor_block) != 0) {
891                 rv = CRYPTO_MECHANISM_PARAM_INVALID;
892                 goto out;
893         }
894         if (!is_encrypt_init) {
895                 /* allocate buffer for storing decrypted plaintext */
896 #ifdef _KERNEL
897                 ccm_ctx->ccm_pt_buf = kmem_alloc(ccm_ctx->ccm_data_len,
898                     kmflag);
899 #else
900                 ccm_ctx->ccm_pt_buf = malloc(ccm_ctx->ccm_data_len);
901 #endif
902                 if (ccm_ctx->ccm_pt_buf == NULL) {
903                         rv = CRYPTO_HOST_MEMORY;
904                 }
905         }
906 out:
907         return (rv);
908 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7477 Thu Apr 30 20:52:31 2015
new/usr/src/common/crypto/modes/ctr.c
4896 Performance improvements for KCF AES modes
**********************************************************
/*
 * 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.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

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

#include <sys/types.h>
#define INLINE_CRYPTO_GET_PTRS
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>
#include <sys/byteorder.h>
#include <sys/cmn_err.h>

boolean_t ctr_fastpath_enabled = B_TRUE;

/*
 * 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)(const uint8_t *, uint8_t *),
    int (*cipher_ctr)(const void *ks, const uint8_t *pt, uint8_t *ct,
    uint64_t len, uint64_t counter[2]))
    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 lower_counter, upper_counter;

        /*
         * CTR encryption/decryption fastpath requirements:
         * - fastpath is enabled
         * - algorithm-specific acceleration function is available
         * - input is block-aligned
         * - the counter value won't overflow the lower counter mask
         * - output is a single contiguous region and doesn't alias input
         */
        if (ctr_fastpath_enabled && cipher_ctr != NULL &&
            ctx->ctr_remainder_len == 0 && (length & (block_size - 1)) == 0 &&
            ntohll(ctx->ctr_cb[1]) <= ctx->ctr_lower_mask -
            length / block_size && CRYPTO_DATA_IS_SINGLE_BLOCK(out)) {
                cipher_ctr(ctx->ctr_keysched, (uint8_t *)data,
                    CRYPTO_DATA_FIRST_BLOCK(out), length, ctx->ctr_cb);
                out->cd_offset += length;
                return (CRYPTO_SUCCESS);
        }

        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.
                 */
                lower_counter = ntohll(ctx->ctr_cb[1] & ctx->ctr_lower_mask);
```

```
127                 lower_counter = htonll(lower_counter + 1);
128                 lower_counter &= ctx->ctr_lower_mask;
129                 ctx->ctr_cb[1] = (ctx->ctr_cb[1] & ~(ctx->ctr_lower_mask)) |
130                     lower_counter;

132                 /* wrap around */
133                 if (lower_counter == 0) {
134                         upper_counter =
135                             ntohll(ctx->ctr_cb[0] & ctx->ctr_upper_mask);
136                         upper_counter = htonll(upper_counter + 1);
137                         upper_counter &= ctx->ctr_upper_mask;
138                         ctx->ctr_cb[0] =
139                             (ctx->ctr_cb[0] & ~(ctx->ctr_upper_mask)) |
140                             upper_counter;
141                 }

143                 /*
144                  * XOR encrypted counter block with the current clear block.
145                  */
146                 xor_block(blockp, lastp);

148                 if (out == NULL) {
149                         if (ctx->ctr_remainder_len > 0) {
150                                 bcopy(lastp, ctx->ctr_copy_to,
151                                     ctx->ctr_remainder_len);
152                                 bcopy(lastp + ctx->ctr_remainder_len, datap,
153                                     need);
154                         }
155                 } else {
156                         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
157                             &out_data_1_len, &out_data_2, block_size);

159                         /* copy block to where it belongs */
160                         bcopy(lastp, out_data_1, out_data_1_len);
161                         if (out_data_2 != NULL) {
162                                 bcopy(lastp + out_data_1_len, out_data_2,
163                                     block_size - out_data_1_len);
164                         }
165                         /* update offset */
166                         out->cd_offset += block_size;
167                 }

169                 /* Update pointer to next block of data to be processed. */
170                 if (ctx->ctr_remainder_len != 0) {
171                         datap += need;
172                         ctx->ctr_remainder_len = 0;
173                 } else {
174                         datap += block_size;
175                 }

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

179                 /* Incomplete last block. */
180                 if (remainder > 0 && remainder < block_size) {
181                         bcopy(datap, ctx->ctr_remainder, remainder);
182                         ctx->ctr_remainder_len = remainder;
183                         ctx->ctr_copy_to = datap;
184                         goto out;
185                 }
186                 ctx->ctr_copy_to = NULL;

188         } while (remainder > 0);

190 out:
191         return (CRYPTO_SUCCESS);
192 }
```

unchanged_portion_omitted_


```
233 int
234 ctr_init_ctx(ctr_ctx_t *ctr_ctx, ulong_t count, uint8_t *cb,
235     void (*copy_block)(const uint8_t *, uint8_t *))
208 void (*copy_block)(uint8_t *, uint8_t *))
236 {
237         uint64_t upper_mask = 0;
238         uint64_t lower_mask = 0;

240         if (count == 0 || count > 128) {
241                 return (CRYPTO_MECHANISM_PARAM_INVALID);
242         }
243         /* upper 64 bits of the mask */
244         if (count >= 64) {
245                 count -= 64;
246                 upper_mask = (count == 64) ? UINT64_MAX : (1ULL << count) - 1;
247                 lower_mask = UINT64_MAX;
248         } else {
249                 /* now the lower 63 bits */
250                 lower_mask = (1ULL << count) - 1;
251         }
252         ctr_ctx->ctr_lower_mask = htonll(lower_mask);
253         ctr_ctx->ctr_upper_mask = htonll(upper_mask);

255         copy_block(cb, (uchar_t *)ctr_ctx->ctr_cb);
256         ctr_ctx->ctr_lastp = (uint8_t *)&ctr_ctx->ctr_cb[0];
257         ctr_ctx->ctr_flags |= CTR_MODE;
258         return (CRYPTO_SUCCESS);
259 }
```

unchanged_portion_omitted_

```
**********************************************************
    5185 Thu Apr 30 20:52:31 2015
new/usr/src/common/crypto/modes/ecb.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

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

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

boolean_t ecb_fastpath_enabled = B_TRUE;

/*
 * Algorithm independent ECB functions.
 * 'cipher' is a single-block version of the cipher function to be performed
 * on each input block. 'cipher_ecb' is an optional parameter, which if
 * passed and the input/output conditions allow it, will be invoked for the
 * entire input buffer once to accelerate the operation.
 */
int
ecb_cipher_contiguous_blocks(ecb_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),
    int (*cipher_ecb)(const void *ks, const uint8_t *pt, uint8_t *ct,
    uint64_t len))
    int (*cipher)(const void *ks, const uint8_t *pt, uint8_t *ct))
{
        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;

        /*
         * ECB encryption/decryption fastpath requirements:
         * - fastpath is enabled
         * - caller passed an accelerated ECB version of the cipher function
         * - input is block-aligned
         * - output is a single contiguous region or the user requested that
         *   we overwrite their input buffer (input/output aliasing allowed)
         */
        if (ecb_fastpath_enabled && cipher_ecb != NULL &&
            ctx->ecb_remainder_len == 0 && length % block_size == 0 &&
            (out == NULL || CRYPTO_DATA_IS_SINGLE_BLOCK(out))) {
                if (out == NULL) {
                        cipher_ecb(ctx->ecb_keysched, (uint8_t *)data,
                            (uint8_t *)data, length);
                } else {
                        cipher_ecb(ctx->ecb_keysched, (uint8_t *)data,
                            CRYPTO_DATA_FIRST_BLOCK(out), length);
                        out->cd_offset += length;
                }
                return (CRYPTO_SUCCESS);
        }

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

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

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

                        if (need > remainder)
                                return (CRYPTO_DATA_LEN_RANGE);

                        bcopy(datap, &((uint8_t *)ctx->ecb_remainder)
                            [ctx->ecb_remainder_len], need);

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

                if (out == NULL) {
                        cipher(ctx->ecb_keysched, blockp, blockp);

                        ctx->ecb_lastp = blockp;
                        lastp = blockp;
```

```
127                         if (ctx->ecb_remainder_len > 0) {
128                                 bcopy(blockp, ctx->ecb_copy_to,
129                                     ctx->ecb_remainder_len);
130                                 bcopy(blockp + ctx->ecb_remainder_len, datap,
131                                     need);
132                         }
133                 } else {
134                         cipher(ctx->ecb_keysched, blockp, lastp);
135                         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
136                             &out_data_1_len, &out_data_2, block_size);

138                         /* copy block to where it belongs */
139                         bcopy(lastp, out_data_1, out_data_1_len);
140                         if (out_data_2 != NULL) {
141                                 bcopy(lastp + out_data_1_len, out_data_2,
142                                     block_size - out_data_1_len);
143                         }
144                         /* update offset */
145                         out->cd_offset += block_size;
146                 }

148                 /* Update pointer to next block of data to be processed. */
149                 if (ctx->ecb_remainder_len != 0) {
150                         datap += need;
151                         ctx->ecb_remainder_len = 0;
152                 } else {
153                         datap += block_size;
154                 }

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

158                 /* Incomplete last block. */
159                 if (remainder > 0 && remainder < block_size) {
160                         bcopy(datap, ctx->ecb_remainder, remainder);
161                         ctx->ecb_remainder_len = remainder;
162                         ctx->ecb_copy_to = datap;
163                         goto out;
164                 }
165                 ctx->ecb_copy_to = NULL;

167         } while (remainder > 0);

169 out:
170         return (CRYPTO_SUCCESS);
171 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   27524 Thu Apr 30 20:52:31 2015
new/usr/src/common/crypto/modes/gcm.c
4896 Performance improvements for KCF AES modes
**********************************************************
/*
 * 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 (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */


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

#include <sys/cmn_err.h>

#include <sys/types.h>
#include <sys/kmem.h>
#define INLINE_CRYPTO_GET_PTRS
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>
#include <sys/byteorder.h>

#define COUNTER_MASK    0x00000000ffffffffULL

#ifdef  _KERNEL
#include <sys/sdt.h>            /* SET_ERROR */
#endif  /* _KERNEL */

#ifdef __amd64

#ifdef _KERNEL
#include <sys/cpuvar.h>         /* cpu_t, CPU */
#include <sys/x86_archext.h>    /* x86_featureset, X86FSET_*, CPUID_* */
#include <sys/disp.h>           /* kpreempt_disable(), kpreempt_enable */
/* Workaround for no XMM kernel thread save/restore */
extern void gcm_accel_save(void *savestate);
extern void gcm_accel_restore(void *savestate);
#define KPREEMPT_DISABLE        kpreempt_disable()
#define KPREEMPT_ENABLE         kpreempt_enable()
```



```
#if     defined(lint) || defined(__lint)
#define GCM_ACCEL_SAVESTATE(name)       uint8_t name[16 * 16 + 8]
#else
#define GCM_ACCEL_SAVESTATE(name) \
        /* stack space for xmm0--xmm15 and cr0 (16 x 128 bits + 64 bits) */ \
        uint8_t name[16 * 16 + 8] __attribute__((aligned(16)))
#endif

/*
 * Disables kernel thread preemption and conditionally gcm_accel_save() iff
 * Intel PCLMULQDQ support is present. Must be balanced by GCM_ACCEL_EXIT.
 * This must be present in all externally callable GCM functions which
 * invoke GHASH operations using FPU-accelerated implementations, or call
 * static functions which do (such as gcm_encrypt_fastpath128()).
 */
#define GCM_ACCEL_ENTER \
        GCM_ACCEL_SAVESTATE(savestate); \
        do { \
                if (intel_pclmulqdq_instruction_present()) { \
                        kpreempt_disable(); \
                        gcm_accel_save(savestate); \
                } \
                _NOTE(CONSTCOND) \
        } while (0)
#define GCM_ACCEL_EXIT \
        do { \
                if (intel_pclmulqdq_instruction_present()) { \
                        gcm_accel_restore(savestate); \
                        kpreempt_enable(); \
                } \
                _NOTE(CONSTCOND) \
        } while (0)

#else   /* _KERNEL */
#include <sys/auxv.h>           /* getisax() */
#include <sys/auxv_386.h>       /* AV_386_PCLMULQDQ bit */
#define SET_ERROR(x)    (x)
#define KPREEMPT_DISABLE
#define KPREEMPT_ENABLE
#endif  /* _KERNEL */

extern void gcm_mul_pclmulqdq(uint64_t *x_in, uint64_t *y, uint64_t *res);
extern void gcm_init_clmul(const uint64_t hash_init[2], uint8_t Htable[256]);
extern void gcm_ghash_clmul(uint64_t ghash[2], const uint8_t Htable[256],
    const uint8_t *inp, size_t length);
static inline int intel_pclmulqdq_instruction_present(void);
#else   /* !__amd64 */
#define GCM_ACCEL_ENTER
#define GCM_ACCEL_EXIT
#endif  /* !__amd64 */
static int intel_pclmulqdq_instruction_present(void);
#endif  /* __amd64 */

struct aes_block {
        uint64_t a;
        uint64_t b;
};


/*
 * gcm_mul()
 * Perform a carry-less multiplication (that is, use XOR instead of the
 * multiply operator) on *x_in and *y and place the result in *res.
 *
 * Byte swap the input (*x_in and *y) and the output (*res).
 *
```

```
 * Note: x_in, y, and res all point to 16-byte numbers (an array of two
 * 64-bit integers).
 */
static inline void
void
gcm_mul(uint64_t *x_in, uint64_t *y, uint64_t *res)
{
#ifdef __amd64
        if (intel_pclmulqdq_instruction_present()) {
                /*
                 * FPU context will have been saved and kernel thread
                 * preemption disabled already.
                 */
                KPREEMPT_DISABLE;
                gcm_mul_pclmulqdq(x_in, y, res);
                KPREEMPT_ENABLE;
        } else
#endif  /* __amd64 */
        {
                static const uint64_t R = 0xe100000000000000ULL;
                struct aes_block z = {0, 0};
                struct aes_block v;
                uint64_t x;
                int i, j;

                v.a = ntohll(y[0]);
                v.b = ntohll(y[1]);

                for (j = 0; j < 2; j++) {
                        x = ntohll(x_in[j]);
                        for (i = 0; i < 64; i++, x <<= 1) {
                                if (x & 0x8000000000000000ULL) {
                                        z.a ^= v.a;
                                        z.b ^= v.b;
                                }
                                if (v.b & 1ULL) {
                                        v.b = (v.a << 63)|(v.b >> 1);
                                        v.a = (v.a >> 1) ^ R;
                                } else {
                                        v.b = (v.a << 63)|(v.b >> 1);
                                        v.a = v.a >> 1;
                                }
                        }
                }
                res[0] = htonll(z.a);
                res[1] = htonll(z.b);
        }
}


#define GHASH(c, d, t) \
        do { \
                xor_block((uint8_t *)(d), (uint8_t *)(c)->gcm_ghash); \
                gcm_mul((uint64_t *)(void *)(c)->gcm_ghash, (c)->gcm_H, \
                    (uint64_t *)(void *)(t)); \
                _NOTE(CONSTCOND) \
        } while (0)
        (uint64_t *)(void *)(t));

boolean_t gcm_fastpath_enabled = B_TRUE;

static void
gcm_fastpath128(gcm_ctx_t *ctx, const uint8_t *data, size_t length,
    uint8_t *out, boolean_t encrypt,
    int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
    void (*xor_block)(const uint8_t *, uint8_t *),
```



```
    int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
    uint64_t *))
{
        /* When decrypting, 'data' holds the ciphertext we need to GHASH. */
        if (!encrypt) {
#ifdef  __amd64
                if (intel_pclmulqdq_instruction_present())
                        gcm_ghash_clmul(ctx->gcm_ghash, ctx->gcm_H_table,
                            data, length);
                else
#endif  /* __amd64 */
                        for (size_t i = 0; i < length; i += 16)
                                GHASH(ctx, &data[i], ctx->gcm_ghash);
        }

        if (cipher_ctr != NULL) {
                /*
                 * GCM is almost but not quite like CTR. GCM increments the
                 * counter value *before* processing the first input block,
                 * whereas CTR does so afterwards. So we need to increment
                 * the counter before calling CTR and decrement it afterwards.
/*
 * Encrypt multiple blocks of data in GCM mode.  Decrypt for GCM mode
 * is done in another function.
                 */
                uint64_t counter = ntohll(ctx->gcm_cb[1]);

                ctx->gcm_cb[1] = htonll((counter & ~COUNTER_MASK) |
                    ((counter & COUNTER_MASK) + 1));
                cipher_ctr(ctx->gcm_keysched, data, out, length, ctx->gcm_cb);
                counter = ntohll(ctx->gcm_cb[1]);
                ctx->gcm_cb[1] = htonll((counter & ~COUNTER_MASK) |
                    ((counter & COUNTER_MASK) - 1));
        } else {
                uint64_t counter = ntohll(ctx->gcm_cb[1]);

                for (size_t i = 0; i < length; i += 16) {
                        /*LINTED(E_BAD_PTR_CAST_ALIGN)*/
                        *(uint64_t *)&out[i] = ctx->gcm_cb[0];
                        /*LINTED(E_BAD_PTR_CAST_ALIGN)*/
                        *(uint64_t *)&out[i + 8] = htonll(counter++);
                        encrypt_block(ctx->gcm_keysched, &out[i], &out[i]);
                        xor_block(&data[i], &out[i]);
                }

                ctx->gcm_cb[1] = htonll(counter);
        }

        /* When encrypting, 'out' holds the ciphertext we need to GHASH. */
        if (encrypt) {
#ifdef  __amd64
                if (intel_pclmulqdq_instruction_present())
                        gcm_ghash_clmul(ctx->gcm_ghash, ctx->gcm_H_table,
                            out, length);
                else
#endif  /* __amd64 */
                        for (size_t i = 0; i < length; i += 16)
                                GHASH(ctx, &out[i], ctx->gcm_ghash);

                /* If no more data comes in, the last block is the auth tag. */
                bcopy(&out[length - 16], ctx->gcm_tmp, 16);
        }

        ctx->gcm_processed_data_len += length;
}
```

```
245 static int
246 gcm_process_contiguous_blocks(gcm_ctx_t *ctx, char *data, size_t length,
247     crypto_data_t *out, size_t block_size, boolean_t encrypt,
130 int
131 gcm_mode_encrypt_contiguous_blocks(gcm_ctx_t *ctx, char *data, size_t length,
132     crypto_data_t *out, size_t block_size,
248     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
249     void (*copy_block)(const uint8_t *, uint8_t *),
250     void (*xor_block)(const uint8_t *, uint8_t *),
251     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
252     uint64_t *))
134     void (*copy_block)(uint8_t *, uint8_t *),
135     void (*xor_block)(uint8_t *, uint8_t *))
253 {
254         size_t remainder = length;
255         size_t need;
256         uint8_t *datap = (uint8_t *)data;
257         uint8_t *blockp;
258         uint8_t *lastp;
259         void *iov_or_mp;
260         offset_t offset;
261         uint8_t *out_data_1;
262         uint8_t *out_data_2;
263         size_t out_data_1_len;
264         uint64_t counter;
265         uint64_t counter_mask = ntohll(0x00000000ffffffffULL);
266         int rv = CRYPTO_SUCCESS;

268         GCM_ACCEL_ENTER;

270         /*
271          * GCM mode fastpath requirements:
272          * - fastpath is enabled
273          * - block size is 128 bits
274          * - input is block-aligned
275          * - the counter value won't overflow
276          * - output is a single contiguous region and doesn't alias input
277          */
278         if (gcm_fastpath_enabled && block_size == 16 &&
279             ctx->gcm_remainder_len == 0 && (length & (block_size - 1)) == 0 &&
280             ntohll(ctx->gcm_cb[1] & counter_mask) <= ntohll(counter_mask) -
281             length / block_size && CRYPTO_DATA_IS_SINGLE_BLOCK(out)) {
282                 gcm_fastpath128(ctx, (uint8_t *)data, length,
283                     CRYPTO_DATA_FIRST_BLOCK(out), encrypt, encrypt_block,
284                     xor_block, cipher_ctr);
285                 out->cd_offset += length;
286                 goto out;
287         }

289         if (length + ctx->gcm_remainder_len < block_size) {
290                 /* accumulate bytes here and return */
291                 bcopy(datap,
292                     (uint8_t *)ctx->gcm_remainder + ctx->gcm_remainder_len,
293                     length);
294                 ctx->gcm_remainder_len += length;
295                 ctx->gcm_copy_to = datap;
296                 goto out;
157                 return (CRYPTO_SUCCESS);
297         }

299         lastp = (uint8_t *)ctx->gcm_cb;
300         if (out != NULL)
301                 crypto_init_ptrs(out, &iov_or_mp, &offset);

303         do {
304                 /* Unprocessed data from last call. */
```



```
305                 if (ctx->gcm_remainder_len > 0) {
306                         need = block_size - ctx->gcm_remainder_len;

308                         if (need > remainder) {
309                                 rv = SET_ERROR(CRYPTO_DATA_LEN_RANGE);
310                                 goto out;
311                         }
169                         if (need > remainder)
170                                 return (CRYPTO_DATA_LEN_RANGE);

313                         bcopy(datap, &((uint8_t *)ctx->gcm_remainder)
314                             [ctx->gcm_remainder_len], need);

316                         blockp = (uint8_t *)ctx->gcm_remainder;
317                 } else {
318                         blockp = datap;
319                 }

321                 /* add ciphertext to the hash */
322                 if (!encrypt)
323                         GHASH(ctx, blockp, ctx->gcm_ghash);

325                 /*
326                  * Increment counter. Counter bits are confined
327                  * to the bottom 32 bits of the counter block.
328                  */
329                 counter = ntohll(ctx->gcm_cb[1] & counter_mask);
330                 counter = htonll(counter + 1);
331                 counter &= counter_mask;
332                 ctx->gcm_cb[1] = (ctx->gcm_cb[1] & ~counter_mask) | counter;

334                 encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_cb,
335                     (uint8_t *)ctx->gcm_tmp);
336                 xor_block(blockp, (uint8_t *)ctx->gcm_tmp);

338                 lastp = (uint8_t *)ctx->gcm_tmp;

340                 ctx->gcm_processed_data_len += block_size;

342                 if (out == NULL) {
343                         if (ctx->gcm_remainder_len > 0) {
344                                 bcopy(blockp, ctx->gcm_copy_to,
345                                     ctx->gcm_remainder_len);
346                                 bcopy(blockp + ctx->gcm_remainder_len, datap,
347                                     need);
348                         }
349                 } else {
350                         crypto_get_ptrs(out, &iov_or_mp, &offset, &out_data_1,
351                             &out_data_1_len, &out_data_2, block_size);

353                         /* copy block to where it belongs */
354                         if (out_data_1_len == block_size) {
355                                 copy_block(lastp, out_data_1);
356                         } else {
357                                 bcopy(lastp, out_data_1, out_data_1_len);
358                                 if (out_data_2 != NULL) {
359                                         bcopy(lastp + out_data_1_len,
360                                             out_data_2,
361                                             block_size - out_data_1_len);
362                                 }
363                         }
364                         /* update offset */
365                         out->cd_offset += block_size;
366                 }

368                 /* add ciphertext to the hash */
```

```
369                 if (encrypt)
370                         GHASH(ctx, ctx->gcm_tmp, ctx->gcm_ghash);

