SIMON and SPECK Implementation Guide

Ray Beaulieu
Douglas Shors
Jason Smith
Stefan Treatman-Clark
Bryan Weeks
Louis Wingers

National Security Agency
9800 Savage Road, Fort Meade, MD, 20755, USA

{rabeaul,djshors,jksmit,sgtreat,beweeks,lrwinge}@tycho.ncsc.mil

June 26, 2018

In this document we provide guidance for software developers. We include reference implementations, test vectors, and suggestions for improving performance on certain processors. We only consider the variants with 64- or 128-bit block sizes since these are probably the most useful variants of SIMON and SPECK for software applications.

The 64- and 128-bit block sized variants of both SIMON and SPECK have been incorporated into Crypto++ 6.1. The 128-bit block variants of SPECK are also included in the Linux kernel starting at v4.17. The developers have implementations which conform to the guidance offered here and their implementations can be used as a further check on accuracy (see [2] and [3]). For the sake of consistency and interoperability, we encourage developers to use the byte and word ordering described in this paper.

1 Some Definitions and Routines

The variants of SIMON and SPECK with a 64-bit block size operate on 32-bit words while the variants with a 128-bit block size operate on 64-bit words. So we need the following types for the C routines found in this paper.

```
#define u8 uint8_t
#define u32 uint32_t
#define u64 uint64_t
```

The round functions of the 32-bit (resp. 64-bit) variants of SIMON and SPECK require the use of bit rotations on 32-bit (resp. 64-bit) words. The rotations used for SIMON and SPECK, on any word size, never exceed 8 so the following definitions do not cause undefined behavior.

```
#define ROTL32(x,r) (((x)<<(r)) | (x>>(32-(r))))
#define ROTR32(x,r) (((x)>>(r)) | ((x)<<(32-(r))))
#define ROTL64(x,r) (((x)<<(r)) | (x>>(64-(r))))
#define ROTR64(x,r) (((x)>>(r)) | ((x)<<(64-(r))))
```

Because the natural word size of SIMON64 and SPECK64 (resp. SIMON128 and SPECK128) is 32-bits (resp. 64-bits), byte strings must be converted to the appropriate word size. For all variants of SIMON and SPECK, a little-endian byte and word ordering is followed. The following C routines convert bytes-to-words and words-to-bytes for 32-bit and 64-bit words.

For all of the conversion routines, we assume the number of bytes in the byte strings is a multiple of the block size (in bytes) and the number of words in the word strings is a multiple of the block size (in words). Otherwise, the routines might give unpredictable results.

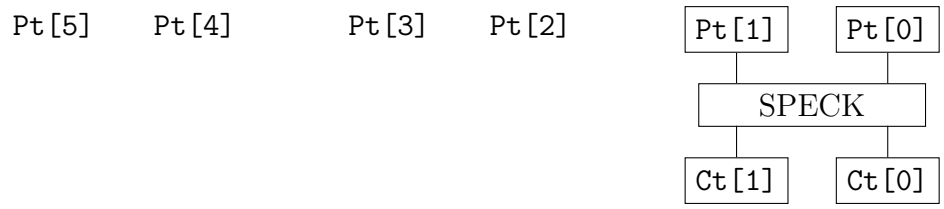
```
void Words32ToBytes(u32 words[],u8 bytes[],int numwords)
{
  int i,j=0;

  for(i=0;i<numwords;i++){
    bytes[j]=(u8)words[i];
    bytes[j+1]=(u8)(words[i]>>8);
    bytes[j+2]=(u8)(words[i]>>16);
    bytes[j+3]=(u8)(words[i]>>24);
    j+=4;
  }
}


void Words64ToBytes(u64 words[],u8 bytes[],int numwords)
{
  int i,j=0;

  for(i=0;i<numwords;i++){
    bytes[j]=(u8)words[i];
    bytes[j+1]=(u8)(words[i]>>8);
    bytes[j+2]=(u8)(words[i]>>16);
    bytes[j+3]=(u8)(words[i]>>24);
    bytes[j+4]=(u8)(words[i]>>32);
    bytes[j+5]=(u8)(words[i]>>40);
    bytes[j+6]=(u8)(words[i]>>48);
    bytes[j+7]=(u8)(words[i]>>56);
    j+=8;
  }
}
```

```
void BytesToWords32(u8 bytes[],u32 words[],int numbytes)
{
  int i,j=0;

  for(i=0;i<numbytes/4;i++){
    words[i]=(u32)bytes[j]  | ((u32)bytes[j+1]<<8)  | ((u32)bytes[j+2]<<16) |
      ((u32)bytes[j+3]<<24); j+=4;
  }
}


void BytesToWords64(u8 bytes[],u64 words[],int numbytes)
{
  int i,j=0;

  for(i=0;i<numbytes/8;i++){
    words[i]=(u64)bytes[j]  | ((u64)bytes[j+1]<<8)  | ((u64)bytes[j+2]<<16) |
      ((u64)bytes[j+3]<<24) | ((u64)bytes[j+4]<<32) | ((u64)bytes[j+5]<<40) |
      ((u64)bytes[j+6]<<48) | ((u64)bytes[j+7]<<56); j+=8;
  }
}
```