372                 /* Update pointer to next block of data to be processed. */
373                 if (ctx->gcm_remainder_len != 0) {
374                         datap += need;
375                         ctx->gcm_remainder_len = 0;
376                 } else {
377                         datap += block_size;
378                 }

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

382                 /* Incomplete last block. */
383                 if (remainder > 0 && remainder < block_size) {
384                         bcopy(datap, ctx->gcm_remainder, remainder);
385                         ctx->gcm_remainder_len = remainder;
386                         ctx->gcm_copy_to = datap;
387                         goto out;
388                 }
389                 ctx->gcm_copy_to = NULL;

391         } while (remainder > 0);
392 out:
393         GCM_ACCEL_EXIT;

395         return (rv);
247         return (CRYPTO_SUCCESS);
396 }


399 /*
400  * Encrypt multiple blocks of data in GCM mode.  Decrypt for GCM mode
401  * is done in another function.
402  */
403 /*ARGSUSED*/
404 int
405 gcm_mode_encrypt_contiguous_blocks(gcm_ctx_t *ctx, char *data, size_t length,
406     crypto_data_t *out, size_t block_size,
407     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
408     void (*copy_block)(const uint8_t *, uint8_t *),
409     void (*xor_block)(const uint8_t *, uint8_t *),
410     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
411     uint64_t *))
412 {
413         return (gcm_process_contiguous_blocks(ctx, data, length, out,
414             block_size, B_TRUE, encrypt_block, copy_block, xor_block,
415             cipher_ctr));
416 }

418 /* ARGSUSED */
419 int
420 gcm_encrypt_final(gcm_ctx_t *ctx, crypto_data_t *out, size_t block_size,
421     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
422     void (*copy_block)(const uint8_t *, uint8_t *),
423     void (*xor_block)(const uint8_t *, uint8_t *))
254     void (*copy_block)(uint8_t *, uint8_t *),
255     void (*xor_block)(uint8_t *, uint8_t *))
424 {
425         uint64_t counter_mask = ntohll(0x00000000ffffffffULL);
426         uint8_t *ghash, *macp;
427         int i, rv;

429         GCM_ACCEL_ENTER;

431         if (out->cd_length < (ctx->gcm_remainder_len + ctx->gcm_tag_len)) {
```



```
432                 rv = CRYPTO_DATA_LEN_RANGE;
433                 goto out;
261         if (out->cd_length <
262             (ctx->gcm_remainder_len + ctx->gcm_tag_len)) {
263                 return (CRYPTO_DATA_LEN_RANGE);
434         }

436         ghash = (uint8_t *)ctx->gcm_ghash;

438         if (ctx->gcm_remainder_len > 0) {
439                 uint64_t counter;
440                 uint8_t *tmpp = (uint8_t *)ctx->gcm_tmp;

442                 /*
443                  * Here is where we deal with data that is not a
444                  * multiple of the block size.
445                  */

447                 /*
448                  * Increment counter.
449                  */
450                 counter = ntohll(ctx->gcm_cb[1] & counter_mask);
451                 counter = htonll(counter + 1);
452                 counter &= counter_mask;
453                 ctx->gcm_cb[1] = (ctx->gcm_cb[1] & ~counter_mask) | counter;

455                 encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_cb,
456                     (uint8_t *)ctx->gcm_tmp);

458                 macp = (uint8_t *)ctx->gcm_remainder;
459                 bzero(macp + ctx->gcm_remainder_len,
460                     block_size - ctx->gcm_remainder_len);

462                 /* XOR with counter block */
463                 for (i = 0; i < ctx->gcm_remainder_len; i++) {
464                         macp[i] ^= tmpp[i];
465                 }

467                 /* add ciphertext to the hash */
468                 GHASH(ctx, macp, ghash);

470                 ctx->gcm_processed_data_len += ctx->gcm_remainder_len;
471         }

473         ctx->gcm_len_a_len_c[1] =
474             htonll(CRYPTO_BYTES2BITS(ctx->gcm_processed_data_len));
475         GHASH(ctx, ctx->gcm_len_a_len_c, ghash);
476         encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_J0,
477             (uint8_t *)ctx->gcm_J0);
478         xor_block((uint8_t *)ctx->gcm_J0, ghash);

480         if (ctx->gcm_remainder_len > 0) {
481                 rv = crypto_put_output_data(macp, out, ctx->gcm_remainder_len);
482                 if (rv != CRYPTO_SUCCESS)
483                         goto out;
313                         return (rv);
484         }
485         out->cd_offset += ctx->gcm_remainder_len;
486         ctx->gcm_remainder_len = 0;
487         rv = crypto_put_output_data(ghash, out, ctx->gcm_tag_len);
488         if (rv != CRYPTO_SUCCESS)
489                 goto out;
490         out->cd_offset += ctx->gcm_tag_len;
491 out:
492         GCM_ACCEL_EXIT;
493         return (rv);
```

```
320         out->cd_offset += ctx->gcm_tag_len;

322         return (CRYPTO_SUCCESS);
494 }

496 /*
497  * This will only deal with decrypting the last block of the input that
498  * might not be a multiple of block length.
499  */
500 /*ARGSUSED*/
501 static void
502 gcm_decrypt_incomplete_block(gcm_ctx_t *ctx, uint8_t *data, size_t length,
503     size_t block_size, crypto_data_t *out,
330 gcm_decrypt_incomplete_block(gcm_ctx_t *ctx, size_t block_size, size_t index,
504     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
505     void (*xor_block)(const uint8_t *, uint8_t *))
332     void (*xor_block)(uint8_t *, uint8_t *))
506 {
334         uint8_t *datap, *outp, *counterp;
507         uint64_t counter;
508         uint64_t counter_mask = ntohll(0x00000000ffffffffULL);
337         int i;

510         /* padd last block and add to GHASH */
511         bcopy(data, ctx->gcm_tmp, length);
512         bzero(((uint8_t *)ctx->gcm_tmp) + length,
513             sizeof (ctx->gcm_tmp) - length);
514         GHASH(ctx, ctx->gcm_tmp, ctx->gcm_ghash);

516         /*
517          * Increment counter.
518          * Counter bits are confined to the bottom 32 bits.
341          * Counter bits are confined to the bottom 32 bits
519          */
520         counter = ntohll(ctx->gcm_cb[1] & counter_mask);
521         counter = htonll(counter + 1);
522         counter &= counter_mask;
523         ctx->gcm_cb[1] = (ctx->gcm_cb[1] & ~counter_mask) | counter;

525         encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_cb,
526             (uint8_t *)ctx->gcm_tmp);
348         datap = (uint8_t *)ctx->gcm_remainder;
349         outp = &((ctx->gcm_pt_buf)[index]);
350         counterp = (uint8_t *)ctx->gcm_tmp;

528         /* XOR with counter block */
529         for (size_t i = 0; i < length; i++)
530                 ((uint8_t *)ctx->gcm_tmp)[i] ^= data[i];
352         /* authentication tag */
353         bzero((uint8_t *)ctx->gcm_tmp, block_size);
354         bcopy(datap, (uint8_t *)ctx->gcm_tmp, ctx->gcm_remainder_len);

532         if (out != NULL) {
533                 (void) crypto_put_output_data((uchar_t *)ctx->gcm_tmp, out,
534                     length);
535                 out->cd_offset += length;
536         } else {
537                 bcopy(ctx->gcm_tmp, data, length);
356         /* add ciphertext to the hash */
357         GHASH(ctx, ctx->gcm_tmp, ctx->gcm_ghash);

359         /* decrypt remaining ciphertext */
360         encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_cb, counterp);

362         /* XOR with counter block */
363         for (i = 0; i < ctx->gcm_remainder_len; i++) {
```



```
364                 outp[i] = datap[i] ^ counterp[i];
538         }
539 }

541 /* ARGSUSED */
542 int
543 gcm_mode_decrypt_contiguous_blocks(gcm_ctx_t *ctx, char *data, size_t length,
544     crypto_data_t *out, size_t block_size,
545     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
546     void (*copy_block)(const uint8_t *, uint8_t *),
547     void (*xor_block)(const uint8_t *, uint8_t *),
548     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
549     uint64_t *))
373     void (*copy_block)(uint8_t *, uint8_t *),
374     void (*xor_block)(uint8_t *, uint8_t *))
550 {
551         int rv = CRYPTO_SUCCESS;
376         size_t new_len;
377         uint8_t *new;

553         GCM_ACCEL_ENTER;

555         /*
556          * Previous calls accumulate data in the input buffer to make sure
557          * we have the auth tag (the last part of the ciphertext) when we
558          * receive a final() call.
380          * Copy contiguous ciphertext input blocks to plaintext buffer.
381          * Ciphertext will be decrypted in the final.
559          */
560         if (ctx->gcm_last_input_fill > 0) {
561                 /* Try to complete the input buffer */
562                 size_t to_copy = MIN(length,
563                     sizeof (ctx->gcm_last_input) - ctx->gcm_last_input_fill);
383         if (length > 0) {
384                 new_len = ctx->gcm_pt_buf_len + length;
385 #ifdef _KERNEL
386                 new = kmem_alloc(new_len, ctx->gcm_kmflag);
387                 bcopy(ctx->gcm_pt_buf, new, ctx->gcm_pt_buf_len);
388                 kmem_free(ctx->gcm_pt_buf, ctx->gcm_pt_buf_len);
389 #else
390                 new = malloc(new_len);
391                 bcopy(ctx->gcm_pt_buf, new, ctx->gcm_pt_buf_len);
392                 free(ctx->gcm_pt_buf);
393 #endif
394                 if (new == NULL)
395                         return (CRYPTO_HOST_MEMORY);

565                 bcopy(data, ctx->gcm_last_input + ctx->gcm_last_input_fill,
566                     to_copy);
567                 data += to_copy;
568                 ctx->gcm_last_input_fill += to_copy;
569                 length -= to_copy;

571                 if (ctx->gcm_last_input_fill < sizeof (ctx->gcm_last_input))
572                         /* Not enough input data to continue */
573                         goto out;

575                 if (length < ctx->gcm_tag_len) {
576                         /*
577                          * There isn't enough data ahead to constitute a full
578                          * auth tag, so only crunch one input block and copy
579                          * the remainder of the input into our buffer.
580                          */
581                         rv = gcm_process_contiguous_blocks(ctx,
582                             (char *)ctx->gcm_last_input, block_size, out,
583                             block_size, B_FALSE, encrypt_block, copy_block,
```

```
584                             xor_block, cipher_ctr);
585                         if (rv != CRYPTO_SUCCESS)
586                                 goto out;
587                         ctx->gcm_last_input_fill -= block_size;
588                         bcopy(ctx->gcm_last_input + block_size,
589                             ctx->gcm_last_input, ctx->gcm_last_input_fill);
590                         bcopy(data, ctx->gcm_last_input +
591                             ctx->gcm_last_input_fill, length);
592                         ctx->gcm_last_input_fill += length;
593                         /* No more input left */
594                         goto out;
397                 ctx->gcm_pt_buf = new;
398                 ctx->gcm_pt_buf_len = new_len;
399                 bcopy(data, &ctx->gcm_pt_buf[ctx->gcm_processed_data_len],
400                     length);
401                 ctx->gcm_processed_data_len += length;
595                 }
596                 /*
597                  * There is enough data ahead for the auth tag, so crunch
598                  * everything in our buffer now and empty it.
599                  */
600                 rv = gcm_process_contiguous_blocks(ctx,
601                     (char *)ctx->gcm_last_input, ctx->gcm_last_input_fill,
602                     out, block_size, B_FALSE, encrypt_block, copy_block,
603                     xor_block, cipher_ctr);
604                 if (rv != CRYPTO_SUCCESS)
605                         goto out;
606                 ctx->gcm_last_input_fill = 0;
607         }
608         /*
609          * Last input buffer is empty, so what's left ahead is block-aligned.
610          * Crunch all the blocks up until the near end, which might be our
611          * auth tag and we must NOT decrypt.
612          */
613         ASSERT(ctx->gcm_last_input_fill == 0);
614         if (length >= block_size + ctx->gcm_tag_len) {
615                 size_t to_decrypt = (length - ctx->gcm_tag_len) &
616                     ~(block_size - 1);

618                 rv = gcm_process_contiguous_blocks(ctx, data, to_decrypt, out,
619                     block_size, B_FALSE, encrypt_block, copy_block, xor_block,
620                     cipher_ctr);
621                 if (rv != CRYPTO_SUCCESS)
622                         goto out;
623                 data += to_decrypt;
624                 length -= to_decrypt;
625         }
626         /*
627          * Copy the remainder into our input buffer, it's potentially
628          * the auth tag and a last partial block.
629          */
630         ASSERT(length < sizeof (ctx->gcm_last_input));
631         bcopy(data, ctx->gcm_last_input, length);
632         ctx->gcm_last_input_fill += length;
633 out:
634         GCM_ACCEL_EXIT;

636         return (rv);
404         ctx->gcm_remainder_len = 0;
405         return (CRYPTO_SUCCESS);
637 }

639 int
640 gcm_decrypt_final(gcm_ctx_t *ctx, crypto_data_t *out, size_t block_size,
641     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
642     void (*copy_block)(const uint8_t *, uint8_t *),
```



```
643     void (*xor_block)(const uint8_t *, uint8_t *),
644     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
645     uint64_t *))
411     void (*xor_block)(uint8_t *, uint8_t *))
646 {
647         int rv = CRYPTO_SUCCESS;
413         size_t pt_len;
414         size_t remainder;
415         uint8_t *ghash;
416         uint8_t *blockp;
417         uint8_t *cbp;
418         uint64_t counter;
419         uint64_t counter_mask = ntohll(0x00000000ffffffffULL);
420         int processed = 0, rv;

649         /* Check there's enough data to at least compute a tag */
650         if (ctx->gcm_last_input_fill < ctx->gcm_tag_len)
651                 return (SET_ERROR(CRYPTO_DATA_LEN_RANGE));
422         ASSERT(ctx->gcm_processed_data_len == ctx->gcm_pt_buf_len);

653         GCM_ACCEL_ENTER;

655         /* Finish any unprocessed input */
656         if (ctx->gcm_last_input_fill > ctx->gcm_tag_len) {
657                 size_t last_blk_len = MIN(block_size,
658                     ctx->gcm_last_input_fill - ctx->gcm_tag_len);

660                 /* Finish last full block */
661                 if (last_blk_len >= block_size) {
662                         rv = gcm_process_contiguous_blocks(ctx,
663                             (char *)ctx->gcm_last_input, block_size, out,
664                             block_size, B_FALSE, encrypt_block, copy_block,
665                             xor_block, cipher_ctr);
666                         if (rv != CRYPTO_SUCCESS)
667                                 goto errout;

669                         last_blk_len -= block_size;
670                         ctx->gcm_processed_data_len += block_size;
671                         ctx->gcm_last_input_fill -= block_size;

673                         /* Shift what remains in the input buffer forward */
674                         bcopy(ctx->gcm_last_input + block_size,
675                             ctx->gcm_last_input, ctx->gcm_last_input_fill);
424         pt_len = ctx->gcm_processed_data_len - ctx->gcm_tag_len;
425         ghash = (uint8_t *)ctx->gcm_ghash;
426         blockp = ctx->gcm_pt_buf;
427         remainder = pt_len;
428         while (remainder > 0) {
429                 /* Incomplete last block */
430                 if (remainder < block_size) {
431                         bcopy(blockp, ctx->gcm_remainder, remainder);
432                         ctx->gcm_remainder_len = remainder;
433                         /*
434                          * not expecting anymore ciphertext, just
435                          * compute plaintext for the remaining input
436                          */
437                         gcm_decrypt_incomplete_block(ctx, block_size,
438                             processed, encrypt_block, xor_block);
439                         ctx->gcm_remainder_len = 0;
440                         goto out;
676                 }
677                 /* Finish last incomplete block before auth tag */
678                 if (last_blk_len > 0) {
679                         gcm_decrypt_incomplete_block(ctx, ctx->gcm_last_input,
680                             last_blk_len, block_size, out, encrypt_block,
681                             xor_block);
```

```
442                 /* add ciphertext to the hash */
443                 GHASH(ctx, blockp, ghash);

683                         ctx->gcm_processed_data_len += last_blk_len;
684                         ctx->gcm_last_input_fill -= last_blk_len;
445                 /*
446                  * Increment counter.
447                  * Counter bits are confined to the bottom 32 bits
448                  */
449                 counter = ntohll(ctx->gcm_cb[1] & counter_mask);
450                 counter = htonll(counter + 1);
451                 counter &= counter_mask;
452                 ctx->gcm_cb[1] = (ctx->gcm_cb[1] & ~counter_mask) | counter;

686                         /* Shift what remains in the input buffer forward */
687                         bcopy(ctx->gcm_last_input + last_blk_len,
688                             ctx->gcm_last_input, ctx->gcm_last_input_fill);
689                 }
690                 /* Now the last_input buffer holds just the auth tag */
691         }
454                 cbp = (uint8_t *)ctx->gcm_tmp;
455                 encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_cb, cbp);

693         ASSERT(ctx->gcm_last_input_fill == ctx->gcm_tag_len);
457                 /* XOR with ciphertext */
458                 xor_block(cbp, blockp);

695         ctx->gcm_len_a_len_c[1] =
696             htonll(CRYPTO_BYTES2BITS(ctx->gcm_processed_data_len));
697         GHASH(ctx, ctx->gcm_len_a_len_c, ctx->gcm_ghash);
460                 processed += block_size;
461                 blockp += block_size;
462                 remainder -= block_size;
463         }
464 out:
465         ctx->gcm_len_a_len_c[1] = htonll(CRYPTO_BYTES2BITS(pt_len));
466         GHASH(ctx, ctx->gcm_len_a_len_c, ghash);
698         encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_J0,
699             (uint8_t *)ctx->gcm_J0);
700         xor_block((uint8_t *)ctx->gcm_J0, (uint8_t *)ctx->gcm_ghash);
469         xor_block((uint8_t *)ctx->gcm_J0, ghash);

702         GCM_ACCEL_EXIT;

704         /* compare the input authentication tag with what we calculated */
705         if (bcmp(&ctx->gcm_last_input, ctx->gcm_ghash, ctx->gcm_tag_len) != 0)
706                 return (SET_ERROR(CRYPTO_INVALID_MAC));

708         return (CRYPTO_SUCCESS);

710 errout:
711         GCM_ACCEL_EXIT;
472         if (bcmp(&ctx->gcm_pt_buf[pt_len], ghash, ctx->gcm_tag_len)) {
473                 /* They don't match */
474                 return (CRYPTO_INVALID_MAC);
475         } else {
476                 rv = crypto_put_output_data(ctx->gcm_pt_buf, out, pt_len);
477                 if (rv != CRYPTO_SUCCESS)
712         return (rv);
479                 out->cd_offset += pt_len;
480         }
481         return (CRYPTO_SUCCESS);
713 }

715 static int
716 gcm_validate_args(CK_AES_GCM_PARAMS *gcm_param)
```



```
717 {
718         size_t tag_len;

720         /*
721          * Check the length of the authentication tag (in bits).
722          */
723         tag_len = gcm_param->ulTagBits;
724         switch (tag_len) {
725         case 32:
726         case 64:
727         case 96:
728         case 104:
729         case 112:
730         case 120:
731         case 128:
732                 break;
733         default:
734                 return (SET_ERROR(CRYPTO_MECHANISM_PARAM_INVALID));
503                 return (CRYPTO_MECHANISM_PARAM_INVALID);
735         }

737         if (gcm_param->ulIvLen == 0)
738                 return (SET_ERROR(CRYPTO_MECHANISM_PARAM_INVALID));
507                 return (CRYPTO_MECHANISM_PARAM_INVALID);

740         return (CRYPTO_SUCCESS);
741 }

743 /*ARGSUSED*/
744 static void
745 gcm_format_initial_blocks(uchar_t *iv, ulong_t iv_len,
746     gcm_ctx_t *ctx, size_t block_size,
747     void (*copy_block)(const uint8_t *, uint8_t *),
748     void (*xor_block)(const uint8_t *, uint8_t *))
515     void (*copy_block)(uint8_t *, uint8_t *),
516     void (*xor_block)(uint8_t *, uint8_t *))
749 {
750         uint8_t *cb;
751         ulong_t remainder = iv_len;
752         ulong_t processed = 0;
753         uint8_t *datap, *ghash;
754         uint64_t len_a_len_c[2];

756         ghash = (uint8_t *)ctx->gcm_ghash;
757         cb = (uint8_t *)ctx->gcm_cb;
758         if (iv_len == 12) {
759                 bcopy(iv, cb, 12);
760                 cb[12] = 0;
761                 cb[13] = 0;
762                 cb[14] = 0;
763                 cb[15] = 1;
764                 /* J0 will be used again in the final */
765                 copy_block(cb, (uint8_t *)ctx->gcm_J0);
766         } else {
767                 /* GHASH the IV */
768                 do {
769                         if (remainder < block_size) {
770                                 bzero(cb, block_size);
771                                 bcopy(&(iv[processed]), cb, remainder);
772                                 datap = (uint8_t *)cb;
773                                 remainder = 0;
774                         } else {
775                                 datap = (uint8_t *)(&(iv[processed]));
776                                 processed += block_size;
777                                 remainder -= block_size;
778                         }
```

```
779                         GHASH(ctx, datap, ghash);
780                 } while (remainder > 0);
781
782                 len_a_len_c[0] = 0;
783                 len_a_len_c[1] = htonll(CRYPTO_BYTES2BITS(iv_len));
784                 GHASH(ctx, len_a_len_c, ctx->gcm_J0);
785
786                 /* J0 will be used again in the final */
787                 copy_block((uint8_t *)ctx->gcm_J0, (uint8_t *)cb);
788         }
789 }
790
791 /*
792  * The following function is called at encrypt or decrypt init time
793  * for AES GCM mode.
794  */
795 int
796 gcm_init(gcm_ctx_t *ctx, unsigned char *iv, size_t iv_len,
797     unsigned char *auth_data, size_t auth_data_len, size_t block_size,
798     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
799     void (*copy_block)(const uint8_t *, uint8_t *),
800     void (*xor_block)(const uint8_t *, uint8_t *))
567     void (*copy_block)(uint8_t *, uint8_t *),
568     void (*xor_block)(uint8_t *, uint8_t *))
801 {
802         uint8_t *ghash, *datap, *authp;
803         size_t remainder, processed;
804
805         GCM_ACCEL_ENTER;
806
807         /* encrypt zero block to get subkey H */
808         bzero(ctx->gcm_H, sizeof (ctx->gcm_H));
809         encrypt_block(ctx->gcm_keysched, (uint8_t *)ctx->gcm_H,
810             (uint8_t *)ctx->gcm_H);
811
812         gcm_format_initial_blocks(iv, iv_len, ctx, block_size,
813             copy_block, xor_block);
814
815 #ifdef  __amd64
816         if (intel_pclmulqdq_instruction_present()) {
817                 uint64_t H_bswap64[2] = {
818                     ntohll(ctx->gcm_H[0]), ntohll(ctx->gcm_H[1])
819                 };
820
821                 gcm_init_clmul(H_bswap64, ctx->gcm_H_table);
822         }
823 #endif
824
825         authp = (uint8_t *)ctx->gcm_tmp;
826         ghash = (uint8_t *)ctx->gcm_ghash;
827         bzero(authp, block_size);
828         bzero(ghash, block_size);
829
830         processed = 0;
831         remainder = auth_data_len;
832         do {
833                 if (remainder < block_size) {
834                         /*
835                          * There's not a block full of data, pad rest of
836                          * buffer with zero
837                          */
838                         bzero(authp, block_size);
839                         bcopy(&(auth_data[processed]), authp, remainder);
840                         datap = (uint8_t *)authp;
841                         remainder = 0;
842                 } else {
```



```
843                         datap = (uint8_t *)(&(auth_data[processed]));
844                         processed += block_size;
845                         remainder -= block_size;
846                 }
847
848                 /* add auth data to the hash */
849                 GHASH(ctx, datap, ghash);
850
851         } while (remainder > 0);
852
853         GCM_ACCEL_EXIT;
854
855         return (CRYPTO_SUCCESS);
856 }
857
858 int
859 gcm_init_ctx(gcm_ctx_t *gcm_ctx, char *param, size_t block_size,
860     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
861     void (*copy_block)(const uint8_t *, uint8_t *),
862     void (*xor_block)(const uint8_t *, uint8_t *))
615     void (*copy_block)(uint8_t *, uint8_t *),
616     void (*xor_block)(uint8_t *, uint8_t *))
863 {
864         /*
865          * No GHASH invocations in this function and gcm_init does its own
866          * FPU saving, so no need to GCM_ACCEL_ENTER/GCM_ACCEL_EXIT here.
867          */
868         int rv;
869         CK_AES_GCM_PARAMS *gcm_param;
870
871         if (param != NULL) {
872                 gcm_param = (CK_AES_GCM_PARAMS *)(void *)param;
873
874                 if ((rv = gcm_validate_args(gcm_param)) != 0) {
875                         return (rv);
876                 }
877
878                 gcm_ctx->gcm_tag_len = gcm_param->ulTagBits;
879                 gcm_ctx->gcm_tag_len >>= 3;
880                 gcm_ctx->gcm_processed_data_len = 0;
881
882                 /* these values are in bits */
883                 gcm_ctx->gcm_len_a_len_c[0]
884                     = htonll(CRYPTO_BYTES2BITS(gcm_param->ulAADLen));
885
886                 rv = CRYPTO_SUCCESS;
887                 gcm_ctx->gcm_flags |= GCM_MODE;
888         } else {
889                 rv = CRYPTO_MECHANISM_PARAM_INVALID;
890                 goto out;
891         }
892
893         if (gcm_init(gcm_ctx, gcm_param->pIv, gcm_param->ulIvLen,
894             gcm_param->pAAD, gcm_param->ulAADLen, block_size,
895             encrypt_block, copy_block, xor_block) != 0) {
896                 rv = CRYPTO_MECHANISM_PARAM_INVALID;
897         }
898 out:
899         return (rv);
900 }
901
902 int
903 gmac_init_ctx(gcm_ctx_t *gcm_ctx, char *param, size_t block_size,
904     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
905     void (*copy_block)(const uint8_t *, uint8_t *),
906     void (*xor_block)(const uint8_t *, uint8_t *))
```

```
655     void (*copy_block)(uint8_t *, uint8_t *),
656     void (*xor_block)(uint8_t *, uint8_t *))
907 {
908         /*
909          * No GHASH invocations in this function and gcm_init does its own
910          * FPU saving, so no need to GCM_ACCEL_ENTER/GCM_ACCEL_EXIT here.
911          */
912         int rv;
913         CK_AES_GMAC_PARAMS *gmac_param;

915         if (param != NULL) {
916                 gmac_param = (CK_AES_GMAC_PARAMS *)(void *)param;

918                 gcm_ctx->gcm_tag_len = CRYPTO_BITS2BYTES(AES_GMAC_TAG_BITS);
919                 gcm_ctx->gcm_processed_data_len = 0;

921                 /* these values are in bits */
922                 gcm_ctx->gcm_len_a_len_c[0]
923                     = htonll(CRYPTO_BYTES2BITS(gmac_param->ulAADLen));

925                 rv = CRYPTO_SUCCESS;
926                 gcm_ctx->gcm_flags |= GMAC_MODE;
927         } else {
928                 rv = CRYPTO_MECHANISM_PARAM_INVALID;
929                 goto out;
930         }

932         if (gcm_init(gcm_ctx, gmac_param->pIv, AES_GMAC_IV_LEN,
933             gmac_param->pAAD, gmac_param->ulAADLen, block_size,
934             encrypt_block, copy_block, xor_block) != 0) {
935                 rv = CRYPTO_MECHANISM_PARAM_INVALID;
936         }
937 out:
938         return (rv);
939 }
```

*unchanged_portion_omitted_*

```
980 #ifdef __amd64
981 /*
982  * Return 1 if executing on Intel with PCLMULQDQ instructions,
983  * otherwise 0 (i.e., Intel without PCLMULQDQ or AMD64).
984  * Cache the result, as the CPU can't change.
985  *
986  * Note: the userland version uses getisax().  The kernel version uses
987  * is_x86_featureset().
988  */
989 static inline int
735 static int
990 intel_pclmulqdq_instruction_present(void)
991 {
992         static int      cached_result = -1;

994         if (cached_result == -1) { /* first time */
995 #ifdef _KERNEL
996                 cached_result =
997                     is_x86_feature(x86_featureset, X86FSET_PCLMULQDQ);
998 #else
999                 uint_t          ui = 0;

1001                (void) getisax(&ui, 1);
1002                cached_result = (ui & AV_386_PCLMULQDQ) != 0;
1003 #endif  /* _KERNEL */
1004        }

1006        return (cached_result);
```



```
1007 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    3104 Thu Apr 30 20:52:31 2015
new/usr/src/common/crypto/modes/modes.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#ifndef _KERNEL
#include <stdlib.h>
#endif

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

/*
 * Initialize by setting iov_or_mp to point to the current iovec or mp,
 * and by setting current_offset to an offset within the current iovec or mp.
 */
void
crypto_init_ptrs(crypto_data_t *out, void **iov_or_mp, offset_t *current_offset)
{
        offset_t offset;

        switch (out->cd_format) {
        case CRYPTO_DATA_RAW:
                *current_offset = out->cd_offset;
                break;

        case CRYPTO_DATA_UIO: {
                uio_t *uiop = out->cd_uio;
                uintptr_t vec_idx;

                offset = out->cd_offset;
                for (vec_idx = 0; vec_idx < uiop->uio_iovcnt &&
                    offset >= uiop->uio_iov[vec_idx].iov_len;
                    offset -= uiop->uio_iov[vec_idx++].iov_len)
                        ;
```



```
                *current_offset = offset;
                *iov_or_mp = (void *)vec_idx;
                break;
        }

        case CRYPTO_DATA_MBLK: {
                mblk_t *mp;

                offset = out->cd_offset;
                for (mp = out->cd_mp; mp != NULL && offset >= MBLKL(mp);
                    offset -= MBLKL(mp), mp = mp->b_cont)
                        ;

                *current_offset = offset;
                *iov_or_mp = mp;
                break;

        }
        } /* end switch */
}

/*
 * Get pointers for where in the output to copy a block of encrypted or
 * decrypted data.  The iov_or_mp argument stores a pointer to the current
 * iovec or mp, and offset stores an offset into the current iovec or mp.
 */
void
crypto_get_ptrs(crypto_data_t *out, void **iov_or_mp, offset_t *current_offset,
    uint8_t **out_data_1, size_t *out_data_1_len, uint8_t **out_data_2,
    size_t amt)
{
        offset_t offset;

        switch (out->cd_format) {
        case CRYPTO_DATA_RAW: {
                iovec_t *iov;

                offset = *current_offset;
                iov = &out->cd_raw;
                if ((offset + amt) <= iov->iov_len) {
                        /* one block fits */
                        *out_data_1 = (uint8_t *)iov->iov_base + offset;
                        *out_data_1_len = amt;
                        *out_data_2 = NULL;
                        *current_offset = offset + amt;
                }
                break;
        }

        case CRYPTO_DATA_UIO: {
                uio_t *uio = out->cd_uio;
                iovec_t *iov;
                offset_t offset;
                uintptr_t vec_idx;
                uint8_t *p;

                offset = *current_offset;
                vec_idx = (uintptr_t)(*iov_or_mp);
                iov = &uio->uio_iov[vec_idx];
                p = (uint8_t *)iov->iov_base + offset;
                *out_data_1 = p;

                if (offset + amt <= iov->iov_len) {
                        /* can fit one block into this iov */
                        *out_data_1_len = amt;
```

```
125                         *out_data_2 = NULL;
126                         *current_offset = offset + amt;
127                 } else {
128                         /* one block spans two iovecs */
129                         *out_data_1_len = iov->iov_len - offset;
130                         if (vec_idx == uio->uio_iovcnt)
131                                 return;
132                         vec_idx++;
133                         iov = &uio->uio_iov[vec_idx];
134                         *out_data_2 = (uint8_t *)iov->iov_base;
135                         *current_offset = amt - *out_data_1_len;
136                 }
137                 *iov_or_mp = (void *)vec_idx;
138                 break;
139         }

141         case CRYPTO_DATA_MBLK: {
142                 mblk_t *mp;
143                 uint8_t *p;

145                 offset = *current_offset;
146                 mp = (mblk_t *)*iov_or_mp;
147                 p = mp->b_rptr + offset;
148                 *out_data_1 = p;
149                 if ((p + amt) <= mp->b_wptr) {
150                         /* can fit one block into this mblk */
151                         *out_data_1_len = amt;
152                         *out_data_2 = NULL;
153                         *current_offset = offset + amt;
154                 } else {
155                         /* one block spans two mblks */
156                         *out_data_1_len = _PTRDIFF(mp->b_wptr, p);
157                         if ((mp = mp->b_cont) == NULL)
158                                 return;
159                         *out_data_2 = mp->b_rptr;
160                         *current_offset = (amt - *out_data_1_len);
161                 }
162                 *iov_or_mp = mp;
163                 break;
164         }
165         } /* end switch */
166 }

168 void
 85 crypto_free_mode_ctx(void *ctx)
 86 {
 87         common_ctx_t *common_ctx = (common_ctx_t *)ctx;

 89         switch (common_ctx->cc_flags &
 90             (ECB_MODE|CBC_MODE|CTR_MODE|CCM_MODE|GCM_MODE|GMAC_MODE)) {
 91         case ECB_MODE:
 92 #ifdef _KERNEL
 93                 kmem_free(common_ctx, sizeof (ecb_ctx_t));
 94 #else
 95                 free(common_ctx);
 96 #endif
 97                 break;

 99         case CBC_MODE:
100 #ifdef _KERNEL
101                 kmem_free(common_ctx, sizeof (cbc_ctx_t));
102 #else
103                 free(common_ctx);
104 #endif
105                 break;
```



```
107         case CTR_MODE:
108 #ifdef _KERNEL
109                 kmem_free(common_ctx, sizeof (ctr_ctx_t));
110 #else
111                 free(common_ctx);
112 #endif
113                 break;

115         case CCM_MODE:
116 #ifdef _KERNEL
117                 if (((ccm_ctx_t *)ctx)->ccm_pt_buf != NULL)
118                         kmem_free(((ccm_ctx_t *)ctx)->ccm_pt_buf,
119                             ((ccm_ctx_t *)ctx)->ccm_data_len);

121                 kmem_free(ctx, sizeof (ccm_ctx_t));
122 #else
123                 if (((ccm_ctx_t *)ctx)->ccm_pt_buf != NULL)
124                         free(((ccm_ctx_t *)ctx)->ccm_pt_buf);
125                 free(ctx);
126 #endif
127                 break;

129         case GCM_MODE:
130         case GMAC_MODE:
131 #ifdef _KERNEL
216                 if (((gcm_ctx_t *)ctx)->gcm_pt_buf != NULL)
217                         kmem_free(((gcm_ctx_t *)ctx)->gcm_pt_buf,
218                             ((gcm_ctx_t *)ctx)->gcm_pt_buf_len);

132                 kmem_free(ctx, sizeof (gcm_ctx_t));
133 #else
222                 if (((gcm_ctx_t *)ctx)->gcm_pt_buf != NULL)
223                         free(((gcm_ctx_t *)ctx)->gcm_pt_buf);
134                 free(ctx);
135 #endif
136         }
137 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   17181 Thu Apr 30 20:52:31 2015
new/usr/src/common/crypto/modes/modes.h
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#ifndef _COMMON_CRYPTO_MODES_H
#define _COMMON_CRYPTO_MODES_H

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/strsun.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/rwlock.h>
#include <sys/kmem.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>

#define ECB_MODE                        0x00000002
#define CBC_MODE                        0x00000004
#define CTR_MODE                        0x00000008
#define CCM_MODE                        0x00000010
#define GCM_MODE                        0x00000020
#define GMAC_MODE                       0x00000040

/*
 * cc_keysched:         Pointer to key schedule.
 *
 * cc_keysched_len:     Length of the key schedule.
 *
 * cc_remainder:        This is for residual data, i.e. data that can't
 *                      be processed because there are too few bytes.
 *                      Must wait until more data arrives.
 *
```



```
 * cc_remainder_len:    Number of bytes in cc_remainder.
 *
 * cc_iv:               Scratch buffer that sometimes contains the IV.
 *
 * cc_lastp:            Pointer to previous block of ciphertext.
 *
 * cc_copy_to:          Pointer to where encrypted residual data needs
 *                      to be copied.
 *
 * cc_flags:            PROVIDER_OWNS_KEY_SCHEDULE
 *                      When a context is freed, it is necessary
 *                      to know whether the key schedule was allocated
 *                      by the caller, or internally, e.g. an init routine.
 *                      If allocated by the latter, then it needs to be freed.
 *
 *                      ECB_MODE, CBC_MODE, CTR_MODE, or CCM_MODE
 */
struct common_ctx {
        void *cc_keysched;
        size_t cc_keysched_len;
        uint64_t cc_iv[2];
        uint64_t cc_remainder[2];
        size_t cc_remainder_len;
        uint8_t *cc_lastp;
        uint8_t *cc_copy_to;
        uint32_t cc_flags;
};
______unchanged_portion_omitted_

#define ccm_keysched            ccm_common.cc_keysched
#define ccm_keysched_len        ccm_common.cc_keysched_len
#define ccm_cb                  ccm_common.cc_iv
#define ccm_remainder           ccm_common.cc_remainder
#define ccm_remainder_len       ccm_common.cc_remainder_len
#define ccm_lastp               ccm_common.cc_lastp
#define ccm_copy_to             ccm_common.cc_copy_to
#define ccm_flags               ccm_common.cc_flags

/*
 * gcm_tag_len:         Length of authentication tag.
 *
 * gcm_ghash:           Stores output from the GHASH function.
 *
 * gcm_processed_data_len:
 *                      Length of processed plaintext (encrypt) or
 *                      length of processed ciphertext (decrypt).
 *
 * gcm_H:               Subkey.
 * gcm_pt_buf:          Stores the decrypted plaintext returned by
 *                      decrypt_final when the computed authentication
 *                      tag matches the user supplied tag.
 *
 * gcm_H_table:         Pipelined Karatsuba multipliers.
 * gcm_pt_buf_len:      Length of the plaintext buffer.
 *
 * gcm_H:               Subkey.
 *
 * gcm_J0:              Pre-counter block generated from the IV.
 *
 * gcm_tmp:             Temp storage for ciphertext when padding is needed.
 *
 * gcm_len_a_len_c:     64-bit representations of the bit lengths of
 *                      AAD and ciphertext.
 *
 * gcm_kmflag:          Current value of kmflag. Used only for allocating
 *                      the plaintext buffer during decryption.
```

```
 206  *
 207  * gcm_last_input:      Buffer of (up to) two last blocks. This is used when
 208  *                      input is not block-aligned and to temporarily hold
 209  *                      the end of the ciphertext stream during decryption,
 210  *                      since that could potentially be the GHASH auth tag
 211  *                      which we must check in the final() call instead of
 212  *                      decrypting it.
 213  *
 214  * gcm_last_input_fill: Number of bytes actually stored in gcm_last_input.
 215  */
 216 typedef struct gcm_ctx {
 217         struct common_ctx gcm_common;
 218         size_t gcm_tag_len;
 219         size_t gcm_processed_data_len;
 210         size_t gcm_pt_buf_len;
 211         uint32_t gcm_tmp[4];
 220         uint64_t gcm_ghash[2];
 221         uint64_t gcm_H[2];
 222 #ifdef  __amd64
 223         uint8_t gcm_H_table[256];
 224 #endif
 225         uint64_t gcm_J0[2];
 226         uint64_t gcm_tmp[2];
 227         uint64_t gcm_len_a_len_c[2];
 216         uint8_t *gcm_pt_buf;
 228         int gcm_kmflag;
 229         uint8_t gcm_last_input[32];
 230         size_t gcm_last_input_fill;
 231 } gcm_ctx_t;
______unchanged_portion_omitted_

 291 #define dc_flags                dcu.dcu_ecb.ecb_common.cc_flags
 292 #define dc_remainder_len        dcu.dcu_ecb.ecb_common.cc_remainder_len
 293 #define dc_keysched             dcu.dcu_ecb.ecb_common.cc_keysched
 294 #define dc_keysched_len         dcu.dcu_ecb.ecb_common.cc_keysched_len
 295 #define dc_iv                   dcu.dcu_ecb.ecb_common.cc_iv
 296 #define dc_lastp                dcu.dcu_ecb.ecb_common.cc_lastp

 298 extern int ecb_cipher_contiguous_blocks(ecb_ctx_t *, char *, size_t,
 299     crypto_data_t *, size_t,
 300     int (*cipher)(const void *, const uint8_t *, uint8_t *),
 301     int (*cipher_ecb)(const void *, const uint8_t *, uint8_t *, uint64_t));
 286     crypto_data_t *, size_t, int (*cipher)(const void *, const uint8_t *,
 287     uint8_t *));

 303 extern int cbc_encrypt_contiguous_blocks(cbc_ctx_t *, char *, size_t,
 304     crypto_data_t *, size_t,
 305     int (*encrypt)(const void *, const uint8_t *, uint8_t *),
 306     void (*copy_block)(const uint8_t *, uint8_t *),
 307     void (*xor_block)(const uint8_t *, uint8_t *),
 308     int (*encrypt_cbc)(const void *, const uint8_t *, uint8_t *,
 309     const uint8_t *, uint64_t));
 292     void (*copy_block)(uint8_t *, uint8_t *),
 293     void (*xor_block)(uint8_t *, uint8_t *));

 311 extern int cbc_decrypt_contiguous_blocks(cbc_ctx_t *, char *, size_t,
 312     crypto_data_t *, size_t,
 313     int (*decrypt)(const void *, const uint8_t *, uint8_t *),
 314     void (*copy_block)(const uint8_t *, uint8_t *),
 315     void (*xor_block)(const uint8_t *, uint8_t *),
 316     int (*decrypt_ecb)(const void *, const uint8_t *, uint8_t *, uint64_t),
 317     void (*xor_block_range)(const uint8_t *, uint8_t *, uint64_t));
 298     void (*copy_block)(uint8_t *, uint8_t *),
 299     void (*xor_block)(uint8_t *, uint8_t *));

 319 extern int ctr_mode_contiguous_blocks(ctr_ctx_t *, char *, size_t,
```



```
 320     crypto_data_t *, size_t,
 321     int (*cipher)(const void *, const uint8_t *, uint8_t *),
 322     void (*xor_block)(const uint8_t *, uint8_t *),
 323     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
 324     uint64_t *));
 304     void (*xor_block)(uint8_t *, uint8_t *));

 326 extern int ccm_mode_encrypt_contiguous_blocks(ccm_ctx_t *, char *, size_t,
 327     crypto_data_t *, size_t,
 328     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 329     void (*copy_block)(const uint8_t *, uint8_t *),
 330     void (*xor_block)(const uint8_t *, uint8_t *));
 309     void (*copy_block)(uint8_t *, uint8_t *),
 310     void (*xor_block)(uint8_t *, uint8_t *));

 332 extern int ccm_mode_decrypt_contiguous_blocks(ccm_ctx_t *, char *, size_t,
 333     crypto_data_t *, size_t,
 334     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 335     void (*copy_block)(const uint8_t *, uint8_t *),
 336     void (*xor_block)(const uint8_t *, uint8_t *));
 315     void (*copy_block)(uint8_t *, uint8_t *),
 316     void (*xor_block)(uint8_t *, uint8_t *));

 338 extern int gcm_mode_encrypt_contiguous_blocks(gcm_ctx_t *, char *, size_t,
 339     crypto_data_t *, size_t,
 340     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 341     void (*copy_block)(const uint8_t *, uint8_t *),
 342     void (*xor_block)(const uint8_t *, uint8_t *),
 343     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
 344     uint64_t *));
 321     void (*copy_block)(uint8_t *, uint8_t *),
 322     void (*xor_block)(uint8_t *, uint8_t *));

 346 extern int gcm_mode_decrypt_contiguous_blocks(gcm_ctx_t *, char *, size_t,
 347     crypto_data_t *, size_t,
 348     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 349     void (*copy_block)(const uint8_t *, uint8_t *),
 350     void (*xor_block)(const uint8_t *, uint8_t *),
 351     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
 352     uint64_t *));
 327     void (*copy_block)(uint8_t *, uint8_t *),
 328     void (*xor_block)(uint8_t *, uint8_t *));

 354 int ccm_encrypt_final(ccm_ctx_t *, crypto_data_t *, size_t,
 355     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 356     void (*xor_block)(const uint8_t *, uint8_t *));
 332     void (*xor_block)(uint8_t *, uint8_t *));

 358 int gcm_encrypt_final(gcm_ctx_t *, crypto_data_t *, size_t,
 359     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 360     void (*copy_block)(const uint8_t *, uint8_t *),
 361     void (*xor_block)(const uint8_t *, uint8_t *));
 336     void (*copy_block)(uint8_t *, uint8_t *),
 337     void (*xor_block)(uint8_t *, uint8_t *));

 363 extern int ccm_decrypt_final(ccm_ctx_t *, crypto_data_t *, size_t,
 364     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 365     void (*copy_block)(const uint8_t *, uint8_t *),
 366     void (*xor_block)(const uint8_t *, uint8_t *));
 341     void (*copy_block)(uint8_t *, uint8_t *),
 342     void (*xor_block)(uint8_t *, uint8_t *));

 368 extern int gcm_decrypt_final(gcm_ctx_t *, crypto_data_t *, size_t,
 369     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
 370     void (*copy_block)(const uint8_t *, uint8_t *),
 371     void (*xor_block)(const uint8_t *, uint8_t *),
```

```
372     int (*cipher_ctr)(const void *, const uint8_t *, uint8_t *, uint64_t,
373     uint64_t *));
346     void (*xor_block)(uint8_t *, uint8_t *));

375 extern int ctr_mode_final(ctr_ctx_t *, crypto_data_t *,
376     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *));

378 extern int cbc_init_ctx(cbc_ctx_t *, char *, size_t, size_t,
379     void (*copy_block)(const uint8_t *, uint64_t *));
352     void (*copy_block)(uint8_t *, uint64_t *));

381 extern int ctr_init_ctx(ctr_ctx_t *, ulong_t, uint8_t *,
382     void (*copy_block)(const uint8_t *, uint8_t *));
355     void (*copy_block)(uint8_t *, uint8_t *));

384 extern int ccm_init_ctx(ccm_ctx_t *, char *, int, boolean_t, size_t,
385     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
386     void (*xor_block)(const uint8_t *, uint8_t *));
359     void (*xor_block)(uint8_t *, uint8_t *));