2 Byte and Word Ordering

The inputs to the SIMON 128 and SPECK 128 round functions and key schedules are 64-bit words. However, it is common practice to present the plaintext, ciphertext and key as a byte string. Therefore, it is necessary to convert strings of bytes to strings of words and strings of words to strings of bytes. The procedure is described using SPECK, though SIMON is similar.

Given an array of plaintext bytes, `pt`, the `BytesToWords64` routine creates an array of 64-bit words, `Pt`. The SPECK128/256 encryption operator acts on these words according to the diagram below, moving right to left, two words at a time. To be consistent with the diagram, given a block of plaintext bytes (i.e., 16 bytes), `pt`, the two 64-bit words of plaintext will be denoted `Pt=(Pt[1],Pt[0])` with `Pt[1]` the left word and `Pt[0]` the right word. `(Pt[1],Pt[0])` is a block of plaintext words and is the input to the SPECK round function.



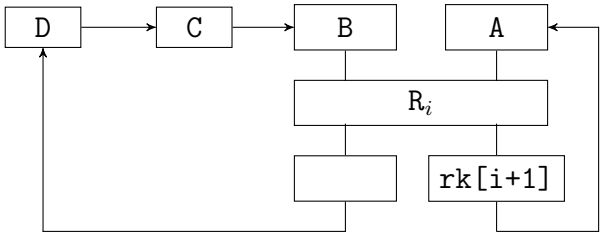
The following C routine encrypts a block of plaintext, given an array of round key words, `rk` (to be described later). The result is a block of ciphertext, `Ct=(Ct[1],Ct[0])`, consisting of two 64-bit words.

```
#define ER64(x,y,s) (x=(ROTR64(x,8)+y)^(s),y=ROTL64(y,3)^x)

void Speck128256Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<34;i++) ER64(Ct[1],Ct[0],rk[i]);
}
```

The SPECK 128/256 key schedule operates on four 64-bit words `(D,C,B,A)`, which are initialized to `(K[3],K[2],K[1],K[0])`, respectively. It uses the SPECK128/256 round function with a counter value being substituted for round key, i.e., it uses the function R_i, where

$$R_i(x, y) = ((ROTR64(x, 8) + y) \oplus i, ROTL64(y, 3) \oplus (ROTR64(x, 8) + y) \oplus i).$$

The first round key, `rk[0]`, is just the 64-bit word `K[0]`. The next round key, `rk[1]`, is the right output word of the round function $\mathtt{R}_0\mathtt{(K[1],K[0])}$. The following diagram illustrates the generation of the round keys.



Note that the four registers shown in the diagram are initialized, left to right, with the key words in the order `K[3],K[2],K[1],K[0]`. So we denote the key, `K=(K[3],K[2],K[1],K[0])`.

```
void Speck128256KeySchedule(u64 K[],u64 rk[])
{
  u64 i,D=K[3],C=K[2],B=K[1],A=K[0];

  for(i=0;i<33;i+=3){
    rk[i]=A;   ER64(B,A,i);
    rk[i+1]=A; ER64(C,A,i+1);
    rk[i+2]=A; ER64(D,A,i+2);
  }
  rk[33]=A;
}
```

In the SIMON and SPECK specification paper [1], the round keys were defined recursively in the following manner. Let $K = (K[3], K[2], K[1], K[0])$ be the key words. Let $\ell[0] = K[1], \ell[1] = K[2]$, and $\ell[2] = K[3]$. Then $rk[0] = K[0]$ and for $0 \leq i < 33$,

$$rk[i+1] = ROL64(rk[i], 3) \oplus \ell[i+3],$$

where $\ell[i+3] = (rk[i] + ROR64(\ell[i], 8)) \oplus i$.

This description of the key schedule is mathematically concise but for implementation purposes, the C routine provided earlier is prefered. We will provide C code for SIMON and SPECK key schedules for all variants with 64- and 128-bit blocks later.