388 extern int gcm_init_ctx(gcm_ctx_t *, char *, size_t,
389     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
390     void (*copy_block)(const uint8_t *, uint8_t *),
391     void (*xor_block)(const uint8_t *, uint8_t *));
363     void (*copy_block)(uint8_t *, uint8_t *),
364     void (*xor_block)(uint8_t *, uint8_t *));

393 extern int gmac_init_ctx(gcm_ctx_t *, char *, size_t,
394     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *),
395     void (*copy_block)(const uint8_t *, uint8_t *),
396     void (*xor_block)(const uint8_t *, uint8_t *));
368     void (*copy_block)(uint8_t *, uint8_t *),
369     void (*xor_block)(uint8_t *, uint8_t *));

398 extern void calculate_ccm_mac(ccm_ctx_t *, uint8_t *,
399     int (*encrypt_block)(const void *, const uint8_t *, uint8_t *));

374 extern void gcm_mul(uint64_t *, uint64_t *, uint64_t *);

401 extern void crypto_init_ptrs(crypto_data_t *, void **, offset_t *);
377 extern void crypto_get_ptrs(crypto_data_t *, void **, offset_t *,
378     uint8_t **, size_t *, uint8_t **, size_t);

403 extern void *ecb_alloc_ctx(int);
404 extern void *cbc_alloc_ctx(int);
405 extern void *ctr_alloc_ctx(int);
406 extern void *ccm_alloc_ctx(int);
407 extern void *gcm_alloc_ctx(int);
408 extern void *gmac_alloc_ctx(int);
409 extern void crypto_free_mode_ctx(void *);
410 extern void gcm_set_kmflag(gcm_ctx_t *, int);

412 #ifdef  INLINE_CRYPTO_GET_PTRS
413 /*
414  * Get pointers for where in the output to copy a block of encrypted or
415  * decrypted data.  The iov_or_mp argument stores a pointer to the current
416  * iovec or mp, and offset stores an offset into the current iovec or mp.
417  */
418 static inline void
419 crypto_get_ptrs(crypto_data_t *out, void **iov_or_mp, offset_t *current_offset,
420     uint8_t **out_data_1, size_t *out_data_1_len, uint8_t **out_data_2,
421     size_t amt)
422 {
423         offset_t offset;

425         switch (out->cd_format) {
```



```
426         case CRYPTO_DATA_RAW: {
427                 iovec_t *iov;

429                 offset = *current_offset;
430                 iov = &out->cd_raw;
431                 if ((offset + amt) <= iov->iov_len) {
432                         /* one block fits */
433                         *out_data_1 = (uint8_t *)iov->iov_base + offset;
434                         *out_data_1_len = amt;
435                         *out_data_2 = NULL;
436                         *current_offset = offset + amt;
437                 }
438                 break;
439         }

441         case CRYPTO_DATA_UIO: {
442                 uio_t *uio = out->cd_uio;
443                 iovec_t *iov;
444                 offset_t offset;
445                 uintptr_t vec_idx;
446                 uint8_t *p;

448                 offset = *current_offset;
449                 vec_idx = (uintptr_t)(*iov_or_mp);
450                 iov = &uio->uio_iov[vec_idx];
451                 p = (uint8_t *)iov->iov_base + offset;
452                 *out_data_1 = p;

454                 if (offset + amt <= iov->iov_len) {
455                         /* can fit one block into this iov */
456                         *out_data_1_len = amt;
457                         *out_data_2 = NULL;
458                         *current_offset = offset + amt;
459                 } else {
460                         /* one block spans two iovecs */
461                         *out_data_1_len = iov->iov_len - offset;
462                         if (vec_idx == uio->uio_iovcnt)
463                                 return;
464                         vec_idx++;
465                         iov = &uio->uio_iov[vec_idx];
466                         *out_data_2 = (uint8_t *)iov->iov_base;
467                         *current_offset = amt - *out_data_1_len;
468                 }
469                 *iov_or_mp = (void *)vec_idx;
470                 break;
471         }

473         case CRYPTO_DATA_MBLK: {
474                 mblk_t *mp;
475                 uint8_t *p;

477                 offset = *current_offset;
478                 mp = (mblk_t *)*iov_or_mp;
479                 p = mp->b_rptr + offset;
480                 *out_data_1 = p;
481                 if ((p + amt) <= mp->b_wptr) {
482                         /* can fit one block into this mblk */
483                         *out_data_1_len = amt;
484                         *out_data_2 = NULL;
485                         *current_offset = offset + amt;
486                 } else {
487                         /* one block spans two mblks */
488                         *out_data_1_len = _PTRDIFF(mp->b_wptr, p);
489                         if ((mp = mp->b_cont) == NULL)
490                                 return;
491                         *out_data_2 = mp->b_rptr;
```

```
492                         *current_offset = (amt - *out_data_1_len);
493                 }
494                 *iov_or_mp = mp;
495                 break;
496         }
497         } /* end switch */
498 }
499 #endif  /* INLINE_CRYPTO_GET_PTRS */

501 /*
502  * Checks whether a crypto_data_t object is composed of a single contiguous
503  * buffer. This is used in all fastpath detection code to avoid the
504  * possibility of having to do partial block splicing.
505  */
506 #define CRYPTO_DATA_IS_SINGLE_BLOCK(cd) \
507         (cd != NULL && (cd->cd_format == CRYPTO_DATA_RAW || \
508         (cd->cd_format == CRYPTO_DATA_UIO && cd->cd_uio->uio_iovcnt == 1) || \
509         (cd->cd_format == CRYPTO_DATA_MBLK && cd->cd_mp->b_next == NULL)))

511 /*
512  * Returns the first contiguous data buffer in a crypto_data_t object.
513  */
514 #define CRYPTO_DATA_FIRST_BLOCK(cd) \
515         (cd->cd_format == CRYPTO_DATA_RAW ? \
516         (void *)(cd->cd_raw.iov_base + cd->cd_offset) : \
517         (cd->cd_format == CRYPTO_DATA_UIO ? \
518         (void *)(cd->cd_uio->uio_iov[0].iov_base + cd->cd_offset) : \
519         (void *)(cd->cd_mp->b_rptr + cd->cd_offset)))

521 #ifdef  __cplusplus
522 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    2843 Thu Apr 30 20:52:31 2015
new/usr/src/lib/pkcs11/libsoftcrypto/amd64/Makefile
4896 Performance improvements for KCF AES modes
**********************************************************
#
# 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 (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2015 by Saso Kiselkov. All rights reserved.
#

LIBRARY =       libsoftcrypto.a
VERS =          .1

include         ../Makefile.com

AES_PSM_OBJS =  aes_amd64.o aes_intel.o aeskey.o
ARCFOUR_PSM_OBJS = arcfour-x86_64.o
BIGNUM_PSM_OBJS = bignum_amd64.o bignum_amd64_asm.o

include         $(SRC)/lib/Makefile.lib
include         $(SRC)/lib/Makefile.lib.64

CERRWARN +=     -_gcc=-Wno-type-limits
CERRWARN +=     -_gcc=-Wno-parentheses
CERRWARN +=     -_gcc=-Wno-uninitialized
C99MODE =       $(C99_ENABLE)

AES_PSM_SRC =   $(AES_DIR)/$(MACH64)/aes_amd64.s \
                $(AES_DIR)/$(MACH64)/aes_intel.s \
                $(AES_DIR)/$(MACH64)/aeskey.c
ARCFOUR_PSM_SRC = arcfour-x86_64.s
BIGNUM_PSM_SRC = $(BIGNUM_DIR)/$(MACH64)/bignum_amd64.c \
                $(BIGNUM_DIR)/$(MACH64)/bignum_amd64_asm.s

# Sources need to be redefined after Makefile.lib inclusion.
SRCS =          $(AES_SRC) $(ARCFOUR_SRC) $(BIGNUM_SRC) $(BLOWFISH_SRC) \
                $(DES_SRC) $(MODES_SRC) $(DH_SRC) $(DSA_SRC) $(RSA_SRC) \
                $(PAD_SRC)

SRCDIR =        $(SRC)/lib/pkcs11/libsoftcrypto/common

LIBS =          $(DYNLIB) $(LINTLIB)
MAPFILEDIR =    ../common

CFLAGS +=       -xO4 -xcrossfile
```



```
CPPFLAGS +=     -I$(CRYPTODIR) -I$(UTSDIR) -D_POSIX_PTHREAD_SEMANTICS
ASFLAGS +=      $(AS_PICFLAGS) -P -D__STDC__ -D_ASM
BIGNUM_FLAGS += -DPSR_MUL
CLEANFILES +=   arcfour-x86_64.s
LDLIBS +=       -lcryptoutil -lc
LINTFLAGS64 +=  $(EXTRA_LINT_FLAGS)

.KEEP_STATE:

all:            $(LIBS)

lint:           $(SRCS)

install:        all $(ROOTLIBS64) $(ROOTLINKS64) $(ROOTLINT64)

pics/%.o: $(AES_DIR)/$(MACH64)/%.c
                $(COMPILE.c) $(AES_FLAGS) -o $@ $<
                $(POST_PROCESS_O)

pics/%.o: $(AES_DIR)/$(MACH64)/%.s
                $(COMPILE.s) $(AES_FLAGS) -o $@ $<
                $(POST_PROCESS_O)

pics/%.o: $(BIGNUM_DIR)/$(MACH64)/%.c
                $(COMPILE.c) $(BIGNUM_FLAGS) -o $@ $<
                $(POST_PROCESS_O)

pics/%.o: $(BIGNUM_DIR)/$(MACH64)/%.s
                $(COMPILE64.s) $(BIGNUM_FLAGS) -o $@ $<
                $(POST_PROCESS_O)

include         ../Makefile.targ

arcfour-x86_64.s: $(ARCFOUR_DIR)/amd64/arcfour-x86_64.pl
                $(PERL) $? $@

pics/%.o: arcfour-x86_64.s
                $(COMPILE64.s) $(ARCFOUR_FLAGS) -o $@ $<
                $(POST_PROCESS_O)
```

```
**********************************************************
    1837 Thu Apr 30 20:52:31 2015
new/usr/src/lib/pkcs11/libsoftcrypto/i386/Makefile
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
   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 #

  22 #
  23 # Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
  24 # Copyright 2015 by Saso Kiselkov. All rights reserved.
  25 #

  27 LIBRARY =       libsoftcrypto.a
  28 VERS =          .1

  30 include         ../Makefile.com

  32 include         $(SRC)/lib/Makefile.lib

  34 CERRWARN +=     -_gcc=-Wno-type-limits
  35 CERRWARN +=     -_gcc=-Wno-parentheses
  36 CERRWARN +=     -_gcc=-Wno-uninitialized
  37 C99MODE =       $(C99_ENABLE)

  39 # Sources need to be redefined after Makefile.lib inclusion.
  40 SRCS =          $(AES_SRC) $(ARCFOUR_SRC) $(BIGNUM_SRC) $(BLOWFISH_SRC) \
  41                 $(DES_SRC) $(MODES_SRC) $(DH_SRC) $(DSA_SRC) $(RSA_SRC) \
  42                 $(PAD_SRC)

  44 SRCDIR =        $(SRC)/lib/pkcs11/libsoftcrypto/common

  46 LIBS =          $(DYNLIB) $(LINTLIB)
  47 MAPFILEDIR =    ../common

  49 CFLAGS +=       $(CCVERBOSE)
  50 CPPFLAGS +=     -I$(CRYPTODIR) -I$(UTSDIR) -D_POSIX_PTHREAD_SEMANTICS
  51 LDLIBS +=       -lcryptoutil -lc
  52 LINTFLAGS +=    $(EXTRA_LINT_FLAGS)

  54 # Symbol capabilities objects are i386.
  55 EXTPICS =       ../capabilities/intel/i386/pics/symcap.o

  57 .KEEP_STATE:

  59 all:            $(LIBS)

  61 lint:           $(SRCS)
```



```
  63 install:        all $(ROOTLIBS) $(ROOTLINKS) $(ROOTLINT)

  65 include         ../Makefile.targ
```

```
**********************************************************
   13382 Thu Apr 30 20:52:32 2015
new/usr/src/uts/common/crypto/core/kcf_prov_lib.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#include <sys/strsun.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/kmem.h>
#include <sys/md5.h>
#include <sys/sha1.h>
#include <sys/sha2.h>
#include <modes/modes.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>

/*
 * Utility routine to apply the command, 'cmd', to the
 * data in the uio structure.
 */
int
crypto_uio_data(crypto_data_t *data, uchar_t *buf, int len, cmd_type_t cmd,
    void *digest_ctx, void (*update)())
{
        uio_t *uiop = data->cd_uio;
        off_t offset = data->cd_offset;
        size_t length = len;
        uint_t vec_idx;
        size_t cur_len;
        uchar_t *datap;

        ASSERT(data->cd_format == CRYPTO_DATA_UIO);
        if (uiop->uio_segflg != UIO_SYSSPACE) {
                return (CRYPTO_ARGUMENTS_BAD);
        }

        /*
         * Jump to the first iovec containing data to be
```



```
         * processed.
         */
        for (vec_idx = 0; vec_idx < uiop->uio_iovcnt &&
            offset >= uiop->uio_iov[vec_idx].iov_len;
            offset -= uiop->uio_iov[vec_idx++].iov_len)
                ;

        if (vec_idx == uiop->uio_iovcnt) {
                /*
                 * The caller specified an offset that is larger than
                 * the total size of the buffers it provided.
                 */
                return (CRYPTO_DATA_LEN_RANGE);
        }

        while (vec_idx < uiop->uio_iovcnt && length > 0) {
                cur_len = MIN(uiop->uio_iov[vec_idx].iov_len -
                    offset, length);

                datap = (uchar_t *)(uiop->uio_iov[vec_idx].iov_base +
                    offset);
                switch (cmd) {
                case COPY_FROM_DATA:
                        bcopy(datap, buf, cur_len);
                        buf += cur_len;
                        break;
                case COPY_TO_DATA:
                        bcopy(buf, datap, cur_len);
                        buf += cur_len;
                        break;
                case COMPARE_TO_DATA:
                        if (bcmp(datap, buf, cur_len))
                                return (CRYPTO_SIGNATURE_INVALID);
                        buf += cur_len;
                        break;
                case MD5_DIGEST_DATA:
                case SHA1_DIGEST_DATA:
                case SHA2_DIGEST_DATA:
                case GHASH_DATA:
                        update(digest_ctx, datap, cur_len);
                        break;
                }

                length -= cur_len;
                vec_idx++;
                offset = 0;
        }

        if (vec_idx == uiop->uio_iovcnt && length > 0) {
                /*
                 * The end of the specified iovec's was reached but
                 * the length requested could not be processed.
                 */
                switch (cmd) {
                case COPY_TO_DATA:
                        data->cd_length = len;
                        return (CRYPTO_BUFFER_TOO_SMALL);
                default:
                        return (CRYPTO_DATA_LEN_RANGE);
                }
        }

        return (CRYPTO_SUCCESS);
}
```

_____*unchanged_portion_omitted_*

```
399 int
400 crypto_update_iov(void *ctx, crypto_data_t *input, crypto_data_t *output,
401     int (*cipher)(void *, caddr_t, size_t, crypto_data_t *),
402     void (*copy_block)(const uint8_t *, uint64_t *))
399     void (*copy_block)(uint8_t *, uint64_t *))
403 {
404         common_ctx_t *common_ctx = ctx;
405         int rv;

407         if (input->cd_miscdata != NULL) {
408                 copy_block((uint8_t *)input->cd_miscdata,
409                     &common_ctx->cc_iv[0]);
410         }

412         if (input->cd_raw.iov_len < input->cd_length)
413                 return (CRYPTO_ARGUMENTS_BAD);

415         rv = (cipher)(ctx, input->cd_raw.iov_base + input->cd_offset,
416             input->cd_length, (input == output) ? NULL : output);

418         return (rv);
419 }

421 int
422 crypto_update_uio(void *ctx, crypto_data_t *input, crypto_data_t *output,
423     int (*cipher)(void *, caddr_t, size_t, crypto_data_t *),
424     void (*copy_block)(const uint8_t *, uint64_t *))
421     void (*copy_block)(uint8_t *, uint64_t *))
425 {
426         common_ctx_t *common_ctx = ctx;
427         uio_t *uiop = input->cd_uio;
428         off_t offset = input->cd_offset;
429         size_t length = input->cd_length;
430         uint_t vec_idx;
431         size_t cur_len;

433         if (input->cd_miscdata != NULL) {
434                 copy_block((uint8_t *)input->cd_miscdata,
435                     &common_ctx->cc_iv[0]);
436         }

438         if (input->cd_uio->uio_segflg != UIO_SYSSPACE) {
439                 return (CRYPTO_ARGUMENTS_BAD);
440         }

442         /*
443          * Jump to the first iovec containing data to be
444          * processed.
445          */
446         for (vec_idx = 0; vec_idx < uiop->uio_iovcnt &&
447             offset >= uiop->uio_iov[vec_idx].iov_len;
448             offset -= uiop->uio_iov[vec_idx++].iov_len)
449                 ;
450         if (vec_idx == uiop->uio_iovcnt) {
451                 /*
452                  * The caller specified an offset that is larger than the
453                  * total size of the buffers it provided.
454                  */
455                 return (CRYPTO_DATA_LEN_RANGE);
456         }

458         /*
459          * Now process the iovecs.
460          */
461         while (vec_idx < uiop->uio_iovcnt && length > 0) {
462                 cur_len = MIN(uiop->uio_iov[vec_idx].iov_len -
```



```
463                     offset, length);

465                 (cipher)(ctx, uiop->uio_iov[vec_idx].iov_base + offset,
466                     cur_len, (input == output) ? NULL : output);

468                 length -= cur_len;
469                 vec_idx++;
470                 offset = 0;
471         }

473         if (vec_idx == uiop->uio_iovcnt && length > 0) {
474                 /*
475                  * The end of the specified iovec's was reached but
476                  * the length requested could not be processed, i.e.
477                  * The caller requested to digest more data than it provided.
478                  */

480                 return (CRYPTO_DATA_LEN_RANGE);
481         }

483         return (CRYPTO_SUCCESS);
484 }

486 int
487 crypto_update_mp(void *ctx, crypto_data_t *input, crypto_data_t *output,
488     int (*cipher)(void *, caddr_t, size_t, crypto_data_t *),
489     void (*copy_block)(const uint8_t *, uint64_t *))
486     void (*copy_block)(uint8_t *, uint64_t *))
490 {
491         common_ctx_t *common_ctx = ctx;
492         off_t offset = input->cd_offset;
493         size_t length = input->cd_length;
494         mblk_t *mp;
495         size_t cur_len;

497         if (input->cd_miscdata != NULL) {
498                 copy_block((uint8_t *)input->cd_miscdata,
499                     &common_ctx->cc_iv[0]);
500         }

502         /*
503          * Jump to the first mblk_t containing data to be processed.
504          */
505         for (mp = input->cd_mp; mp != NULL && offset >= MBLKL(mp);
506             offset -= MBLKL(mp), mp = mp->b_cont)
507                 ;
508         if (mp == NULL) {
509                 /*
510                  * The caller specified an offset that is larger than the
511                  * total size of the buffers it provided.
512                  */
513                 return (CRYPTO_DATA_LEN_RANGE);
514         }

516         /*
517          * Now do the processing on the mblk chain.
518          */
519         while (mp != NULL && length > 0) {
520                 cur_len = MIN(MBLKL(mp) - offset, length);
521                 (cipher)(ctx, (char *)(mp->b_rptr + offset), cur_len,
522                     (input == output) ? NULL : output);

524                 length -= cur_len;
525                 offset = 0;
526                 mp = mp->b_cont;
527         }
```

```
529         if (mp == NULL && length > 0) {
530                 /*
531                  * The end of the mblk was reached but the length requested
532                  * could not be processed, i.e. The caller requested
533                  * to digest more data than it provided.
534                  */
535                 return (CRYPTO_DATA_LEN_RANGE);
536         }

538         return (CRYPTO_SUCCESS);
539 }
```

unchanged_portion_omitted_

```
**********************************************************
   40706 Thu Apr 30 20:52:32 2015
new/usr/src/uts/common/crypto/io/aes.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

/*
 * AES provider for the Kernel Cryptographic Framework (KCF)
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/modctl.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/crypto/common.h>
#include <sys/crypto/impl.h>
#include <sys/crypto/spi.h>
#include <sys/sysmacros.h>
#include <sys/strsun.h>
#include <modes/modes.h>
#define _AES_IMPL
#include <aes/aes_impl.h>

extern struct mod_ops mod_cryptoops;

/*
 * Module linkage information for the kernel.
 */
static struct modlcrypto modlcrypto = {
        &mod_cryptoops,
        "AES Kernel SW Provider"
};
```

*_____unchanged_portion_omitted_*

```
static void
aes_copy_block64(const uint8_t *in, uint64_t *out)
aes_copy_block64(uint8_t *in, uint64_t *out)
{
        if (IS_P2ALIGNED(in, sizeof (uint64_t))) {
                /* LINTED: pointer alignment */
                out[0] = *(uint64_t *)&in[0];
```



```
                /* LINTED: pointer alignment */
                out[1] = *(uint64_t *)&in[8];
        } else {
                uint8_t *iv8 = (uint8_t *)&out[0];

                AES_COPY_BLOCK_UNALIGNED(in, iv8);
                AES_COPY_BLOCK(in, iv8);
        }
}


static int
aes_encrypt(crypto_ctx_t *ctx, crypto_data_t *plaintext,
    crypto_data_t *ciphertext, crypto_req_handle_t req)
{
        int ret = CRYPTO_FAILED;

        aes_ctx_t *aes_ctx;
        size_t saved_length, saved_offset, length_needed;

        ASSERT(ctx->cc_provider_private != NULL);
        aes_ctx = ctx->cc_provider_private;

        /*
         * For block ciphers, plaintext must be a multiple of AES block size.
         * This test is only valid for ciphers whose blocksize is a power of 2.
         */
        if (((aes_ctx->ac_flags & (CTR_MODE|CCM_MODE|GCM_MODE|GMAC_MODE))
            == 0) && (plaintext->cd_length & (AES_BLOCK_LEN - 1)) != 0)
                return (CRYPTO_DATA_LEN_RANGE);

        AES_ARG_INPLACE(plaintext, ciphertext);

        /*
         * We need to just return the length needed to store the output.
         * We should not destroy the context for the following case.
         */
        switch (aes_ctx->ac_flags & (CCM_MODE|GCM_MODE|GMAC_MODE)) {
        case CCM_MODE:
                length_needed = plaintext->cd_length + aes_ctx->ac_mac_len;
                break;
        case GCM_MODE:
                length_needed = plaintext->cd_length + aes_ctx->ac_tag_len;
                break;
        case GMAC_MODE:
                if (plaintext->cd_length != 0)
                        return (CRYPTO_ARGUMENTS_BAD);

                length_needed = aes_ctx->ac_tag_len;
                break;
        default:
                length_needed = plaintext->cd_length;
        }

        if (ciphertext->cd_length < length_needed) {
                ciphertext->cd_length = length_needed;
                return (CRYPTO_BUFFER_TOO_SMALL);
        }

        saved_length = ciphertext->cd_length;
        saved_offset = ciphertext->cd_offset;

        /*
         * Do an update on the specified input data.
         */
        ret = aes_encrypt_update(ctx, plaintext, ciphertext, req);
```

```
460         if (ret != CRYPTO_SUCCESS) {
461                 return (ret);
462         }