Given an array of 16 plaintext bytes, `pt`, and an array of 32 key bytes, `k`, the array of 16 ciphertext bytes, `ct`, is computed as follows.

```
BytesToWords64(pt,Pt,16);
BytesToWords64(k,K,32);
Speck128256KeySchedule(K,rk);
Speck128256Encrypt(Pt,Ct,rk);
Words64ToBytes(Ct,ct,2);
```

Example. Consider the 16-byte ascii text, "pooner. In those".[1] Viewed as a binary string, this will be read in as a 16-byte hex array

```
pt = 70 6f 6f 6e 65 72 2e 20 49 6e 20 74 68 6f 73 65,
```

where `pt[0]=70, pt[1]=6f, pt[2]=6f, pt[3]=6e,` etc. Let us also assume a 32-byte (256-bit) key as the binary string

```
k = 00 01 02 03 04 05 06 06 08 09 0a 0b 0c 0d 0e 0f
                            10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

Here, `k[0]=00, k[1]=01, k[2]=02, k[3]=03, etc.`

To encrypt the plaintext `pt` with the key `k`, use the five subroutines provided above.

(1) `BytesToWords64(pt,Pt,16);`
`Pt=(Pt[1],Pt[0])=(65736f6874206e49,202e72656e6f6f70)`

(2) `BytesToWords64(k,K,32);`
`K=(K[3],K[2],K[1],K[0])`
`=(1f1e1d1c1b1a1918,1716151413121110,0f0e0d0c0b0a0908,0706050403020100)`

Note that these are the given plaintext and key words used for generating the SPECK 128/256 test vectors found in [1].

(3) `Speck128256KeySchedule(K,rk);`
`rk[0]=0706050403020100, rk[1]=37253b31171d0309, rk[2]=fe1588ce93d80d52,`
`rk[3]=e698e09f31334dfe, rk[4]=db60f14bcbd834fd, rk[5]=2dafa7c34cc2c2f8,`
`rk[6]=fbb8e2705e64a1db, rk[7]=db6f99e4e383eaef, rk[8]=291a8d359c8ab92d,`
`rk[9]=0b653abee296e282, rk[10]=604236be5c109d7f, rk[11]=b62528f28e15d89c,`
`etc.`

(4) `Speck128256Encrypt(Pt,Ct,rk);`
`Ct=(Ct[1],Ct[0])=(4109010405c0f53e,4eeeb48d9c188f43)`

These are the ciphertext words provided for the SPECK128/256 test vectors in [1].

(5) `Words64ToBytes(Ct,ct,2);`
`ct = 43 8f 18 9c 8d b4 ee 4e 3e f5 c0 05 04 01 09 41`

[1]This is a fragment of "Literally this word means Fat-Cutter; usage, however, in time made it equivalent to Chief Harpooner. In those days ..." — Moby Dick, Chapter 33.

3 SIMON64 Reference Code

Reference code for SIMON64/96, and SIMON64/128. For Feistel ciphers like SIMON, it is a common trick to encrypt two rounds at a time so as to avoid swapping words. The same sort of trick can be applied to the various key schedules, which we do.

```
#define f32(x) ((ROTL32(x,1) & ROTL32(x,8)) ^ ROTL32(x,2))
#define R32x2(x,y,k1,k2) (y^=f32(x), y^=k1, x^=f32(y), x^=k2)



void Simon6496KeySchedule(u32 K[],u32 rk[])
{
  u32 i,c=0xfffffffc;
  u64 z=0x7369f885192c0ef5LL;

  rk[0]=K[0]; rk[1]=K[1]; rk[2]=K[2];

  for(i=3;i<42;i++){
    rk[i]=c^(z&1)^rk[i-3]^ROTR32(rk[i-1],3)^ROTR32(rk[i-1],4);
    z>>=1;
  }
}



void Simon6496Encrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  u32 i;

  Ct[1]=Pt[1]; Ct[0]=Pt[0];
  for(i=0;i<42;) R32x2(Ct[1],Ct[0],rk[i++],rk[i++]);
}



void Simon6496Decrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  int i;

  Pt[1]=Ct[1]; Pt[0]=Ct[0];
  for(i=41;i>=0;) R32x2(Pt[0],Pt[1],rk[i--],rk[i--]);
}
```

```
void Simon64128KeySchedule(u32 K[],u32 rk[])
{
  u32 i,c=0xfffffffc;
  u64 z=0xfc2ce51207a635dbLL;

  rk[0]=K[0]; rk[1]=K[1]; rk[2]=K[2]; rk[3]=K[3];

  for(i=4;i<44;i++){
    rk[i]=c^(z&1)^rk[i-4]^ROTR32(rk[i-1],3)^rk[i-3]
      ^ROTR32(rk[i-1],4)^ROTR32(rk[i-3],1);
    z>>=1;
  }
}



void Simon64128Encrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  u32 i;

  Ct[1]=Pt[1]; Ct[0]=Pt[0];
  for(i=0;i<44;) R32x2(Ct[1],Ct[0],rk[i++],rk[i++]);
}



void Simon64128Decrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  int i;

  Pt[1]=Ct[1]; Pt[0]=Ct[0];
  for(i=43;i>=0;) R32x2(Pt[0],Pt[1],rk[i--],rk[i--]);
}
```

4 SIMON128 Reference Code

Reference code for Simon128/128, Simon128/192, and Simon128/256. For Feistel ciphers, it is a common trick to encrypt two rounds at a time so as to avoid swapping words. The same sort of trick can be applied to the various key schedules.