464         /*
465          * For CCM mode, aes_ccm_encrypt_final() will take care of any
466          * left-over unprocessed data, and compute the MAC
467          */
468         if (aes_ctx->ac_flags & CCM_MODE) {
469                 /*
470                  * ccm_encrypt_final() will compute the MAC and append
471                  * it to existing ciphertext. So, need to adjust the left over
472                  * length value accordingly
473                  */

475                 /* order of following 2 lines MUST not be reversed */
476                 ciphertext->cd_offset = ciphertext->cd_length;
477                 ciphertext->cd_length = saved_length - ciphertext->cd_length;
478                 ret = ccm_encrypt_final((ccm_ctx_t *)aes_ctx, ciphertext,
479                     AES_BLOCK_LEN, aes_encrypt_block, AES_XOR_BLOCK);
478                     AES_BLOCK_LEN, aes_encrypt_block, aes_xor_block);
480                 if (ret != CRYPTO_SUCCESS) {
481                         return (ret);
482                 }

484                 if (plaintext != ciphertext) {
485                         ciphertext->cd_length =
486                             ciphertext->cd_offset - saved_offset;
487                 }
488                 ciphertext->cd_offset = saved_offset;
489         } else if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE)) {
490                 /*
491                  * gcm_encrypt_final() will compute the MAC and append
492                  * it to existing ciphertext. So, need to adjust the left over
493                  * length value accordingly
494                  */

496                 /* order of following 2 lines MUST not be reversed */
497                 ciphertext->cd_offset = ciphertext->cd_length;
498                 ciphertext->cd_length = saved_length - ciphertext->cd_length;
499                 ret = gcm_encrypt_final((gcm_ctx_t *)aes_ctx, ciphertext,
500                     AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
501                     AES_XOR_BLOCK);
499                     AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
500                     aes_xor_block);
502                 if (ret != CRYPTO_SUCCESS) {
503                         return (ret);
504                 }

506                 if (plaintext != ciphertext) {
507                         ciphertext->cd_length =
508                             ciphertext->cd_offset - saved_offset;
509                 }
510                 ciphertext->cd_offset = saved_offset;
511         }

513         ASSERT(aes_ctx->ac_remainder_len == 0);
514         (void) aes_free_context(ctx);

516         return (ret);
517 }


520 static int
521 aes_decrypt(crypto_ctx_t *ctx, crypto_data_t *ciphertext,
522     crypto_data_t *plaintext, crypto_req_handle_t req)
```



```
523 {
524         int ret = CRYPTO_FAILED;

526         aes_ctx_t *aes_ctx;
527         off_t saved_offset;
528         size_t saved_length, length_needed;

530         ASSERT(ctx->cc_provider_private != NULL);
531         aes_ctx = ctx->cc_provider_private;

533         /*
534          * For block ciphers, plaintext must be a multiple of AES block size.
535          * This test is only valid for ciphers whose blocksize is a power of 2.
536          */
537         if (((aes_ctx->ac_flags & (CTR_MODE|CCM_MODE|GCM_MODE|GMAC_MODE))
538             == 0) && (ciphertext->cd_length & (AES_BLOCK_LEN - 1)) != 0) {
539                 return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
540         }

542         AES_ARG_INPLACE(ciphertext, plaintext);

544         /*
545          * Return length needed to store the output.
546          * Do not destroy context when plaintext buffer is too small.
547          *
548          * CCM:  plaintext is MAC len smaller than cipher text
549          * GCM:  plaintext is TAG len smaller than cipher text
550          * GMAC: plaintext length must be zero
551          */
552         switch (aes_ctx->ac_flags & (CCM_MODE|GCM_MODE|GMAC_MODE)) {
553         case CCM_MODE:
554                 length_needed = aes_ctx->ac_processed_data_len;
555                 break;
556         case GCM_MODE:
557                 length_needed = ciphertext->cd_length - aes_ctx->ac_tag_len;
558                 break;
559         case GMAC_MODE:
560                 if (plaintext->cd_length != 0)
561                         return (CRYPTO_ARGUMENTS_BAD);

563                 length_needed = 0;
564                 break;
565         default:
566                 length_needed = ciphertext->cd_length;
567         }

569         if (plaintext->cd_length < length_needed) {
570                 plaintext->cd_length = length_needed;
571                 return (CRYPTO_BUFFER_TOO_SMALL);
572         }

574         saved_offset = plaintext->cd_offset;
575         saved_length = plaintext->cd_length;

577         /*
578          * Do an update on the specified input data.
579          */
580         ret = aes_decrypt_update(ctx, ciphertext, plaintext, req);
581         if (ret != CRYPTO_SUCCESS) {
582                 goto cleanup;
583         }

585         if (aes_ctx->ac_flags & CCM_MODE) {
586                 ASSERT(aes_ctx->ac_processed_data_len == aes_ctx->ac_data_len);
587                 ASSERT(aes_ctx->ac_processed_mac_len == aes_ctx->ac_mac_len);
```

```
589                 /* order of following 2 lines MUST not be reversed */
590                 plaintext->cd_offset = plaintext->cd_length;
591                 plaintext->cd_length = saved_length - plaintext->cd_length;

593                 ret = ccm_decrypt_final((ccm_ctx_t *)aes_ctx, plaintext,
594                     AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
595                     AES_XOR_BLOCK);
593                     AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
594                     aes_xor_block);
596                 if (ret == CRYPTO_SUCCESS) {
597                         if (plaintext != ciphertext) {
598                                 plaintext->cd_length =
599                                     plaintext->cd_offset - saved_offset;
600                         }
601                 } else {
602                         plaintext->cd_length = saved_length;
603                 }

605                 plaintext->cd_offset = saved_offset;
606         } else if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE)) {
607                 /* order of following 2 lines MUST not be reversed */
608                 plaintext->cd_offset = plaintext->cd_length;
609                 plaintext->cd_length = saved_length - plaintext->cd_length;

611                 ret = gcm_decrypt_final((gcm_ctx_t *)aes_ctx, plaintext,
612                     AES_BLOCK_LEN, aes_encrypt_block, AES_XOR_BLOCK,
613                     AES_COPY_BLOCK, aes_ctr_mode);
611                     AES_BLOCK_LEN, aes_encrypt_block, aes_xor_block);
614                 if (ret == CRYPTO_SUCCESS) {
615                         if (plaintext != ciphertext) {
616                                 plaintext->cd_length =
617                                     plaintext->cd_offset - saved_offset;
618                         }
619                 } else {
620                         plaintext->cd_length = saved_length;
621                 }

623                 plaintext->cd_offset = saved_offset;
624         }

626         ASSERT(aes_ctx->ac_remainder_len == 0);

628 cleanup:
629         (void) aes_free_context(ctx);

631         return (ret);
632 }
```

_____***unchanged_portion_omitted_***

```
711 static int
712 aes_decrypt_update(crypto_ctx_t *ctx, crypto_data_t *ciphertext,
713     crypto_data_t *plaintext, crypto_req_handle_t req)
714 {
715         off_t saved_offset;
716         size_t saved_length, out_len;
717         int ret = CRYPTO_SUCCESS;
718         aes_ctx_t *aes_ctx;

720         ASSERT(ctx->cc_provider_private != NULL);
721         aes_ctx = ctx->cc_provider_private;

723         AES_ARG_INPLACE(ciphertext, plaintext);

725         /*
726          * Compute number of bytes that will hold the plaintext.
```



```
727          * This is not necessary for CCM and GMAC since these
725          * This is not necessary for CCM, GCM, and GMAC since these
728          * mechanisms never return plaintext for update operations.
729          */
730         if ((aes_ctx->ac_flags & (CCM_MODE|GMAC_MODE)) == 0) {
728         if ((aes_ctx->ac_flags & (CCM_MODE|GCM_MODE|GMAC_MODE)) == 0) {
731                 out_len = aes_ctx->ac_remainder_len;
732                 out_len += ciphertext->cd_length;
733                 out_len &= ~(AES_BLOCK_LEN - 1);
734                 if (aes_ctx->ac_flags & GCM_MODE)
735                         out_len -= ((gcm_ctx_t *)aes_ctx)->gcm_tag_len;

737                 /* return length needed to store the output */
738                 if (plaintext->cd_length < out_len) {
739                         plaintext->cd_length = out_len;
740                         return (CRYPTO_BUFFER_TOO_SMALL);
741                 }
742         }

744         saved_offset = plaintext->cd_offset;
745         saved_length = plaintext->cd_length;

747         if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE))
748                 gcm_set_kmflag((gcm_ctx_t *)aes_ctx, crypto_kmflag(req));

750         /*
751          * Do the AES update on the specified input data.
752          */
753         switch (ciphertext->cd_format) {
754         case CRYPTO_DATA_RAW:
755                 ret = crypto_update_iov(ctx->cc_provider_private,
756                     ciphertext, plaintext, aes_decrypt_contiguous_blocks,
757                     aes_copy_block64);
758                 break;
759         case CRYPTO_DATA_UIO:
760                 ret = crypto_update_uio(ctx->cc_provider_private,
761                     ciphertext, plaintext, aes_decrypt_contiguous_blocks,
762                     aes_copy_block64);
763                 break;
764         case CRYPTO_DATA_MBLK:
765                 ret = crypto_update_mp(ctx->cc_provider_private,
766                     ciphertext, plaintext, aes_decrypt_contiguous_blocks,
767                     aes_copy_block64);
768                 break;
769         default:
770                 ret = CRYPTO_ARGUMENTS_BAD;
771         }

773         /*
774          * Since AES counter mode is a stream cipher, we call
775          * ctr_mode_final() to pick up any remaining bytes.
776          * It is an internal function that does not destroy
777          * the context like *normal* final routines.
778          */
779         if ((aes_ctx->ac_flags & CTR_MODE) && (aes_ctx->ac_remainder_len > 0)) {
780                 ret = ctr_mode_final((ctr_ctx_t *)aes_ctx, plaintext,
781                     aes_encrypt_block);
782                 if (ret == CRYPTO_DATA_LEN_RANGE)
783                         ret = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
784         }

786         if (ret == CRYPTO_SUCCESS) {
787                 if (ciphertext != plaintext)
788                         plaintext->cd_length =
789                             plaintext->cd_offset - saved_offset;
790         } else {
```

```
791                 plaintext->cd_length = saved_length;
792         }
793         plaintext->cd_offset = saved_offset;


796         return (ret);
797 }

799 /* ARGSUSED */
800 static int
801 aes_encrypt_final(crypto_ctx_t *ctx, crypto_data_t *data,
802     crypto_req_handle_t req)
803 {
804         aes_ctx_t *aes_ctx;
805         int ret;

807         ASSERT(ctx->cc_provider_private != NULL);
808         aes_ctx = ctx->cc_provider_private;

810         if (data->cd_format != CRYPTO_DATA_RAW &&
811             data->cd_format != CRYPTO_DATA_UIO &&
812             data->cd_format != CRYPTO_DATA_MBLK) {
813                 return (CRYPTO_ARGUMENTS_BAD);
814         }

816         if (aes_ctx->ac_flags & CTR_MODE) {
817                 if (aes_ctx->ac_remainder_len > 0) {
818                         ret = ctr_mode_final((ctr_ctx_t *)aes_ctx, data,
819                             aes_encrypt_block);
820                         if (ret != CRYPTO_SUCCESS)
821                                 return (ret);
822                 }
823         } else if (aes_ctx->ac_flags & CCM_MODE) {
824                 ret = ccm_encrypt_final((ccm_ctx_t *)aes_ctx, data,
825                     AES_BLOCK_LEN, aes_encrypt_block, AES_XOR_BLOCK);
821                     AES_BLOCK_LEN, aes_encrypt_block, aes_xor_block);
826                 if (ret != CRYPTO_SUCCESS) {
827                         return (ret);
828                 }
829         } else if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE)) {
830                 size_t saved_offset = data->cd_offset;

832                 ret = gcm_encrypt_final((gcm_ctx_t *)aes_ctx, data,
833                     AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
834                     AES_XOR_BLOCK);
829                     AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
830                     aes_xor_block);
835                 if (ret != CRYPTO_SUCCESS) {
836                         return (ret);
837                 }
838                 data->cd_length = data->cd_offset - saved_offset;
839                 data->cd_offset = saved_offset;
840         } else {
841                 /*
842                  * There must be no unprocessed plaintext.
843                  * This happens if the length of the last data is
844                  * not a multiple of the AES block length.
845                  */
846                 if (aes_ctx->ac_remainder_len > 0) {
847                         return (CRYPTO_DATA_LEN_RANGE);
848                 }
849                 data->cd_length = 0;
850         }

852         (void) aes_free_context(ctx);
```



```
854         return (CRYPTO_SUCCESS);
855 }

857 /* ARGSUSED */
858 static int
859 aes_decrypt_final(crypto_ctx_t *ctx, crypto_data_t *data,
860     crypto_req_handle_t req)
861 {
862         aes_ctx_t *aes_ctx;
863         int ret;
864         off_t saved_offset;
865         size_t saved_length;

867         ASSERT(ctx->cc_provider_private != NULL);
868         aes_ctx = ctx->cc_provider_private;

870         if (data->cd_format != CRYPTO_DATA_RAW &&
871             data->cd_format != CRYPTO_DATA_UIO &&
872             data->cd_format != CRYPTO_DATA_MBLK) {
873                 return (CRYPTO_ARGUMENTS_BAD);
874         }

876         /*
877          * There must be no unprocessed ciphertext.
878          * This happens if the length of the last ciphertext is
879          * not a multiple of the AES block length.
880          */
881         if (aes_ctx->ac_remainder_len > 0) {
882                 if ((aes_ctx->ac_flags & CTR_MODE) == 0)
883                         return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
884                 else {
885                         ret = ctr_mode_final((ctr_ctx_t *)aes_ctx, data,
886                             aes_encrypt_block);
887                         if (ret == CRYPTO_DATA_LEN_RANGE)
888                                 ret = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
889                         if (ret != CRYPTO_SUCCESS)
890                                 return (ret);
891                 }
892         }

894         if (aes_ctx->ac_flags & CCM_MODE) {
895                 /*
896                  * This is where all the plaintext is returned, make sure
897                  * the plaintext buffer is big enough
898                  */
899                 size_t pt_len = aes_ctx->ac_data_len;
900                 if (data->cd_length < pt_len) {
901                         data->cd_length = pt_len;
902                         return (CRYPTO_BUFFER_TOO_SMALL);
903                 }

905                 ASSERT(aes_ctx->ac_processed_data_len == pt_len);
906                 ASSERT(aes_ctx->ac_processed_mac_len == aes_ctx->ac_mac_len);
907                 saved_offset = data->cd_offset;
908                 saved_length = data->cd_length;
909                 ret = ccm_decrypt_final((ccm_ctx_t *)aes_ctx, data,
910                     AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
911                     AES_XOR_BLOCK);
906                     AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
907                     aes_xor_block);
912                 if (ret == CRYPTO_SUCCESS) {
913                         data->cd_length = data->cd_offset - saved_offset;
914                 } else {
915                         data->cd_length = saved_length;
916                 }
```

```
                data->cd_offset = saved_offset;
                if (ret != CRYPTO_SUCCESS) {
                        return (ret);
                }
        } else if (aes_ctx->ac_flags & (GCM_MODE|GMAC_MODE)) {
                /*
                 * Check to make sure there is enough space for remaining
                 * plaintext.
                 * This is where all the plaintext is returned, make sure
                 * the plaintext buffer is big enough
                 */
                gcm_ctx_t *ctx = (gcm_ctx_t *)aes_ctx;
                size_t pt_len = ctx->gcm_last_input_fill - ctx->gcm_tag_len;
                size_t pt_len = ctx->gcm_processed_data_len - ctx->gcm_tag_len;

                if (data->cd_length < pt_len) {
                        data->cd_length = pt_len;
                        return (CRYPTO_BUFFER_TOO_SMALL);
                }

                saved_offset = data->cd_offset;
                saved_length = data->cd_length;
                ret = gcm_decrypt_final((gcm_ctx_t *)aes_ctx, data,
                    AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
                    AES_XOR_BLOCK, aes_ctr_mode);
                    AES_BLOCK_LEN, aes_encrypt_block, aes_xor_block);
                if (ret == CRYPTO_SUCCESS) {
                        data->cd_length = data->cd_offset - saved_offset;
                } else {
                        data->cd_length = saved_length;
                }

                data->cd_offset = saved_offset;
                if (ret != CRYPTO_SUCCESS) {
                        return (ret);
                }
        }


        if ((aes_ctx->ac_flags & (CTR_MODE|CCM_MODE|GCM_MODE|GMAC_MODE)) == 0) {
                data->cd_length = 0;
        }

        (void) aes_free_context(ctx);

        return (CRYPTO_SUCCESS);
}

/* ARGSUSED */
static int
aes_encrypt_atomic(crypto_provider_handle_t provider,
    crypto_session_id_t session_id, crypto_mechanism_t *mechanism,
    crypto_key_t *key, crypto_data_t *plaintext, crypto_data_t *ciphertext,
    crypto_spi_ctx_template_t template, crypto_req_handle_t req)
{
        aes_ctx_t aes_ctx;      /* on the stack */
        off_t saved_offset;
        size_t saved_length;
        size_t length_needed;
        int ret;

        AES_ARG_INPLACE(plaintext, ciphertext);

        /*
         * CTR, CCM, GCM, and GMAC modes do not require that plaintext
         * be a multiple of AES block size.
```



```
         */
        switch (mechanism->cm_type) {
        case AES_CTR_MECH_INFO_TYPE:
        case AES_CCM_MECH_INFO_TYPE:
        case AES_GCM_MECH_INFO_TYPE:
        case AES_GMAC_MECH_INFO_TYPE:
                break;
        default:
                if ((plaintext->cd_length & (AES_BLOCK_LEN - 1)) != 0)
                        return (CRYPTO_DATA_LEN_RANGE);
        }

        if ((ret = aes_check_mech_param(mechanism, NULL, 0)) != CRYPTO_SUCCESS)
                return (ret);

        bzero(&aes_ctx, sizeof (aes_ctx_t));

        ret = aes_common_init_ctx(&aes_ctx, template, mechanism, key,
            crypto_kmflag(req), B_TRUE);
        if (ret != CRYPTO_SUCCESS)
                return (ret);

        switch (mechanism->cm_type) {
        case AES_CCM_MECH_INFO_TYPE:
                length_needed = plaintext->cd_length + aes_ctx.ac_mac_len;
                break;
        case AES_GMAC_MECH_INFO_TYPE:
                if (plaintext->cd_length != 0)
                        return (CRYPTO_ARGUMENTS_BAD);
                /* FALLTHRU */
        case AES_GCM_MECH_INFO_TYPE:
                length_needed = plaintext->cd_length + aes_ctx.ac_tag_len;
                break;
        default:
                length_needed = plaintext->cd_length;
        }

        /* return size of buffer needed to store output */
        if (ciphertext->cd_length < length_needed) {
                ciphertext->cd_length = length_needed;
                ret = CRYPTO_BUFFER_TOO_SMALL;
                goto out;
        }

        saved_offset = ciphertext->cd_offset;
        saved_length = ciphertext->cd_length;

        /*
         * Do an update on the specified input data.
         */
        switch (plaintext->cd_format) {
        case CRYPTO_DATA_RAW:
                ret = crypto_update_iov(&aes_ctx, plaintext, ciphertext,
                    aes_encrypt_contiguous_blocks, aes_copy_block64);
                break;
        case CRYPTO_DATA_UIO:
                ret = crypto_update_uio(&aes_ctx, plaintext, ciphertext,
                    aes_encrypt_contiguous_blocks, aes_copy_block64);
                break;
        case CRYPTO_DATA_MBLK:
                ret = crypto_update_mp(&aes_ctx, plaintext, ciphertext,
                    aes_encrypt_contiguous_blocks, aes_copy_block64);
                break;
        default:
                ret = CRYPTO_ARGUMENTS_BAD;
        }
```

```
1045         if (ret == CRYPTO_SUCCESS) {
1046                 if (mechanism->cm_type == AES_CCM_MECH_INFO_TYPE) {
1047                         ret = ccm_encrypt_final((ccm_ctx_t *)&aes_ctx,
1048                             ciphertext, AES_BLOCK_LEN, aes_encrypt_block,
1049                             AES_XOR_BLOCK);
1046                             aes_xor_block);
1050                         if (ret != CRYPTO_SUCCESS)
1051                                 goto out;
1052                         ASSERT(aes_ctx.ac_remainder_len == 0);
1053                 } else if (mechanism->cm_type == AES_GCM_MECH_INFO_TYPE ||
1054                     mechanism->cm_type == AES_GMAC_MECH_INFO_TYPE) {
1055                         ret = gcm_encrypt_final((gcm_ctx_t *)&aes_ctx,
1056                             ciphertext, AES_BLOCK_LEN, aes_encrypt_block,
1057                             AES_COPY_BLOCK, AES_XOR_BLOCK);
1054                             aes_copy_block, aes_xor_block);
1058                         if (ret != CRYPTO_SUCCESS)
1059                                 goto out;
1060                         ASSERT(aes_ctx.ac_remainder_len == 0);
1061                 } else if (mechanism->cm_type == AES_CTR_MECH_INFO_TYPE) {
1062                         if (aes_ctx.ac_remainder_len > 0) {
1063                                 ret = ctr_mode_final((ctr_ctx_t *)&aes_ctx,
1064                                     ciphertext, aes_encrypt_block);
1065                                 if (ret != CRYPTO_SUCCESS)
1066                                         goto out;
1067                         }
1068                 } else {
1069                         ASSERT(aes_ctx.ac_remainder_len == 0);
1070                 }

1072                 if (plaintext != ciphertext) {
1073                         ciphertext->cd_length =
1074                             ciphertext->cd_offset - saved_offset;
1075                 }
1076         } else {
1077                 ciphertext->cd_length = saved_length;
1078         }
1079         ciphertext->cd_offset = saved_offset;

1081 out:
1082         if (aes_ctx.ac_flags & PROVIDER_OWNS_KEY_SCHEDULE) {
1083                 bzero(aes_ctx.ac_keysched, aes_ctx.ac_keysched_len);
1084                 kmem_free(aes_ctx.ac_keysched, aes_ctx.ac_keysched_len);
1085         }

1087         return (ret);
1088 }

1090 /* ARGSUSED */
1091 static int
1092 aes_decrypt_atomic(crypto_provider_handle_t provider,
1093     crypto_session_id_t session_id, crypto_mechanism_t *mechanism,
1094     crypto_key_t *key, crypto_data_t *ciphertext, crypto_data_t *plaintext,
1095     crypto_spi_ctx_template_t template, crypto_req_handle_t req)
1096 {
1097         aes_ctx_t aes_ctx;      /* on the stack */
1098         off_t saved_offset;
1099         size_t saved_length;
1100         size_t length_needed;
1101         int ret;

1103         AES_ARG_INPLACE(ciphertext, plaintext);

1105         /*
1106          * CCM, GCM, CTR, and GMAC modes do not require that ciphertext
1107          * be a multiple of AES block size.
```



```
1108          */
1109         switch (mechanism->cm_type) {
1110         case AES_CTR_MECH_INFO_TYPE:
1111         case AES_CCM_MECH_INFO_TYPE:
1112         case AES_GCM_MECH_INFO_TYPE:
1113         case AES_GMAC_MECH_INFO_TYPE:
1114                 break;
1115         default:
1116                 if ((ciphertext->cd_length & (AES_BLOCK_LEN - 1)) != 0)
1117                         return (CRYPTO_ENCRYPTED_DATA_LEN_RANGE);
1118         }

1120         if ((ret = aes_check_mech_param(mechanism, NULL, 0)) != CRYPTO_SUCCESS)
1121                 return (ret);

1123         bzero(&aes_ctx, sizeof (aes_ctx_t));

1125         ret = aes_common_init_ctx(&aes_ctx, template, mechanism, key,
1126             crypto_kmflag(req), B_FALSE);
1127         if (ret != CRYPTO_SUCCESS)
1128                 return (ret);

1130         switch (mechanism->cm_type) {
1131         case AES_CCM_MECH_INFO_TYPE:
1132                 length_needed = aes_ctx.ac_data_len;
1133                 break;
1134         case AES_GCM_MECH_INFO_TYPE:
1135                 length_needed = ciphertext->cd_length - aes_ctx.ac_tag_len;
1136                 break;
1137         case AES_GMAC_MECH_INFO_TYPE:
1138                 if (plaintext->cd_length != 0)
1139                         return (CRYPTO_ARGUMENTS_BAD);
1140                 length_needed = 0;
1141                 break;
1142         default:
1143                 length_needed = ciphertext->cd_length;
1144         }

1146         /* return size of buffer needed to store output */
1147         if (plaintext->cd_length < length_needed) {
1148                 plaintext->cd_length = length_needed;
1149                 ret = CRYPTO_BUFFER_TOO_SMALL;
1150                 goto out;
1151         }

1153         saved_offset = plaintext->cd_offset;
1154         saved_length = plaintext->cd_length;

1156         if (mechanism->cm_type == AES_GCM_MECH_INFO_TYPE ||
1157             mechanism->cm_type == AES_GMAC_MECH_INFO_TYPE)
1158                 gcm_set_kmflag((gcm_ctx_t *)&aes_ctx, crypto_kmflag(req));

1160         /*
1161          * Do an update on the specified input data.
1162          */
1163         switch (ciphertext->cd_format) {
1164         case CRYPTO_DATA_RAW:
1165                 ret = crypto_update_iov(&aes_ctx, ciphertext, plaintext,
1166                     aes_decrypt_contiguous_blocks, aes_copy_block64);
1167                 break;
1168         case CRYPTO_DATA_UIO:
1169                 ret = crypto_update_uio(&aes_ctx, ciphertext, plaintext,
1170                     aes_decrypt_contiguous_blocks, aes_copy_block64);
1171                 break;
1172         case CRYPTO_DATA_MBLK:
1173                 ret = crypto_update_mp(&aes_ctx, ciphertext, plaintext,
```

```
1174                    aes_decrypt_contiguous_blocks, aes_copy_block64);
1175                break;
1176        default:
1177                ret = CRYPTO_ARGUMENTS_BAD;
1178        }

1180        if (ret == CRYPTO_SUCCESS) {
1181                if (mechanism->cm_type == AES_CCM_MECH_INFO_TYPE) {
1182                        ASSERT(aes_ctx.ac_processed_data_len
1183                            == aes_ctx.ac_data_len);
1184                        ASSERT(aes_ctx.ac_processed_mac_len
1185                            == aes_ctx.ac_mac_len);
1186                        ret = ccm_decrypt_final((ccm_ctx_t *)&aes_ctx,
1187                            plaintext, AES_BLOCK_LEN, aes_encrypt_block,
1188                            AES_COPY_BLOCK, AES_XOR_BLOCK);
1185                            aes_copy_block, aes_xor_block);
1189                        ASSERT(aes_ctx.ac_remainder_len == 0);
1190                        if ((ret == CRYPTO_SUCCESS) &&
1191                            (ciphertext != plaintext)) {
1192                                plaintext->cd_length =
1193                                    plaintext->cd_offset - saved_offset;
1194                        } else {
1195                                plaintext->cd_length = saved_length;
1196                        }
1197                } else if (mechanism->cm_type == AES_GCM_MECH_INFO_TYPE ||
1198                    mechanism->cm_type == AES_GMAC_MECH_INFO_TYPE) {
1199                        ret = gcm_decrypt_final((gcm_ctx_t *)&aes_ctx,
1200                            plaintext, AES_BLOCK_LEN, aes_encrypt_block,
1201                            AES_COPY_BLOCK, AES_XOR_BLOCK, aes_ctr_mode);
1198                            aes_xor_block);
1202                        ASSERT(aes_ctx.ac_remainder_len == 0);
1203                        if ((ret == CRYPTO_SUCCESS) &&
1204                            (ciphertext != plaintext)) {
1205                                plaintext->cd_length =
1206                                    plaintext->cd_offset - saved_offset;
1207                        } else {
1208                                plaintext->cd_length = saved_length;
1209                        }
1210                } else if (mechanism->cm_type != AES_CTR_MECH_INFO_TYPE) {
1211                        ASSERT(aes_ctx.ac_remainder_len == 0);
1212                        if (ciphertext != plaintext)
1213                                plaintext->cd_length =
1214                                    plaintext->cd_offset - saved_offset;
1215                } else {
1216                        if (aes_ctx.ac_remainder_len > 0) {
1217                                ret = ctr_mode_final((ctr_ctx_t *)&aes_ctx,
1218                                    plaintext, aes_encrypt_block);
1219                                if (ret == CRYPTO_DATA_LEN_RANGE)
1220                                        ret = CRYPTO_ENCRYPTED_DATA_LEN_RANGE;
1221                                if (ret != CRYPTO_SUCCESS)
1222                                        goto out;
1223                        }
1224                        if (ciphertext != plaintext)
1225                                plaintext->cd_length =
1226                                    plaintext->cd_offset - saved_offset;
1227                }
1228        } else {
1229                plaintext->cd_length = saved_length;
1230        }
1231        plaintext->cd_offset = saved_offset;

1233 out:
1234        if (aes_ctx.ac_flags & PROVIDER_OWNS_KEY_SCHEDULE) {
1235                bzero(aes_ctx.ac_keysched, aes_ctx.ac_keysched_len);
1236                kmem_free(aes_ctx.ac_keysched, aes_ctx.ac_keysched_len);
1237        }
```



```
1239        if (aes_ctx.ac_flags & CCM_MODE) {
1240                if (aes_ctx.ac_pt_buf != NULL) {
1241                        kmem_free(aes_ctx.ac_pt_buf, aes_ctx.ac_data_len);
1242                }
1240        } else if (aes_ctx.ac_flags & (GCM_MODE|GMAC_MODE)) {
1241                if (((gcm_ctx_t *)&aes_ctx)->gcm_pt_buf != NULL) {
1242                        kmem_free(((gcm_ctx_t *)&aes_ctx)->gcm_pt_buf,
1243                            ((gcm_ctx_t *)&aes_ctx)->gcm_pt_buf_len);
1243        }
1245        }

1245        return (ret);
1246 }
_____unchanged_portion_omitted_


1311 static int
1312 aes_common_init_ctx(aes_ctx_t *aes_ctx, crypto_spi_ctx_template_t *template,
1313     crypto_mechanism_t *mechanism, crypto_key_t *key, int kmflag,
1314     boolean_t is_encrypt_init)
1315 {
1316        int rv = CRYPTO_SUCCESS;
1317        void *keysched;
1318        size_t size;

1320        if (template == NULL) {
1321                if ((keysched = aes_alloc_keysched(&size, kmflag)) == NULL)
1322                        return (CRYPTO_HOST_MEMORY);
1323                /*
1324                 * Initialize key schedule.
1325                 * Key length is stored in the key.
1326                 */
1327                if ((rv = init_keysched(key, keysched)) != CRYPTO_SUCCESS) {
1328                        kmem_free(keysched, size);
1329                        return (rv);
1330                }

1332                aes_ctx->ac_flags |= PROVIDER_OWNS_KEY_SCHEDULE;
1333                aes_ctx->ac_keysched_len = size;
1334        } else {
1335                keysched = template;
1336        }
1337        aes_ctx->ac_keysched = keysched;

1339        switch (mechanism->cm_type) {
1340        case AES_CBC_MECH_INFO_TYPE:
1341                rv = cbc_init_ctx((cbc_ctx_t *)aes_ctx, mechanism->cm_param,
1342                    mechanism->cm_param_len, AES_BLOCK_LEN, aes_copy_block64);
1343                break;
1344        case AES_CTR_MECH_INFO_TYPE: {
1345                CK_AES_CTR_PARAMS *pp;

1347                if (mechanism->cm_param == NULL ||
1348                    mechanism->cm_param_len != sizeof (CK_AES_CTR_PARAMS)) {
1349                        return (CRYPTO_MECHANISM_PARAM_INVALID);
1350                }
1351                pp = (CK_AES_CTR_PARAMS *)(void *)mechanism->cm_param;
1352                rv = ctr_init_ctx((ctr_ctx_t *)aes_ctx, pp->ulCounterBits,
1353                    pp->cb, AES_COPY_BLOCK);
1355                    pp->cb, aes_copy_block);
1354                break;
1355        }
1356        case AES_CCM_MECH_INFO_TYPE:
1357                if (mechanism->cm_param == NULL ||
1358                    mechanism->cm_param_len != sizeof (CK_AES_CCM_PARAMS)) {
```

```
1359                         return (CRYPTO_MECHANISM_PARAM_INVALID);
1360                 }
1361                 rv = ccm_init_ctx((ccm_ctx_t *)aes_ctx, mechanism->cm_param,
1362                     kmflag, is_encrypt_init, AES_BLOCK_LEN, aes_encrypt_block,
1363                     AES_XOR_BLOCK);
1365                     aes_xor_block);
1364                 break;
1365         case AES_GCM_MECH_INFO_TYPE:
1366                 if (mechanism->cm_param == NULL ||
1367                     mechanism->cm_param_len != sizeof (CK_AES_GCM_PARAMS)) {
1368                         return (CRYPTO_MECHANISM_PARAM_INVALID);
1369                 }
1370                 rv = gcm_init_ctx((gcm_ctx_t *)aes_ctx, mechanism->cm_param,
1371                     AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
1372                     AES_XOR_BLOCK);
1373                     AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
1374                     aes_xor_block);
1373                 break;
1374         case AES_GMAC_MECH_INFO_TYPE:
1375                 if (mechanism->cm_param == NULL ||
1376                     mechanism->cm_param_len != sizeof (CK_AES_GMAC_PARAMS)) {
1377                         return (CRYPTO_MECHANISM_PARAM_INVALID);
1378                 }
1379                 rv = gmac_init_ctx((gcm_ctx_t *)aes_ctx, mechanism->cm_param,
1380                     AES_BLOCK_LEN, aes_encrypt_block, AES_COPY_BLOCK,
1381                     AES_XOR_BLOCK);
1382                     AES_BLOCK_LEN, aes_encrypt_block, aes_copy_block,
1383                     aes_xor_block);
1382                 break;
1383         case AES_ECB_MECH_INFO_TYPE:
1384                 aes_ctx->ac_flags |= ECB_MODE;
1385         }

1387         if (rv != CRYPTO_SUCCESS) {
1388                 if (aes_ctx->ac_flags & PROVIDER_OWNS_KEY_SCHEDULE) {
1389                         bzero(keysched, size);
1390                         kmem_free(keysched, size);
1391                 }
1392         }

1394         return (rv);
1395 }
```

_____***unchanged_portion_omitted_***

****
22879 Thu Apr 30 20:52:32 2015
new/usr/src/uts/common/crypto/io/blowfish.c
4896 Performance improvements for KCF AES modes
****

```
  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 2010 Sun Microsystems, Inc.  All rights reserved.
 23  * Use is subject to license terms.
 24  */
 25 /*
 26  * Copyright 2015 by Saso Kiselkov. All rights reserved.
 27  */

 29 /*
 30  * Blowfish provider for the Kernel Cryptographic Framework (KCF)
 31  */

 33 #include <sys/types.h>
 34 #include <sys/systm.h>
 35 #include <sys/modctl.h>
 36 #include <sys/cmn_err.h>
 37 #include <sys/ddi.h>
 38 #include <sys/crypto/common.h>
 39 #include <sys/crypto/spi.h>
 40 #include <sys/sysmacros.h>
 41 #include <sys/strsun.h>
 42 #include <sys/note.h>
 43 #include <modes/modes.h>
 44 #include <blowfish/blowfish_impl.h>

 46 extern struct mod_ops mod_cryptoops;

 48 /*
 49  * Module linkage information for the kernel.
 50  */
 51 static struct modlcrypto modlcrypto = {
 52         &mod_cryptoops,
 53         "Blowfish Kernel SW Provider"
 54 };
```

*_____unchanged_portion_omitted_*

```
 324 static void
 325 blowfish_copy_block64(const uint8_t *in, uint64_t *out)
 322 blowfish_copy_block64(uint8_t *in, uint64_t *out)
 326 {
 327         if (IS_P2ALIGNED(in, sizeof (uint64_t))) {
```



```
 328                 /* LINTED: pointer alignment */
 329                 out[0] = *(uint64_t *)&in[0];
 330         } else {
 331                 uint8_t *iv8 = (uint8_t *)&out[0];

 333                 BLOWFISH_COPY_BLOCK(in, iv8);
 334         }
 335 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   28913 Thu Apr 30 20:52:32 2015
new/usr/src/uts/common/des/des_crypt.c
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * Portions of this source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */

/*
 * des_crypt.c, DES encryption library routines
 */

#include <sys/errno.h>
#include <sys/modctl.h>

#include <sys/systm.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/crypto/common.h>
#include <sys/crypto/spi.h>
#include <sys/sysmacros.h>
#include <sys/strsun.h>
#include <sys/note.h>
#include <modes/modes.h>
#define _DES_IMPL
#include <des/des_impl.h>

#include <sys/types.h>
#include <rpc/des_crypt.h>
#include <des/des.h>

#ifdef sun_hardware
```



```
#include <sys/ioctl.h>
#ifdef _KERNEL
#include <sys/conf.h>
static int g_desfd = -1;
#define getdesfd()      (cdevsw[11].d_open(0, 0) ? -1 : 0)
#define ioctl(a, b, c)  (cdevsw[11].d_ioctl(0, b, c, 0) ? -1 : 0)
#else
#define getdesfd()      (open("/dev/des", 0, 0))
#endif  /* _KERNEL */
#endif  /* sun */

static int common_crypt(char *key, char *buf, size_t len,
    unsigned int mode, struct desparams *desp);

extern int _des_crypt(char *buf, size_t len, struct desparams *desp);

extern struct mod_ops mod_cryptoops;

/*
 * Module linkage information for the kernel.
 */
static struct modlmisc modlmisc = {
        &mod_miscops,
        "des encryption",
};
_____unchanged_portion_omitted_

static void
des_copy_block64(const uint8_t *in, uint64_t *out)
des_copy_block64(uint8_t *in, uint64_t *out)
{
        if (IS_P2ALIGNED(in, sizeof (uint64_t))) {
                /* LINTED: pointer alignment */
                out[0] = *(uint64_t *)&in[0];
        } else {
                uint64_t tmp64;

#ifdef _BIG_ENDIAN
                tmp64 = (((uint64_t)in[0] << 56) |
                    ((uint64_t)in[1] << 48) |
                    ((uint64_t)in[2] << 40) |
                    ((uint64_t)in[3] << 32) |
                    ((uint64_t)in[4] << 24) |
                    ((uint64_t)in[5] << 16) |
                    ((uint64_t)in[6] << 8) |
                    (uint64_t)in[7]);
#else
                tmp64 = (((uint64_t)in[7] << 56) |
                    ((uint64_t)in[6] << 48) |
                    ((uint64_t)in[5] << 40) |
                    ((uint64_t)in[4] << 32) |
                    ((uint64_t)in[3] << 24) |
                    ((uint64_t)in[2] << 16) |
                    ((uint64_t)in[1] << 8) |
                    (uint64_t)in[0]);
#endif /* _BIG_ENDIAN */

                out[0] = tmp64;
        }
}
_____unchanged_portion_omitted_
```

```
**********************************************************
   53911 Thu Apr 30 20:52:32 2015
new/usr/src/uts/common/sys/crypto/impl.h
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
/*
 * 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 (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 by Saso Kiselkov. All rights reserved.
 */

#ifndef _SYS_CRYPTO_IMPL_H
#define _SYS_CRYPTO_IMPL_H

/*
 * Kernel Cryptographic Framework private implementation definitions.
 */

#include <sys/types.h>
#include <sys/param.h>

#ifdef _KERNEL
#include <sys/crypto/common.h>
#include <sys/crypto/api.h>
#include <sys/crypto/spi.h>
#include <sys/crypto/ioctl.h>
#include <sys/tnf_probe.h>
#include <sys/atomic.h>
#include <sys/project.h>
#include <sys/taskq.h>
#include <sys/rctl.h>
#include <sys/cpuvar.h>
#endif /* _KERNEL */

#ifdef __cplusplus
extern "C" {
#endif

#ifdef _KERNEL

/*
 * Prefixes convention: structures internal to the kernel cryptographic
 * framework start with 'kcf_'. Exposed structure start with 'crypto_'.
 */

/* Provider stats. Not protected. */
typedef struct kcf_prov_stats {
        kstat_named_t   ps_ops_total;
        kstat_named_t   ps_ops_passed;
        kstat_named_t   ps_ops_failed;
        kstat_named_t   ps_ops_busy_rval;
} kcf_prov_stats_t;
```



```
_____unchanged_portion_omitted_

/* resource control framework handle used by /dev/crypto */
extern rctl_hndl_t rc_project_crypto_mem;
/*
 * Return codes for internal functions
 */
#define KCF_SUCCESS             0x0     /* Successful call */
#define KCF_INVALID_MECH_NUMBER 0x1     /* invalid mechanism number */
#define KCF_INVALID_MECH_NAME   0x2     /* invalid mechanism name */
#define KCF_INVALID_MECH_CLASS  0x3     /* invalid mechanism class */
#define KCF_MECH_TAB_FULL       0x4     /* Need more room in the mech tabs. */
#define KCF_INVALID_INDX        ((ushort_t)-1)

/*
 * kCF internal mechanism and function group for tracking RNG providers.
 */
#define SUN_RANDOM              "random"
#define CRYPTO_FG_RANDOM        0x80000000      /* generate_random() */

/*
 * Wrappers for ops vectors. In the wrapper definitions below, the pd
 * argument always corresponds to a pointer to a provider descriptor
 * of type kcf_prov_desc_t.
 */

#define KCF_PROV_CONTROL_OPS(pd)        ((pd)->pd_ops_vector->co_control_ops)
#define KCF_PROV_CTX_OPS(pd)            ((pd)->pd_ops_vector->co_ctx_ops)
#define KCF_PROV_DIGEST_OPS(pd)         ((pd)->pd_ops_vector->co_digest_ops)
#define KCF_PROV_CIPHER_OPS(pd)         ((pd)->pd_ops_vector->co_cipher_ops)
#define KCF_PROV_MAC_OPS(pd)            ((pd)->pd_ops_vector->co_mac_ops)
#define KCF_PROV_SIGN_OPS(pd)           ((pd)->pd_ops_vector->co_sign_ops)
#define KCF_PROV_VERIFY_OPS(pd)         ((pd)->pd_ops_vector->co_verify_ops)
#define KCF_PROV_DUAL_OPS(pd)           ((pd)->pd_ops_vector->co_dual_ops)
#define KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) \
        ((pd)->pd_ops_vector->co_dual_cipher_mac_ops)
#define KCF_PROV_RANDOM_OPS(pd)         ((pd)->pd_ops_vector->co_random_ops)
#define KCF_PROV_SESSION_OPS(pd)        ((pd)->pd_ops_vector->co_session_ops)
#define KCF_PROV_OBJECT_OPS(pd)         ((pd)->pd_ops_vector->co_object_ops)
#define KCF_PROV_KEY_OPS(pd)            ((pd)->pd_ops_vector->co_key_ops)
#define KCF_PROV_PROVIDER_OPS(pd)       ((pd)->pd_ops_vector->co_provider_ops)
#define KCF_PROV_MECH_OPS(pd)           ((pd)->pd_ops_vector->co_mech_ops)
#define KCF_PROV_NOSTORE_KEY_OPS(pd)    \
        ((pd)->pd_ops_vector->co_nostore_key_ops)
#define KCF_PROV_FIPS140_OPS(pd)        ((pd)->pd_ops_vector->co_fips140_ops)
#define KCF_PROV_PROVMGMT_OPS(pd)       ((pd)->pd_ops_vector->co_provider_ops)

/*
 * Wrappers for crypto_control_ops(9S) entry points.
 */

#define KCF_PROV_STATUS(pd, status) ( \
        (KCF_PROV_CONTROL_OPS(pd) && \
        KCF_PROV_CONTROL_OPS(pd)->provider_status) ? \
        KCF_PROV_CONTROL_OPS(pd)->provider_status( \
            (pd)->pd_prov_handle, status) : \
        CRYPTO_NOT_SUPPORTED)

/*
 * Wrappers for crypto_ctx_ops(9S) entry points.
 */
```

```
#define KCF_PROV_CREATE_CTX_TEMPLATE(pd, mech, key, template, size, req) ( \
        (KCF_PROV_CTX_OPS(pd) && KCF_PROV_CTX_OPS(pd)->create_ctx_template) ? \
        KCF_PROV_CTX_OPS(pd)->create_ctx_template( \
            (pd)->pd_prov_handle, mech, key, template, size, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_FREE_CONTEXT(pd, ctx) ( \
        (KCF_PROV_CTX_OPS(pd) && KCF_PROV_CTX_OPS(pd)->free_context) ? \
        KCF_PROV_CTX_OPS(pd)->free_context(ctx) : CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_COPYIN_MECH(pd, umech, kmech, errorp, mode) ( \
        (KCF_PROV_MECH_OPS(pd) && KCF_PROV_MECH_OPS(pd)->copyin_mechanism) ? \
        KCF_PROV_MECH_OPS(pd)->copyin_mechanism( \
            (pd)->pd_prov_handle, umech, kmech, errorp, mode) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_COPYOUT_MECH(pd, kmech, umech, errorp, mode) ( \
        (KCF_PROV_MECH_OPS(pd) && KCF_PROV_MECH_OPS(pd)->copyout_mechanism) ? \
        KCF_PROV_MECH_OPS(pd)->copyout_mechanism( \
            (pd)->pd_prov_handle, kmech, umech, errorp, mode) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_FREE_MECH(pd, prov_mech) ( \
        (KCF_PROV_MECH_OPS(pd) && KCF_PROV_MECH_OPS(pd)->free_mechanism) ? \
        KCF_PROV_MECH_OPS(pd)->free_mechanism( \
            (pd)->pd_prov_handle, prov_mech) : CRYPTO_NOT_SUPPORTED)

/*
 * Wrappers for crypto_digest_ops(9S) entry points.
 */

#define KCF_PROV_DIGEST_INIT(pd, ctx, mech, req) ( \
        (KCF_PROV_DIGEST_OPS(pd) && KCF_PROV_DIGEST_OPS(pd)->digest_init) ? \
        KCF_PROV_DIGEST_OPS(pd)->digest_init(ctx, mech, req) : \
        CRYPTO_NOT_SUPPORTED)