```
#define f64(x) ((ROTL64(x,1) & ROTL64(x,8)) ^ ROTL64(x,2))
#define R64x2(x,y,k1,k2) (y^=f64(x), y^=k1, x^=f64(y), x^=k2)


void Simon128128KeySchedule(u64 K[],u64 rk[])
{
  u64 i,B=K[1],A=K[0];
  u64 c=0xfffffffffffffffcLL, z=0x7369f885192c0ef5LL;

  for(i=0;i<64;){
    rk[i++]=A; A^=c^(z&1)^ROTR64(B,3)^ROTR64(B,4); z>>=1;
    rk[i++]=B; B^=c^(z&1)^ROTR64(A,3)^ROTR64(A,4); z>>=1;
  }

  rk[64]=A; A^=c^1^ROTR64(B,3)^ROTR64(B,4);
  rk[65]=B; B^=c^0^ROTR64(A,3)^ROTR64(A,4);
  rk[66]=A; rk[67]=B;
}


void Simon128128Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<68;i+=2) R64x2(Ct[1],Ct[0],rk[i],rk[i+1]);
}


void Simon128128Decrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=67;i>=0;i-=2) R64x2(Pt[0],Pt[1],rk[i],rk[i-1]);
}
```

```
void Simon128192KeySchedule(u64 K[],u64 rk[])
{
  u64 i,C=K[2],B=K[1],A=K[0];
  u64 c=0xfffffffffffffffcLL, z=0xfc2ce51207a635dbLL;

  for(i=0;i<63;){
    rk[i++]=A; A^=c^(z&1)^ROTR64(C,3)^ROTR64(C,4); z>>=1;
    rk[i++]=B; B^=c^(z&1)^ROTR64(A,3)^ROTR64(A,4); z>>=1;
    rk[i++]=C; C^=c^(z&1)^ROTR64(B,3)^ROTR64(B,4); z>>=1;
  }

  rk[63]=A; A^=c^1^ROTR64(C,3)^ROTR64(C,4);
  rk[64]=B; B^=c^0^ROTR64(A,3)^ROTR64(A,4);
  rk[65]=C; C^=c^1^ROTR64(B,3)^ROTR64(B,4);
  rk[66]=A; rk[67]=B; rk[68]=C;
}



void Simon128192Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i,t;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<68;i+=2) R64x2(Ct[1],Ct[0],rk[i],rk[i+1]);

  t=Ct[1]; Ct[1]=Ct[0]^f(Ct[1])^rk[68]; Ct[0]=t;
}



void Simon128192Decrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  int i;
  u64 t;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  t=Pt[0]; Pt[0]=Pt[1]^f(Pt[0])^rk[68]; Pt[1]=t;

  for(i=67;i>=0;i--) R64x2(Pt[0],Pt[1],rk[i],rk[i-1]);
}
```

```
void Simon128256KeySchedule(u64 K[],u64 rk[])
{
  u64 i,D=K[3],C=K[2],B=K[1],A=K[0];
  u64 c=0xfffffffffffffffcLL, z=0xfdc94c3a046d678bLL;

  for(i=0;i<64;){
    rk[i++]=A; A^=c^(z&1)^ROTR64(D,3)^ROTR64(D,4)^B^ROTR64(B,1); z>>=1;
    rk[i++]=B; B^=c^(z&1)^ROTR64(A,3)^ROTR64(A,4)^C^ROTR64(C,1); z>>=1;
    rk[i++]=C; C^=c^(z&1)^ROTR64(B,3)^ROTR64(B,4)^D^ROTR64(D,1); z>>=1;
    rk[i++]=D; D^=c^(z&1)^ROTR64(C,3)^ROTR64(C,4)^A^ROTR64(A,1); z>>=1;
  }

  rk[64]=A; A^=c^0^ROTR64(D,3)^ROTR64(D,4)^B^ROTR64(B,1);
  rk[65]=B; B^=c^1^ROTR64(A,3)^ROTR64(A,4)^C^ROTR64(C,1);
  rk[66]=C; C^=c^0^ROTR64(B,3)^ROTR64(B,4)^D^ROTR64(D,1);
  rk[67]=D; D^=c^0^ROTR64(C,3)^ROTR64(C,4)^A^ROTR64(A,1);
  rk[68]=A; rk[69]=B; rk[70]=C; rk[71]=D;
}



void Simon128256Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<72;i+=2) R64x2(Ct[1],Ct[0],rk[i],rk[i+1]);
}



void Simon128256Decrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=71;i>=0;i-=2) R64x2(Pt[0],Pt[1],rk[i],rk[i-1]);
}
```

5 SPECK64 Reference Code

Reference code for SPECK64/96, and SPECK64/128.