/*
 * The _ (underscore) in _digest is needed to avoid replacing the
 * function digest().
 */
#define KCF_PROV_DIGEST(pd, ctx, data, _digest, req) ( \
        (KCF_PROV_DIGEST_OPS(pd) && KCF_PROV_DIGEST_OPS(pd)->digest) ? \
        KCF_PROV_DIGEST_OPS(pd)->digest(ctx, data, _digest, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DIGEST_UPDATE(pd, ctx, data, req) ( \
        (KCF_PROV_DIGEST_OPS(pd) && KCF_PROV_DIGEST_OPS(pd)->digest_update) ? \
        KCF_PROV_DIGEST_OPS(pd)->digest_update(ctx, data, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DIGEST_KEY(pd, ctx, key, req) ( \
        (KCF_PROV_DIGEST_OPS(pd) && KCF_PROV_DIGEST_OPS(pd)->digest_key) ? \
        KCF_PROV_DIGEST_OPS(pd)->digest_key(ctx, key, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DIGEST_FINAL(pd, ctx, digest, req) ( \
        (KCF_PROV_DIGEST_OPS(pd) && KCF_PROV_DIGEST_OPS(pd)->digest_final) ? \
        KCF_PROV_DIGEST_OPS(pd)->digest_final(ctx, digest, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DIGEST_ATOMIC(pd, session, mech, data, digest, req) ( \
        (KCF_PROV_DIGEST_OPS(pd) && KCF_PROV_DIGEST_OPS(pd)->digest_atomic) ? \
        KCF_PROV_DIGEST_OPS(pd)->digest_atomic( \
            (pd)->pd_prov_handle, session, mech, data, digest, req) : \
        CRYPTO_NOT_SUPPORTED)
```



```
/*
 * Wrappers for crypto_cipher_ops(9S) entry points.
 */

#define KCF_PROV_ENCRYPT_INIT(pd, ctx, mech, key, template, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->encrypt_init) ? \
        KCF_PROV_CIPHER_OPS(pd)->encrypt_init(ctx, mech, key, template, \
            req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_ENCRYPT(pd, ctx, plaintext, ciphertext, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->encrypt) ? \
        KCF_PROV_CIPHER_OPS(pd)->encrypt(ctx, plaintext, ciphertext, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_ENCRYPT_UPDATE(pd, ctx, plaintext, ciphertext, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->encrypt_update) ? \
        KCF_PROV_CIPHER_OPS(pd)->encrypt_update(ctx, plaintext, \
            ciphertext, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_ENCRYPT_FINAL(pd, ctx, ciphertext, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->encrypt_final) ? \
        KCF_PROV_CIPHER_OPS(pd)->encrypt_final(ctx, ciphertext, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_ENCRYPT_ATOMIC(pd, session, mech, key, plaintext, ciphertext, \
            template, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->encrypt_atomic) ? \
        KCF_PROV_CIPHER_OPS(pd)->encrypt_atomic( \
            (pd)->pd_prov_handle, session, mech, key, plaintext, ciphertext, \
            template, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DECRYPT_INIT(pd, ctx, mech, key, template, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->decrypt_init) ? \
        KCF_PROV_CIPHER_OPS(pd)->decrypt_init(ctx, mech, key, template, \
            req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DECRYPT(pd, ctx, ciphertext, plaintext, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->decrypt) ? \
        KCF_PROV_CIPHER_OPS(pd)->decrypt(ctx, ciphertext, plaintext, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DECRYPT_UPDATE(pd, ctx, ciphertext, plaintext, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->decrypt_update) ? \
        KCF_PROV_CIPHER_OPS(pd)->decrypt_update(ctx, ciphertext, \
            plaintext, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DECRYPT_FINAL(pd, ctx, plaintext, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->decrypt_final) ? \
        KCF_PROV_CIPHER_OPS(pd)->decrypt_final(ctx, plaintext, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_DECRYPT_ATOMIC(pd, session, mech, key, ciphertext, plaintext, \
            template, req) ( \
        (KCF_PROV_CIPHER_OPS(pd) && KCF_PROV_CIPHER_OPS(pd)->decrypt_atomic) ? \
        KCF_PROV_CIPHER_OPS(pd)->decrypt_atomic( \
            (pd)->pd_prov_handle, session, mech, key, ciphertext, plaintext, \
            template, req) : \
        CRYPTO_NOT_SUPPORTED)

/*
```

```
 736  * Wrappers for crypto_mac_ops(9S) entry points.
 737  */

 739 #define KCF_PROV_MAC_INIT(pd, ctx, mech, key, template, req) ( \
 740         (KCF_PROV_MAC_OPS(pd) && KCF_PROV_MAC_OPS(pd)->mac_init) ? \
 741         KCF_PROV_MAC_OPS(pd)->mac_init(ctx, mech, key, template, req) \
 742         : CRYPTO_NOT_SUPPORTED)

 744 /*
 745  * The _ (underscore) in _mac is needed to avoid replacing the
 746  * function mac().
 747  */
 748 #define KCF_PROV_MAC(pd, ctx, data, _mac, req) ( \
 749         (KCF_PROV_MAC_OPS(pd) && KCF_PROV_MAC_OPS(pd)->mac) ? \
 750         KCF_PROV_MAC_OPS(pd)->mac(ctx, data, _mac, req) : \
 751         CRYPTO_NOT_SUPPORTED)

 753 #define KCF_PROV_MAC_UPDATE(pd, ctx, data, req) ( \
 754         (KCF_PROV_MAC_OPS(pd) && KCF_PROV_MAC_OPS(pd)->mac_update) ? \
 755         KCF_PROV_MAC_OPS(pd)->mac_update(ctx, data, req) : \
 756         CRYPTO_NOT_SUPPORTED)

 758 #define KCF_PROV_MAC_FINAL(pd, ctx, mac, req) ( \
 759         (KCF_PROV_MAC_OPS(pd) && KCF_PROV_MAC_OPS(pd)->mac_final) ? \
 760         KCF_PROV_MAC_OPS(pd)->mac_final(ctx, mac, req) : \
 761         CRYPTO_NOT_SUPPORTED)

 763 #define KCF_PROV_MAC_ATOMIC(pd, session, mech, key, data, mac, template, \
 764             req) ( \
 765         (KCF_PROV_MAC_OPS(pd) && KCF_PROV_MAC_OPS(pd)->mac_atomic) ? \
 766         KCF_PROV_MAC_OPS(pd)->mac_atomic( \
 767             (pd)->pd_prov_handle, session, mech, key, data, mac, template, \
 768             req) : \
 769         CRYPTO_NOT_SUPPORTED)

 771 #define KCF_PROV_MAC_VERIFY_ATOMIC(pd, session, mech, key, data, mac, \
 772             template, req) ( \
 773         (KCF_PROV_MAC_OPS(pd) && KCF_PROV_MAC_OPS(pd)->mac_verify_atomic) ? \
 774         KCF_PROV_MAC_OPS(pd)->mac_verify_atomic( \
 775             (pd)->pd_prov_handle, session, mech, key, data, mac, template, \
 776             req) : \
 777         CRYPTO_NOT_SUPPORTED)

 779 /*
 780  * Wrappers for crypto_sign_ops(9S) entry points.
 781  */

 783 #define KCF_PROV_SIGN_INIT(pd, ctx, mech, key, template, req) ( \
 784         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign_init) ? \
 785         KCF_PROV_SIGN_OPS(pd)->sign_init( \
 786             ctx, mech, key, template, req) : CRYPTO_NOT_SUPPORTED)

 788 #define KCF_PROV_SIGN(pd, ctx, data, sig, req) ( \
 789         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign) ? \
 790         KCF_PROV_SIGN_OPS(pd)->sign(ctx, data, sig, req) : \
 791         CRYPTO_NOT_SUPPORTED)

 793 #define KCF_PROV_SIGN_UPDATE(pd, ctx, data, req) ( \
 794         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign_update) ? \
 795         KCF_PROV_SIGN_OPS(pd)->sign_update(ctx, data, req) : \
 796         CRYPTO_NOT_SUPPORTED)

 798 #define KCF_PROV_SIGN_FINAL(pd, ctx, sig, req) ( \
 799         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign_final) ? \
 800         KCF_PROV_SIGN_OPS(pd)->sign_final(ctx, sig, req) : \
 801         CRYPTO_NOT_SUPPORTED)
```



```
 803 #define KCF_PROV_SIGN_ATOMIC(pd, session, mech, key, data, template, \
 804             sig, req) ( \
 805         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign_atomic) ? \
 806         KCF_PROV_SIGN_OPS(pd)->sign_atomic( \
 807             (pd)->pd_prov_handle, session, mech, key, data, sig, template, \
 808             req) : CRYPTO_NOT_SUPPORTED)

 810 #define KCF_PROV_SIGN_RECOVER_INIT(pd, ctx, mech, key, template, \
 811             req) ( \
 812         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign_recover_init) ? \
 813         KCF_PROV_SIGN_OPS(pd)->sign_recover_init(ctx, mech, key, template, \
 814             req) : CRYPTO_NOT_SUPPORTED)

 816 #define KCF_PROV_SIGN_RECOVER(pd, ctx, data, sig, req) ( \
 817         (KCF_PROV_SIGN_OPS(pd) && KCF_PROV_SIGN_OPS(pd)->sign_recover) ? \
 818         KCF_PROV_SIGN_OPS(pd)->sign_recover(ctx, data, sig, req) : \
 819         CRYPTO_NOT_SUPPORTED)

 821 #define KCF_PROV_SIGN_RECOVER_ATOMIC(pd, session, mech, key, data, template, \
 822             sig, req) ( \
 823         (KCF_PROV_SIGN_OPS(pd) && \
 824         KCF_PROV_SIGN_OPS(pd)->sign_recover_atomic) ? \
 825         KCF_PROV_SIGN_OPS(pd)->sign_recover_atomic( \
 826             (pd)->pd_prov_handle, session, mech, key, data, sig, template, \
 827             req) : CRYPTO_NOT_SUPPORTED)

 829 /*
 830  * Wrappers for crypto_verify_ops(9S) entry points.
 831  */

 833 #define KCF_PROV_VERIFY_INIT(pd, ctx, mech, key, template, req) ( \
 834         (KCF_PROV_VERIFY_OPS(pd) && KCF_PROV_VERIFY_OPS(pd)->verify_init) ? \
 835         KCF_PROV_VERIFY_OPS(pd)->verify_init(ctx, mech, key, template, \
 836             req) : CRYPTO_NOT_SUPPORTED)

 838 #define KCF_PROV_VERIFY(pd, ctx, data, sig, req) ( \
 839         (KCF_PROV_VERIFY_OPS(pd) && KCF_PROV_VERIFY_OPS(pd)->verify) ? \
 840         KCF_PROV_VERIFY_OPS(pd)->verify(ctx, data, sig, req) : \
 841         CRYPTO_NOT_SUPPORTED)

 843 #define KCF_PROV_VERIFY_UPDATE(pd, ctx, data, req) ( \
 844         (KCF_PROV_VERIFY_OPS(pd) && KCF_PROV_VERIFY_OPS(pd)->verify_update) ? \
 845         KCF_PROV_VERIFY_OPS(pd)->verify_update(ctx, data, req) : \
 846         CRYPTO_NOT_SUPPORTED)

 848 #define KCF_PROV_VERIFY_FINAL(pd, ctx, sig, req) ( \
 849         (KCF_PROV_VERIFY_OPS(pd) && KCF_PROV_VERIFY_OPS(pd)->verify_final) ? \
 850         KCF_PROV_VERIFY_OPS(pd)->verify_final(ctx, sig, req) : \
 851         CRYPTO_NOT_SUPPORTED)

 853 #define KCF_PROV_VERIFY_ATOMIC(pd, session, mech, key, data, template, sig, \
 854             req) ( \
 855         (KCF_PROV_VERIFY_OPS(pd) && KCF_PROV_VERIFY_OPS(pd)->verify_atomic) ? \
 856         KCF_PROV_VERIFY_OPS(pd)->verify_atomic( \
 857             (pd)->pd_prov_handle, session, mech, key, data, sig, template, \
 858             req) : CRYPTO_NOT_SUPPORTED)

 860 #define KCF_PROV_VERIFY_RECOVER_INIT(pd, ctx, mech, key, template, \
 861             req) ( \
 862         (KCF_PROV_VERIFY_OPS(pd) && \
 863         KCF_PROV_VERIFY_OPS(pd)->verify_recover_init) ? \
 864         KCF_PROV_VERIFY_OPS(pd)->verify_recover_init(ctx, mech, key, \
 865             template, req) : CRYPTO_NOT_SUPPORTED)

 867 /* verify_recover() CSPI routine has different argument order than verify() */
```

```
868 #define KCF_PROV_VERIFY_RECOVER(pd, ctx, sig, data, req) ( \
869         (KCF_PROV_VERIFY_OPS(pd) && KCF_PROV_VERIFY_OPS(pd)->verify_recover) ? \
870         KCF_PROV_VERIFY_OPS(pd)->verify_recover(ctx, sig, data, req) : \
871         CRYPTO_NOT_SUPPORTED)

873 /*
874  * verify_recover_atomic() CSPI routine has different argument order
875  * than verify_atomic().
876  */
877 #define KCF_PROV_VERIFY_RECOVER_ATOMIC(pd, session, mech, key, sig, \
878             template, data,  req) ( \
879         (KCF_PROV_VERIFY_OPS(pd) && \
880         KCF_PROV_VERIFY_OPS(pd)->verify_recover_atomic) ? \
881         KCF_PROV_VERIFY_OPS(pd)->verify_recover_atomic( \
882             (pd)->pd_prov_handle, session, mech, key, sig, data, template, \
883             req) : CRYPTO_NOT_SUPPORTED)

885 /*
886  * Wrappers for crypto_dual_ops(9S) entry points.
887  */

889 #define KCF_PROV_DIGEST_ENCRYPT_UPDATE(digest_ctx, encrypt_ctx, plaintext, \
890             ciphertext, req) ( \
891         (KCF_PROV_DUAL_OPS(pd) && \
892         KCF_PROV_DUAL_OPS(pd)->digest_encrypt_update) ? \
893         KCF_PROV_DUAL_OPS(pd)->digest_encrypt_update( \
894             digest_ctx, encrypt_ctx, plaintext, ciphertext, req) : \
895         CRYPTO_NOT_SUPPORTED)

897 #define KCF_PROV_DECRYPT_DIGEST_UPDATE(decrypt_ctx, digest_ctx, ciphertext, \
898             plaintext, req) ( \
899         (KCF_PROV_DUAL_OPS(pd) && \
900         KCF_PROV_DUAL_OPS(pd)->decrypt_digest_update) ? \
901         KCF_PROV_DUAL_OPS(pd)->decrypt_digest_update( \
902             decrypt_ctx, digest_ctx, ciphertext, plaintext, req) : \
903         CRYPTO_NOT_SUPPORTED)

905 #define KCF_PROV_SIGN_ENCRYPT_UPDATE(sign_ctx, encrypt_ctx, plaintext, \
906             ciphertext, req) ( \
907         (KCF_PROV_DUAL_OPS(pd) && \
908         KCF_PROV_DUAL_OPS(pd)->sign_encrypt_update) ? \
909         KCF_PROV_DUAL_OPS(pd)->sign_encrypt_update( \
910             sign_ctx, encrypt_ctx, plaintext, ciphertext, req) : \
911         CRYPTO_NOT_SUPPORTED)

913 #define KCF_PROV_DECRYPT_VERIFY_UPDATE(decrypt_ctx, verify_ctx, ciphertext, \
914             plaintext, req) ( \
915         (KCF_PROV_DUAL_OPS(pd) && \
916         KCF_PROV_DUAL_OPS(pd)->decrypt_verify_update) ? \
917         KCF_PROV_DUAL_OPS(pd)->decrypt_verify_update( \
918             decrypt_ctx, verify_ctx, ciphertext, plaintext, req) : \
919         CRYPTO_NOT_SUPPORTED)

921 /*
922  * Wrappers for crypto_dual_cipher_mac_ops(9S) entry points.
923  */

925 #define KCF_PROV_ENCRYPT_MAC_INIT(pd, ctx, encr_mech, encr_key, mac_mech, \
926             mac_key, encr_ctx_template, mac_ctx_template, req) ( \
927         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
928         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_init) ? \
929         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_init( \
930             ctx, encr_mech, encr_key, mac_mech, mac_key, encr_ctx_template, \
931             mac_ctx_template, req) : \
932         CRYPTO_NOT_SUPPORTED)
```



```
934 #define KCF_PROV_ENCRYPT_MAC(pd, ctx, plaintext, ciphertext, mac, req) ( \
935         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
936         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac) ? \
937         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac( \
938             ctx, plaintext, ciphertext, mac, req) : \
939         CRYPTO_NOT_SUPPORTED)

941 #define KCF_PROV_ENCRYPT_MAC_UPDATE(pd, ctx, plaintext, ciphertext, req) ( \
942         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
943         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_update) ? \
944         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_update( \
945             ctx, plaintext, ciphertext, req) : \
946         CRYPTO_NOT_SUPPORTED)

948 #define KCF_PROV_ENCRYPT_MAC_FINAL(pd, ctx, ciphertext, mac, req) ( \
949         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
950         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_final) ? \
951         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_final( \
952             ctx, ciphertext, mac, req) : \
953         CRYPTO_NOT_SUPPORTED)

955 #define KCF_PROV_ENCRYPT_MAC_ATOMIC(pd, session, encr_mech, encr_key, \
956             mac_mech, mac_key, plaintext, ciphertext, mac, \
957             encr_ctx_template, mac_ctx_template, req) ( \
958         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
959         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_atomic) ? \
960         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->encrypt_mac_atomic( \
961             (pd)->pd_prov_handle, session, encr_mech, encr_key, \
962             mac_mech, mac_key, plaintext, ciphertext, mac, \
963             encr_ctx_template, mac_ctx_template, req) : \
964         CRYPTO_NOT_SUPPORTED)

966 #define KCF_PROV_MAC_DECRYPT_INIT(pd, ctx, mac_mech, mac_key, decr_mech, \
967             decr_key, mac_ctx_template, decr_ctx_template, req) ( \
968         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
969         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_init) ? \
970         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_init( \
971             ctx, mac_mech, mac_key, decr_mech, decr_key, mac_ctx_template, \
972             decr_ctx_template, req) : \
973         CRYPTO_NOT_SUPPORTED)

975 #define KCF_PROV_MAC_DECRYPT(pd, ctx, ciphertext, mac, plaintext, req) ( \
976         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
977         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt) ? \
978         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt( \
979             ctx, ciphertext, mac, plaintext, req) : \
980         CRYPTO_NOT_SUPPORTED)

982 #define KCF_PROV_MAC_DECRYPT_UPDATE(pd, ctx, ciphertext, plaintext, req) ( \
983         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
984         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_update) ? \
985         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_update( \
986             ctx, ciphertext, plaintext, req) : \
987         CRYPTO_NOT_SUPPORTED)

989 #define KCF_PROV_MAC_DECRYPT_FINAL(pd, ctx, mac, plaintext, req) ( \
990         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
991         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_final) ? \
992         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_final( \
993             ctx, mac, plaintext, req) : \
994         CRYPTO_NOT_SUPPORTED)

996 #define KCF_PROV_MAC_DECRYPT_ATOMIC(pd, session, mac_mech, mac_key, \
997             decr_mech, decr_key, ciphertext, mac, plaintext, \
998             mac_ctx_template, decr_ctx_template, req) ( \
999         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
```

```
1000         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_atomic) ? \
1001         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_decrypt_atomic( \
1002             (pd)->pd_prov_handle, session, mac_mech, mac_key, \
1003             decr_mech, decr_key, ciphertext, mac, plaintext, \
1004             mac_ctx_template, decr_ctx_template, req) : \
1005         CRYPTO_NOT_SUPPORTED)

1007 #define KCF_PROV_MAC_VERIFY_DECRYPT_ATOMIC(pd, session, mac_mech, mac_key, \
1008             decr_mech, decr_key, ciphertext, mac, plaintext, \
1009             mac_ctx_template, decr_ctx_template, req) ( \
1010         (KCF_PROV_DUAL_CIPHER_MAC_OPS(pd) && \
1011         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_verify_decrypt_atomic \
1012             != NULL) ? \
1013         KCF_PROV_DUAL_CIPHER_MAC_OPS(pd)->mac_verify_decrypt_atomic( \
1014             (pd)->pd_prov_handle, session, mac_mech, mac_key, \
1015             decr_mech, decr_key, ciphertext, mac, plaintext, \
1016             mac_ctx_template, decr_ctx_template, req) : \
1017         CRYPTO_NOT_SUPPORTED)

1019 /*
1020  * Wrappers for crypto_random_number_ops(9S) entry points.
1021  */

1023 #define KCF_PROV_SEED_RANDOM(pd, session, buf, len, est, flags, req) ( \
1024         (KCF_PROV_RANDOM_OPS(pd) && KCF_PROV_RANDOM_OPS(pd)->seed_random) ? \
1025         KCF_PROV_RANDOM_OPS(pd)->seed_random((pd)->pd_prov_handle, \
1026             session, buf, len, est, flags, req) : CRYPTO_NOT_SUPPORTED)

1028 #define KCF_PROV_GENERATE_RANDOM(pd, session, buf, len, req) ( \
1029         (KCF_PROV_RANDOM_OPS(pd) && \
1030         KCF_PROV_RANDOM_OPS(pd)->generate_random) ? \
1031         KCF_PROV_RANDOM_OPS(pd)->generate_random((pd)->pd_prov_handle, \
1032             session, buf, len, req) : CRYPTO_NOT_SUPPORTED)

1034 /*
1035  * Wrappers for crypto_session_ops(9S) entry points.
1036  *
1037  * ops_pd is the provider descriptor that supplies the ops_vector.
1038  * pd is the descriptor that supplies the provider handle.
1039  * Only session open/close needs two handles.
1040  */

1042 #define KCF_PROV_SESSION_OPEN(ops_pd, session, req, pd) ( \
1043         (KCF_PROV_SESSION_OPS(ops_pd) && \
1044         KCF_PROV_SESSION_OPS(ops_pd)->session_open) ? \
1045         KCF_PROV_SESSION_OPS(ops_pd)->session_open((pd)->pd_prov_handle, \
1046             session, req) : CRYPTO_NOT_SUPPORTED)

1048 #define KCF_PROV_SESSION_CLOSE(ops_pd, session, req, pd) ( \
1049         (KCF_PROV_SESSION_OPS(ops_pd) && \
1050         KCF_PROV_SESSION_OPS(ops_pd)->session_close) ? \
1051         KCF_PROV_SESSION_OPS(ops_pd)->session_close((pd)->pd_prov_handle, \
1052             session, req) : CRYPTO_NOT_SUPPORTED)

1054 #define KCF_PROV_SESSION_LOGIN(pd, session, user_type, pin, len, req) ( \
1055         (KCF_PROV_SESSION_OPS(pd) && \
1056         KCF_PROV_SESSION_OPS(pd)->session_login) ? \
1057         KCF_PROV_SESSION_OPS(pd)->session_login((pd)->pd_prov_handle, \
1058             session, user_type, pin, len, req) : CRYPTO_NOT_SUPPORTED)

1060 #define KCF_PROV_SESSION_LOGOUT(pd, session, req) ( \
1061         (KCF_PROV_SESSION_OPS(pd) && \
1062         KCF_PROV_SESSION_OPS(pd)->session_logout) ? \
1063         KCF_PROV_SESSION_OPS(pd)->session_logout((pd)->pd_prov_handle, \
1064             session, req) : CRYPTO_NOT_SUPPORTED)
```



```
1066 /*
1067  * Wrappers for crypto_object_ops(9S) entry points.
1068  */

1070 #define KCF_PROV_OBJECT_CREATE(pd, session, template, count, object, req) ( \
1071         (KCF_PROV_OBJECT_OPS(pd) && KCF_PROV_OBJECT_OPS(pd)->object_create) ? \
1072         KCF_PROV_OBJECT_OPS(pd)->object_create((pd)->pd_prov_handle, \
1073             session, template, count, object, req) : CRYPTO_NOT_SUPPORTED)