```
#define ER32(x,y,k) (x=ROTR32(x,8), x+=y, x^=k, y=ROTL32(y,3), y^=x)
#define DR32(x,y,k) (y^=x, y=ROTR32(y,3), x^=k, x-=y, x=ROTL32(x,8))



void Speck6496KeySchedule(u32 K[],u32 rk[])
{
  u32 i,C=K[2],B=K[1],A=K[0];

  for(i=0;i<26;){
    rk[i]=A; ER32(B,A,i++);
    rk[i]=A; ER32(C,A,i++);
  }
  rk[i]=A;
}



void Speck6496Encrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  u32 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<26;) ER32(Ct[1],Ct[0],rk[i++]);
}



void Speck6496Decrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=25;i>=0;) DR32(Pt[1],Pt[0],rk[i--]);
}
```

```
void Speck64128KeySchedule(u32 K[],u32 rk[])
{
  u32 i,D=K[3],C=K[2],B=K[1],A=K[0];

  for(i=0;i<27;){
    rk[i]=A; ER32(B,A,i++);
    rk[i]=A; ER32(C,A,i++);
    rk[i]=A; ER32(D,A,i++);
  }
}



void Speck64128Encrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  u32 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<27;) ER32(Ct[1],Ct[0],rk[i++]);
}



void Speck64128Decrypt(u32 Pt[],u32 Ct[],u32 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=26;i>=0;) DR32(Pt[1],Pt[0],rk[i--]);
}
```

6 SPECK128 Reference Code

Reference code for SPECK128/128, SPECK128/192, and SPECK128/256.

```
#define ER64(x,y,k) (x=ROTR64(x,8), x+=y, x^=k, y=ROTL64(y,3), y^=x)
#define DR64(x,y,k) (y^=x, y=ROTR64(y,3), x^=k, x-=y, x=ROTL64(x,8))



void Speck128128KeySchedule(u64 K[],u64 rk[])
{
  u64 i,B=K[1],A=K[0];

  for(i=0;i<31;){
    rk[i]=A;
    ER64(B,A,i++);
  }
  rk[i]=A;
}



void Speck128128Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<32;) ER64(Ct[1],Ct[0],rk[i++]);
}



void Speck128128Decrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=31;i>=0;) DR64(Pt[1],Pt[0],rk[i--]);
}
```

```
void Speck128192KeySchedule(u64 K[],u64 rk[])
{
  u64 i,C=K[2],B=K[1],A=K[0];

  for(i=0;i<32;){
    rk[i]=A; ER64(B,A,i++);
    rk[i]=A; ER64(C,A,i++);
  }
  rk[i]=A;
}



void Speck128192Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<33;) ER64(Ct[1],Ct[0],rk[i++]);
}



void Speck128192Decrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=32;i>=0;) DR64(Pt[1],Pt[0],rk[i--]);
}
```

```
void Speck128256KeySchedule(u64 K[],u64 rk[])
{
  u64 i,D=K[3],C=K[2],B=K[1],A=K[0];

  for(i=0;i<33;){
    rk[i]=A; ER64(B,A,i++);
    rk[i]=A; ER64(C,A,i++);
    rk[i]=A; ER64(D,A,i++);
  }
  rk[i]=A;
}



void Speck128256Encrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  u64 i;

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<34;) ER64(Ct[1],Ct[0],rk[i++]);
}



void Speck128256Decrypt(u64 Pt[],u64 Ct[],u64 rk[])
{
  int i;

  Pt[0]=Ct[0]; Pt[1]=Ct[1];
  for(i=33;i>=0;) DR64(Pt[1],Pt[0],rk[i--]);
}
```

7 Speck Encryption with on-the-fly Key Expansion

Speck lends itself to simple, on-the-fly key expansion for encryption. This does not work for decryption but in some modes, like counter mode, the decryption operator is not required. On more constrained devices, on-the-fly key expansion can reduce memory at the expense of reduced speed (since the key is expanded with each plaintext block).