1075 #define KCF_PROV_OBJECT_COPY(pd, session, object, template, count, \
1076             new_object, req) ( \
1077         (KCF_PROV_OBJECT_OPS(pd) && KCF_PROV_OBJECT_OPS(pd)->object_copy) ? \
1078         KCF_PROV_OBJECT_OPS(pd)->object_copy((pd)->pd_prov_handle, \
1079         session, object, template, count, new_object, req) : \
1080             CRYPTO_NOT_SUPPORTED)

1082 #define KCF_PROV_OBJECT_DESTROY(pd, session, object, req) ( \
1083         (KCF_PROV_OBJECT_OPS(pd) && KCF_PROV_OBJECT_OPS(pd)->object_destroy) ? \
1084         KCF_PROV_OBJECT_OPS(pd)->object_destroy((pd)->pd_prov_handle, \
1085             session, object, req) : CRYPTO_NOT_SUPPORTED)

1087 #define KCF_PROV_OBJECT_GET_SIZE(pd, session, object, size, req) ( \
1088         (KCF_PROV_OBJECT_OPS(pd) && \
1089         KCF_PROV_OBJECT_OPS(pd)->object_get_size) ? \
1090         KCF_PROV_OBJECT_OPS(pd)->object_get_size((pd)->pd_prov_handle, \
1091             session, object, size, req) : CRYPTO_NOT_SUPPORTED)

1093 #define KCF_PROV_OBJECT_GET_ATTRIBUTE_VALUE(pd, session, object, template, \
1094             count, req) ( \
1095         (KCF_PROV_OBJECT_OPS(pd) && \
1096         KCF_PROV_OBJECT_OPS(pd)->object_get_attribute_value) ? \
1097         KCF_PROV_OBJECT_OPS(pd)->object_get_attribute_value( \
1098         (pd)->pd_prov_handle, session, object, template, count, req) : \
1099             CRYPTO_NOT_SUPPORTED)

1101 #define KCF_PROV_OBJECT_SET_ATTRIBUTE_VALUE(pd, session, object, template, \
1102             count, req) ( \
1103         (KCF_PROV_OBJECT_OPS(pd) && \
1104         KCF_PROV_OBJECT_OPS(pd)->object_set_attribute_value) ? \
1105         KCF_PROV_OBJECT_OPS(pd)->object_set_attribute_value( \
1106         (pd)->pd_prov_handle, session, object, template, count, req) : \
1107             CRYPTO_NOT_SUPPORTED)

1109 #define KCF_PROV_OBJECT_FIND_INIT(pd, session, template, count, ppriv, \
1110             req) ( \
1111         (KCF_PROV_OBJECT_OPS(pd) && \
1112         KCF_PROV_OBJECT_OPS(pd)->object_find_init) ? \
1113         KCF_PROV_OBJECT_OPS(pd)->object_find_init((pd)->pd_prov_handle, \
1114         session, template, count, ppriv, req) : CRYPTO_NOT_SUPPORTED)

1116 #define KCF_PROV_OBJECT_FIND(pd, ppriv, objects, max_objects, object_count, \
1117             req) ( \
1118         (KCF_PROV_OBJECT_OPS(pd) && KCF_PROV_OBJECT_OPS(pd)->object_find) ? \
1119         KCF_PROV_OBJECT_OPS(pd)->object_find( \
1120         (pd)->pd_prov_handle, ppriv, objects, max_objects, object_count, \
1121         req) : CRYPTO_NOT_SUPPORTED)

1123 #define KCF_PROV_OBJECT_FIND_FINAL(pd, ppriv, req) ( \
1124         (KCF_PROV_OBJECT_OPS(pd) && \
1125         KCF_PROV_OBJECT_OPS(pd)->object_find_final) ? \
1126         KCF_PROV_OBJECT_OPS(pd)->object_find_final( \
1127             (pd)->pd_prov_handle, ppriv, req) : CRYPTO_NOT_SUPPORTED)

1129 /*
1130  * Wrappers for crypto_key_ops(9S) entry points.
1131  */
```

```
#define KCF_PROV_KEY_GENERATE(pd, session, mech, template, count, object, \
            req) ( \
        (KCF_PROV_KEY_OPS(pd) && KCF_PROV_KEY_OPS(pd)->key_generate) ? \
        KCF_PROV_KEY_OPS(pd)->key_generate((pd)->pd_prov_handle, \
            session, mech, template, count, object, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_KEY_GENERATE_PAIR(pd, session, mech, pub_template, \
            pub_count, priv_template, priv_count, pub_key, priv_key, req) ( \
        (KCF_PROV_KEY_OPS(pd) && KCF_PROV_KEY_OPS(pd)->key_generate_pair) ? \
        KCF_PROV_KEY_OPS(pd)->key_generate_pair((pd)->pd_prov_handle, \
            session, mech, pub_template, pub_count, priv_template, \
            priv_count, pub_key, priv_key, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_KEY_WRAP(pd, session, mech, wrapping_key, key, wrapped_key, \
            wrapped_key_len, req) ( \
        (KCF_PROV_KEY_OPS(pd) && KCF_PROV_KEY_OPS(pd)->key_wrap) ? \
        KCF_PROV_KEY_OPS(pd)->key_wrap((pd)->pd_prov_handle, \
            session, mech, wrapping_key, key, wrapped_key, wrapped_key_len, \
            req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_KEY_UNWRAP(pd, session, mech, unwrapping_key, wrapped_key, \
            wrapped_key_len, template, count, key, req) ( \
        (KCF_PROV_KEY_OPS(pd) && KCF_PROV_KEY_OPS(pd)->key_unwrap) ? \
        KCF_PROV_KEY_OPS(pd)->key_unwrap((pd)->pd_prov_handle, \
            session, mech, unwrapping_key, wrapped_key, wrapped_key_len, \
            template, count, key, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_KEY_DERIVE(pd, session, mech, base_key, template, count, \
            key, req) ( \
        (KCF_PROV_KEY_OPS(pd) && KCF_PROV_KEY_OPS(pd)->key_derive) ? \
        KCF_PROV_KEY_OPS(pd)->key_derive((pd)->pd_prov_handle, \
            session, mech, base_key, template, count, key, req) : \
        CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_KEY_CHECK(pd, mech, key) ( \
        (KCF_PROV_KEY_OPS(pd) && KCF_PROV_KEY_OPS(pd)->key_check) ? \
        KCF_PROV_KEY_OPS(pd)->key_check((pd)->pd_prov_handle, mech, key) : \
        CRYPTO_NOT_SUPPORTED)

/*
 * Wrappers for crypto_provider_management_ops(9S) entry points.
 *
 * ops_pd is the provider descriptor that supplies the ops_vector.
 * pd is the descriptor that supplies the provider handle.
 * Only ext_info needs two handles.
 */

#define KCF_PROV_EXT_INFO(ops_pd, provext_info, req, pd) ( \
        (KCF_PROV_PROVIDER_OPS(ops_pd) && \
        KCF_PROV_PROVIDER_OPS(ops_pd)->ext_info) ? \
        KCF_PROV_PROVIDER_OPS(ops_pd)->ext_info((pd)->pd_prov_handle, \
            provext_info, req) : CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_INIT_TOKEN(pd, pin, pin_len, label, req) ( \
        (KCF_PROV_PROVIDER_OPS(pd) && KCF_PROV_PROVIDER_OPS(pd)->init_token) ? \
        KCF_PROV_PROVIDER_OPS(pd)->init_token((pd)->pd_prov_handle, \
            pin, pin_len, label, req) : CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_INIT_PIN(pd, session, pin, pin_len, req) ( \
        (KCF_PROV_PROVIDER_OPS(pd) && KCF_PROV_PROVIDER_OPS(pd)->init_pin) ? \
        KCF_PROV_PROVIDER_OPS(pd)->init_pin((pd)->pd_prov_handle, \
```



```
            session, pin, pin_len, req) : CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_SET_PIN(pd, session, old_pin, old_len, new_pin, new_len, \
            req) ( \
        (KCF_PROV_PROVIDER_OPS(pd) && KCF_PROV_PROVIDER_OPS(pd)->set_pin) ? \
        KCF_PROV_PROVIDER_OPS(pd)->set_pin((pd)->pd_prov_handle, \
        session, old_pin, old_len, new_pin, new_len, req) : \
            CRYPTO_NOT_SUPPORTED)

/*
 * Wrappers for crypto_nostore_key_ops(9S) entry points.
 */

#define KCF_PROV_NOSTORE_KEY_GENERATE(pd, session, mech, template, count, \
            out_template, out_count, req) ( \
        (KCF_PROV_NOSTORE_KEY_OPS(pd) && \
            KCF_PROV_NOSTORE_KEY_OPS(pd)->nostore_key_generate) ? \
        KCF_PROV_NOSTORE_KEY_OPS(pd)->nostore_key_generate( \
            (pd)->pd_prov_handle, session, mech, template, count, \
            out_template, out_count, req) : CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_NOSTORE_KEY_GENERATE_PAIR(pd, session, mech, pub_template, \
            pub_count, priv_template, priv_count, out_pub_template, \
            out_pub_count, out_priv_template, out_priv_count, req) ( \
        (KCF_PROV_NOSTORE_KEY_OPS(pd) && \
            KCF_PROV_NOSTORE_KEY_OPS(pd)->nostore_key_generate_pair) ? \
        KCF_PROV_NOSTORE_KEY_OPS(pd)->nostore_key_generate_pair( \
            (pd)->pd_prov_handle, session, mech, pub_template, pub_count, \
            priv_template, priv_count, out_pub_template, out_pub_count, \
            out_priv_template, out_priv_count, req) : CRYPTO_NOT_SUPPORTED)

#define KCF_PROV_NOSTORE_KEY_DERIVE(pd, session, mech, base_key, template, \
            count, out_template, out_count, req) ( \
        (KCF_PROV_NOSTORE_KEY_OPS(pd) && \
            KCF_PROV_NOSTORE_KEY_OPS(pd)->nostore_key_derive) ? \
        KCF_PROV_NOSTORE_KEY_OPS(pd)->nostore_key_derive( \
            (pd)->pd_prov_handle, session, mech, base_key, template, count, \
            out_template, out_count, req) : CRYPTO_NOT_SUPPORTED)

/*
 * The following routines are exported by the kcf module (/kernel/misc/kcf)
 * to the crypto and cryptoadmin modules.
 */

/* Digest/mac/cipher entry points that take a provider descriptor and session */
extern int crypto_digest_single(crypto_context_t, crypto_data_t *,
    crypto_data_t *, crypto_call_req_t *);

extern int crypto_mac_single(crypto_context_t, crypto_data_t *,
    crypto_data_t *, crypto_call_req_t *);

extern int crypto_encrypt_single(crypto_context_t, crypto_data_t *,
    crypto_data_t *, crypto_call_req_t *);

extern int crypto_decrypt_single(crypto_context_t, crypto_data_t *,
    crypto_data_t *, crypto_call_req_t *);


/* Other private digest/mac/cipher entry points not exported through k-API */
extern int crypto_digest_key_prov(crypto_context_t, crypto_key_t *,
    crypto_call_req_t *);

/* Private sign entry points exported by KCF */
extern int crypto_sign_single(crypto_context_t, crypto_data_t *,
    crypto_data_t *, crypto_call_req_t *);
```

```
1264 extern int crypto_sign_recover_single(crypto_context_t, crypto_data_t *,
1265     crypto_data_t *, crypto_call_req_t *);

1267 /* Private verify entry points exported by KCF */
1268 extern int crypto_verify_single(crypto_context_t, crypto_data_t *,
1269     crypto_data_t *, crypto_call_req_t *);

1271 extern int crypto_verify_recover_single(crypto_context_t, crypto_data_t *,
1272     crypto_data_t *, crypto_call_req_t *);

1274 /* Private dual operations entry points exported by KCF */
1275 extern int crypto_digest_encrypt_update(crypto_context_t, crypto_context_t,
1276     crypto_data_t *, crypto_data_t *, crypto_call_req_t *);
1277 extern int crypto_decrypt_digest_update(crypto_context_t, crypto_context_t,
1278     crypto_data_t *, crypto_data_t *, crypto_call_req_t *);
1279 extern int crypto_sign_encrypt_update(crypto_context_t, crypto_context_t,
1280     crypto_data_t *, crypto_data_t *, crypto_call_req_t *);
1281 extern int crypto_decrypt_verify_update(crypto_context_t, crypto_context_t,
1282     crypto_data_t *, crypto_data_t *, crypto_call_req_t *);

1284 /* Random Number Generation */
1285 int crypto_seed_random(crypto_provider_handle_t provider, uchar_t *buf,
1286     size_t len, crypto_call_req_t *req);
1287 int crypto_generate_random(crypto_provider_handle_t provider, uchar_t *buf,
1288     size_t len, crypto_call_req_t *req);

1290 /* Provider Management */
1291 int crypto_get_provider_info(crypto_provider_id_t id,
1292     crypto_provider_info_t **info, crypto_call_req_t *req);
1293 int crypto_get_provider_mechanisms(crypto_minor_t *, crypto_provider_id_t id,
1294     uint_t *count, crypto_mech_name_t **list);
1295 int crypto_init_token(crypto_provider_handle_t provider, char *pin,
1296     size_t pin_len, char *label, crypto_call_req_t *);
1297 int crypto_init_pin(crypto_provider_handle_t provider, char *pin,
1298     size_t pin_len, crypto_call_req_t *req);
1299 int crypto_set_pin(crypto_provider_handle_t provider, char *old_pin,
1300     size_t old_len, char *new_pin, size_t new_len, crypto_call_req_t *req);
1301 void crypto_free_provider_list(crypto_provider_entry_t *list, uint_t count);
1302 void crypto_free_provider_info(crypto_provider_info_t *info);

1304 /* Administrative */
1305 int crypto_get_dev_list(uint_t *count, crypto_dev_list_entry_t **list);
1306 int crypto_get_soft_list(uint_t *count, char **list, size_t *len);
1307 int crypto_get_dev_info(char *name, uint_t instance, uint_t *count,
1308     crypto_mech_name_t **list);
1309 int crypto_get_soft_info(caddr_t name, uint_t *count,
1310     crypto_mech_name_t **list);
1311 int crypto_load_dev_disabled(char *name, uint_t instance, uint_t count,
1312     crypto_mech_name_t *list);
1313 int crypto_load_soft_disabled(caddr_t name, uint_t count,
1314     crypto_mech_name_t *list);
1315 int crypto_unload_soft_module(caddr_t path);
1316 int crypto_load_soft_config(caddr_t name, uint_t count,
1317     crypto_mech_name_t *list);
1318 int crypto_load_door(uint_t did);
1319 void crypto_free_mech_list(crypto_mech_name_t *list, uint_t count);
1320 void crypto_free_dev_list(crypto_dev_list_entry_t *list, uint_t count);
1321 extern void kcf_activate();

1323 /* Miscellaneous */
1324 int crypto_get_mechanism_number(caddr_t name, crypto_mech_type_t *number);
1325 int crypto_get_function_list(crypto_provider_id_t id,
1326     crypto_function_list_t **list, int kmflag);
1327 void crypto_free_function_list(crypto_function_list_t *list);
1328 int crypto_build_permitted_mech_names(kcf_provider_desc_t *,
1329     crypto_mech_name_t **, uint_t *, int);
```



```
1330 extern void kcf_init_mech_tabs(void);
1331 extern int kcf_add_mech_provider(short, kcf_provider_desc_t *,
1332     kcf_prov_mech_desc_t **);
1333 extern void kcf_remove_mech_provider(char *, kcf_provider_desc_t *);
1334 extern int kcf_get_mech_entry(crypto_mech_type_t, kcf_mech_entry_t **);
1335 extern kcf_provider_desc_t *kcf_alloc_provider_desc(crypto_provider_info_t *);
1336 extern void kcf_free_provider_desc(kcf_provider_desc_t *);
1337 extern void kcf_soft_config_init(void);
1338 extern int get_sw_provider_for_mech(crypto_mech_name_t, char **);
1339 extern crypto_mech_type_t crypto_mech2id_common(char *, boolean_t);
1340 extern void undo_register_provider(kcf_provider_desc_t *, boolean_t);
1341 extern void redo_register_provider(kcf_provider_desc_t *);
1342 extern void kcf_rnd_init();
1343 extern boolean_t kcf_rngprov_check(void);
1344 extern int kcf_rnd_get_pseudo_bytes(uint8_t *, size_t);
1345 extern int kcf_rnd_get_bytes(uint8_t *, size_t, boolean_t);
1346 extern int random_add_pseudo_entropy(uint8_t *, size_t, uint_t);
1347 extern void kcf_rnd_chpoll(short, int, short *, struct pollhead **);
1348 extern int crypto_uio_data(crypto_data_t *, uchar_t *, int, cmd_type_t,
1349     void *, void (*update)());
1350 extern int crypto_mblk_data(crypto_data_t *, uchar_t *, int, cmd_type_t,
1351     void *, void (*update)());
1352 extern int crypto_put_output_data(uchar_t *, crypto_data_t *, int);
1353 extern int crypto_get_input_data(crypto_data_t *, uchar_t **, uchar_t *);
1354 extern int crypto_copy_key_to_ctx(crypto_key_t *, crypto_key_t **, size_t *,
1355     int kmflag);
1356 extern int crypto_digest_data(crypto_data_t *, void *, uchar_t *,
1357     void (*update)(), void (*final)(), uchar_t);
1358 extern int crypto_update_iov(void *, crypto_data_t *, crypto_data_t *,
1359     int (*cipher)(void *, caddr_t, size_t, crypto_data_t *),
1360     void (*copy_block)(const uint8_t *, uint64_t *));
1359     void (*copy_block)(uint8_t *, uint64_t *));
1361 extern int crypto_update_uio(void *, crypto_data_t *, crypto_data_t *,
1362     int (*cipher)(void *, caddr_t, size_t, crypto_data_t *),
1363     void (*copy_block)(const uint8_t *, uint64_t *));
1362     void (*copy_block)(uint8_t *, uint64_t *));
1364 extern int crypto_update_mp(void *, crypto_data_t *, crypto_data_t *,
1365     int (*cipher)(void *, caddr_t, size_t, crypto_data_t *),
1366     void (*copy_block)(const uint8_t *, uint64_t *));
1365     void (*copy_block)(uint8_t *, uint64_t *));
1367 extern int crypto_get_key_attr(crypto_key_t *, crypto_attr_type_t, uchar_t **,
1368     ssize_t *);

1370 /* Access to the provider's table */
1371 extern void kcf_prov_tab_init(void);
1372 extern int kcf_prov_tab_add_provider(kcf_provider_desc_t *);
1373 extern int kcf_prov_tab_rem_provider(crypto_provider_id_t);
1374 extern kcf_provider_desc_t *kcf_prov_tab_lookup_by_name(char *);
1375 extern kcf_provider_desc_t *kcf_prov_tab_lookup_by_dev(char *, uint_t);
1376 extern int kcf_get_hw_prov_tab(uint_t *, kcf_provider_desc_t ***, int,
1377     char *, uint_t, boolean_t);
1378 extern int kcf_get_slot_list(uint_t *, kcf_provider_desc_t ***, boolean_t);
1379 extern void kcf_free_provider_tab(uint_t, kcf_provider_desc_t **);
1380 extern kcf_provider_desc_t *kcf_prov_tab_lookup(crypto_provider_id_t);
1381 extern int kcf_get_sw_prov(crypto_mech_type_t, kcf_provider_desc_t **,
1382     kcf_mech_entry_t **, boolean_t);

1384 extern kmutex_t prov_tab_mutex;
1385 extern boolean_t kcf_need_provtab_walk;
1386 extern int kcf_get_refcnt(kcf_provider_desc_t *, boolean_t);

1388 /* Access to the policy table */
1389 extern boolean_t is_mech_disabled(kcf_provider_desc_t *, crypto_mech_name_t);
1390 extern boolean_t is_mech_disabled_byname(crypto_provider_type_t, char *,
1391     uint_t, crypto_mech_name_t);
1392 extern void kcf_policy_tab_init(void);
```

```
1393 extern void kcf_policy_free_desc(kcf_policy_desc_t *);
1394 extern void kcf_policy_remove_by_name(char *, uint_t *, crypto_mech_name_t **);
1395 extern void kcf_policy_remove_by_dev(char *, uint_t, uint_t *,
1396     crypto_mech_name_t **);
1397 extern kcf_policy_desc_t *kcf_policy_lookup_by_name(char *);
1398 extern kcf_policy_desc_t *kcf_policy_lookup_by_dev(char *, uint_t);
1399 extern int kcf_policy_load_soft_disabled(char *, uint_t, crypto_mech_name_t *,
1400     uint_t *, crypto_mech_name_t **);
1401 extern int kcf_policy_load_dev_disabled(char *, uint_t, uint_t,
1402     crypto_mech_name_t *, uint_t *, crypto_mech_name_t **);
1403 extern void remove_soft_config(char *);

1405 #endif  /* _KERNEL */

1407 #ifdef  __cplusplus
1408 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2523 Thu Apr 30 20:52:32 2015
new/usr/src/uts/intel/kcf/Makefile
4896 Performance improvements for KCF AES modes
**********************************************************
```

```
#
# 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 2010 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
# Copyright 2015 by Saso Kiselkov. All rights reserved.
#
#       This makefile drives the production of the Kernel Cryptographic
#       Framework core module.
#
#       Intel implementation architecture dependent
#


#
#       Path to the base of the uts directory tree (usually /usr/src/uts).
#
UTSBASE = ../..
COM_DIR = $(COMMONBASE)/crypto

#
#       Define the module and object file sets.
#
MODULE          = kcf
LINTS           = $(KCF_OBJS:%.o=$(LINTS_DIR)/%.ln)
KCF_OBJS_32     =
KCF_OBJS_64     = gcm_intel.o gcm_intel_cryptogams.o
KCF_OBJS_64     = gcm_intel.o
KCF_OBJS        += $(KCF_OBJS_$(CLASS))
OBJECTS         = $(KCF_OBJS:%=$(OBJS_DIR)/%)
ROOTMODULE      = $(ROOT_MISC_DIR)/$(MODULE)

#
#       Include common rules.
#
include $(UTSBASE)/intel/Makefile.intel

#
#       Define targets
#
ALL_TARGET      = $(BINARY)
LINT_TARGET     = $(MODULE).lint
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE)
```



```
#
# Linkage dependencies
#

#
# lint pass one enforcement
#
CFLAGS          += $(CCVERBOSE) -I$(COM_DIR)
AS_CPPFLAGS     += -I../../$(PLATFORM)

LINTTAGS        += -I$(COM_DIR)

CERRWARN        += -_gcc=-Wno-switch
CERRWARN        += -_gcc=-Wno-uninitialized
CERRWARN        += -_gcc=-Wno-parentheses
CERRWARN        += -_gcc=-Wno-unused-label

#
#       Default build targets.
#
.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

#
#       Include common targets.
#
include $(UTSBASE)/intel/Makefile.targ

$(OBJS_DIR)/%.o: $(COM_DIR)/modes/amd64/%.s
        $(COMPILE.s) -o $@ $(COM_DIR)/modes/amd64/${@F:.o=.s}
        $(POST_PROCESS_O)

$(OBJS_DIR)/%.ln: $(COM_DIR)/modes/amd64/%.s
        @($(LHEAD) $(LINT.s) $(COM_DIR)/modes/amd64/${@F:.ln=.s} $(LTAIL))
```