```
void Speck128128EncryptExpandKey(u64 Pt[],u64 Ct[],u64 K[])
{
  u64 i,B=K[1],A=K[0];

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<32;){
    ER64(Ct[1],Ct[0],A); ER64(B,A,i++);
  }
}
```

```
void Speck128192EncryptExpandKey(u64 Pt[],u64 Ct[],u64 K[])
{
  u64 i,C=K[2],B=K[1],A=K[0];

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<32;){
    ER64(Ct[1],Ct[0],A); ER64(B,A,i++);
    ER64(Ct[1],Ct[0],A); ER64(C,A,i++);
  }
  ER64(Ct[1],Ct[0],A);
}
```

```
void Speck128256EncryptExpandKey(u64 Pt[],u64 Ct[],u64 K[])
{
  u64 i,D=K[3],C=K[2],B=K[1],A=K[0];

  Ct[0]=Pt[0]; Ct[1]=Pt[1];
  for(i=0;i<33;){
    ER64(Ct[1],Ct[0],A); ER64(B,A,i++);
    ER64(Ct[1],Ct[0],A); ER64(C,A,i++);
    ER64(Ct[1],Ct[0],A); ER64(D,A,i++);
  }
  ER64(Ct[1],Ct[0],A);
}
```

8 Processor Specific Tricks

In this section, we provide techniques to improve performance for some specific processors, including AI-32, ARM NEON and ARM ARMv8.

x86 SIMD Implementations

Many Intel x86 processors provide SIMD instructions which can be used to perform multiple Simon or Speck encryptions in parallel. This can greatly improve performance when the block ciphers are run in parallelizable modes, e.g., counter mode. Examples of SIMD instruction sets are SSE4, AVX2, and AVX512 which operate on 128-, 256- and 512-bit words, respectively. For our C implementations, good performance for long messages is obtained by performing $8N/bs$ parallel encryptions, where N is the SIMD register size (in bits) and bs is the block size (in bits).

Example. On an x86 processor which supports SSE4 with 128-bit registers, it is possible to obtain high-speed implementations of the 64-bit block sized variants of Simon or Speck by performing $8 \cdot 128/64 = 16$ parallel encryptions, or, for the 128-bit block sized variants, $8 \cdot 128/128 = 8$ parallel encryptions.

In the case of Simon128 or Speck128, eight plaintext blocks (128 bytes),

`(Pt[15],Pt[14]),...,(Pt[1],Pt[0])`

are loaded into eight 128-bit unsigned words,

`X[0],X[1],X[2],X[3]` and `Y[0],Y[1],Y[2],Y[3]`.

The `X` words hold the eight left 64-bit words of the eight plaintexts, i.e.,

`Pt[1],Pt[3],...,Pt[13],Pt[15]`

and the eight 128-bit `Y` words hold the eight right 64-bit words of plaintext, i.e.,

`Pt[0],Pt[2],...,Pt[12],Pt[14]`.

```
#define u8 uint8_t
#define u64 uint64_t
#define u128 __m128i
#define SET _mm_set_epi64x
#define LOW _mm_unpacklo_epi64
#define HIGH _mm_unpackhi_epi64
#define LD(ip) _mm_loadu_si128((u128 *)(ip))
#define ST(ip,X) _mm_storeu_si128((u128 *)(ip),X)

#define SL _mm_slli_epi64
#define SR _mm_srli_epi64
#define ADD _mm_add_epi64
#define XOR _mm_xor_si128
```

```
#define SHFL _mm_shuffle_epi8
#define R8 SET(0x080f0e0d0c0b0a09LL,0x0007060504030201LL)
#define ROR8(X) (SHFL(X,R8))
#define ROL(X,r) (XOR(SL(X,r),SR(X,(64-r))))

#define Rx8(X,Y,k) (X[0]=ADD(ROR8(X[0]),Y[0]), X[0]=XOR(X[0],k), \
                    Y[0]=ROL(Y[0],3), Y[0]=XOR(Y[0],X[0]), \
                    X[1]=ADD(ROR8(X[1]),Y[1]), X[1]=XOR(X[1],k), \
                    Y[1]=ROL(Y[1],3), Y[1]=XOR(Y[1],X[1]), \
                    X[2]=ADD(ROR8(X[2]),Y[2]), X[2]=XOR(X[2],k), \
                    Y[2]=ROL(Y[2],3), Y[2]=XOR(Y[2],X[2]), \
                    X[3]=ADD(ROR8(X[3]),Y[3]), X[3]=XOR(X[3],k), \
                    Y[3]=ROL(Y[3],3), Y[3]=XOR(Y[3],X[3]))


void Speck128128KeyScheduleSSE4(u64 K[],u128 rk[])
{
  u64 i,B=K[1],A=K[0];

  for(i=0;i<31;){
    rk[i]=SET(A,A);
    ER64(B,A,i++);
  }
  rk[31]=SET(A,A);
}
```

Below, we assume that the number of bytes to encrypt is a multiple of 128 bytes. If the plaintext byte string is not a multiple of 128 bytes, then one can encrypt eight blocks (128 bytes) at a time until the remainder of the byte string is less than 128 bytes. Then try to encrypt six blocks at a time, then four at a time and finally two blocks at a time. When the remainder is less than two blocks, it is probably faster just to use the regular 64-bit registers. Our SUPERCOP implementations [4] uses this strategy to obtain uniformly good performance.

```
void Speck128128EncryptSSE4(u8 pt[],u8 ct[],u8 k[],u64 len)
{
  u128 rk[32];

  BytesToWords64(k,K,16);
  Speck128128KeyScheduleSSE4(K,rk);

  while(len>=128){
    Encrypt8(pt,ct,rk);
```

```
    len-=128; pt+=128; ct+=128;
  }
}

void Encrypt8(u8 pt[],u8 ct[],u128 rk[])
{
  u128 X[4],Y[4];

  LOAD(pt,X,Y);

  for(i=0;i<32;i++) Rx8(X,Y,rk[i]);

  STORE(ct,X,Y);
}

void LOAD(u8 pt[],u128 X[],u128 Y[])
{
  u128 R,S;

  R=LD(pt);    S=LD(pt+16);  X[0]=HIGH(R,S); Y[0]=LOW(R,S);
  R=LD(pt+32); S=LD(pt+48);  X[1]=HIGH(R,S); Y[1]=LOW(R,S);
  R=LD(pt+64); S=LD(pt+80);  X[2]=HIGH(R,S); Y[2]=LOW(R,S);
  R=LD(pt+96); S=LD(pt+112); X[3]=HIGH(R,S); Y[3]=LOW(R,S);
}

void STORE(u8 ct[],u128 X[],u128 Y[])
{
  u128 R,S;

  R=LOW(Y[0],X[0]); ST(ct,R);    S=HIGH(Y[0],X[0]); ST(ct+16,S);
  R=LOW(Y[1],X[1]); ST(ct+32,R); S=HIGH(Y[1],X[1]); ST(ct+48,S);
  R=LOW(Y[2],X[2]); ST(ct+64,R); S=HIGH(Y[2],X[2]); ST(ct+80,S);
  R=LOW(Y[3],X[3]); ST(ct+96,R); S=HIGH(Y[3],X[3]); ST(ct+112,S);
}
```

It is possible to speed things up a little by reordering the instructions. This will probably be compiler dependent.

Much of the code is devoted to carrying out the left rotation by 3, which must be done using left and right end-off shifts since, for example, SSE4 and AVX2 do not have instructions for arbitrary rotations. However, Simon and Speck both have a rotation by 8 and this can be accomplished more efficiently using the "shuffle" command as done.

For best performance, the Simon ciphers (not Speck ciphers) can be bitsliced. This must be done carefully so as to avoid using too many SIMD registers which will slow down performance. We will not describe such implementations since they are somewhat complicated but refer to our SUPERCOP code where these sorts of implementations can be found.

Expected Performance

We have developed (for benchmarking purposes only) SIMD implementations of SIMON and SPECK for x86 processors supporting SSE4 and AVX2 SIMD instruction sets. They have been benchmarked using the SUPERCOP benchmarking toolkit [5] with Turbo Boost turned off.

On an Intel Xeon E5640, which supports SSE4, for long messages, parallel bitsliced implementations of SIMON 64/96, SIMON 64/128, SIMON 128/128, SIMON 128/192, and SIMON 128/256 reach speeds of 3.88 cycles/byte, 4.04 cycles/byte, 5.93 cycles/byte, 5.99 cycles/byte, and 6.31 cycles/byte, respectively.

For SPECK, the corresponding numbers are 2.41 cycles/byte, 2.50 cycles/byte, 2.98 cycles/byte, 3.07 cycles/byte, and 3.16 cycles/byte, respectively.

For comparison, AES-128, AES-192, and AES-256, using the hardware based AES-NI instruction reach speeds 1.17 cycles/byte, 1.38 cycles/byte, and 1.71 cycles/byte, respectively. So if AES-NI is supported, it is the best performer. Otherwise, either SIMON or SPECK will outperform the best AES implementations.

On a Corei7-4770, which supports AVX2, for long messages, parallel bitsliced implementations of SIMON 64/96, SIMON 64/128, SIMON 128/128, SIMON 128/192, and SIMON 128/256 reach speeds of 1.46 cycles/byte, 1.57 cycles/byte, 2.22 cycles/byte, 2.31 cycles/byte, and 2.37 cycles/byte, respectively.

For SPECK, the corresponding numbers are 1.04 cycles/byte, 1.09 cycles/byte, 1.28 cycles/byte, 1.31 cycles/byte, and 1.36 cycles/byte, respectively.

For comparison, AES-128, AES-192, and AES-256, using the hardware based AES-NI instruction reach speeds 0.66 cycles/byte, 0.75 cycles/byte, and 0.89 cycles/byte, respectively. So if AES-NI is supported, it is the best performer.

ARM Neon Implementations

Some ARM processors support the NEON instruction set which operates on 128-bit words in a manner similar to SSE4. Using these instructions, parallel implementations of SIMON and SPECK encryption can be developed. The same level of parallelization for SSE4 seems to work well here and the implementations are mostly similar.

The only performance tricks (besides bitslicing SIMON) that we make use of are for the rotations. For a rotation by eight, use the following code. Although it looks more complicated than a generic rotation (see below), we found it to be faster and it compiles to two instructions with GCC.

```
#define SET(a,b) vcombine_u64((uint64x1_t)(a),(uint64x1_t)(b))
#define tableR vcreate_u8(0x0007060504030201LL)
#define tableL vcreate_u8(0x0605040302010007LL)
```

```
#define ROR8(X) SET(vtbl1_u8((uint8x8_t)vget_low_u64(X),tableR),\
                    vtbl1_u8((uint8x8_t)vget_high_u6(X),tableR))
#define ROL8(X) SET(vtbl1_u8((uint8x8_t)vget_low_u64(X),tableL),\
                    vtbl1_u8((uint8x8_t)vget_high_u6(X),tableL))
```

If the rotation is not by eight, define the rotations as follows.

```
#define SL vshlq_n_u64
#define ROR(X,r) vsriq_n_u64(SL(X,(64-r)),X,r)
#define ROL(X,r) ROR(X,(64-r))
```

Expected Performance

We have developed (for benchmarking purposes only) SIMD implementations of SIMON and SPECK for ARM processors supporting the NEON instruction set. They have been benchmarked using the SUPERCOP benchmarking toolkit [5].

On a Samsung Exynos 5 Dual based on an ARM Cortex-A15, for long messages, a bitsliced implementation of SIMON64/96, SIMON64/128, SIMON128/128, SIMON128/192, and SIMON 128/256 reached speeds of 11.03 cycles/byte, 11.45 cycles/byte, 16.91 cycles/byte, 17.36 cycles/byte, and 17.96 cycles/byte, respectively.

The corresponding numbers for SPECK are 5.24 cycles/byte, 5.44 cycles/byte, 6.22 cycles/byte, 6.44 cycles/byte, and 6.63 cycles/byte, respectively.

When using the GCC compiler, significantly higher speed code might be obtained when using the `-Os` compiler option rather than the `-O3` option.

9 SIMON64/96 Test Vectors

```
pt = 63 6c 69 6e 67 20 72 6f
k = 00 01 02 03 08 09 0a 0b 10 11 12 13

BytesToWords32(pt,Pt,8);
Pt=(Pt[1],Pt[0])=(6f722067,6e696c63)

BytesToWords32(k,K,12);
K=(K[2],K[1],K[0])=(13121110,0b0a0908,03020100)

Simon6496KeySchedule(K,rk);

rk[0]=03020100          rk[11]=9e635793         rk[22]=0046cb1b         rk[32]=b4db50fe
rk[1]=0b0a0908          rk[12]=a6965478         rk[23]=59ce0704         rk[33]=3481f018
rk[2]=13121110          rk[13]=8b052e75         rk[24]=3dfb4191         rk[34]=ee1d573f
rk[3]=ffae9dce          rk[14]=884c5f47         rk[25]=cbd9e8cc         rk[35]=4806d097
rk[4]=c4facc91          rk[15]=d0e4e598         rk[26]=f3f75b6d         rk[36]=56feb8ff
rk[5]=c83d1bb6          rk[16]=e3e80363         rk[27]=a34520b7         rk[37]=0e529452
rk[6]=b5d510ff          rk[17]=35f020e1         rk[28]=ba7ae12d         rk[38]=d6d654a4
rk[7]=36e2c07c          rk[18]=1afa1c76         rk[29]=60e056a6         rk[39]=7eb6e8dd
rk[8]=72709043          rk[19]=bee71ed6         rk[30]=f6a8d0f4         rk[40]=8990d838
rk[9]=1343f40e          rk[20]=763d4d2a         rk[31]=943a89c1         rk[41]=b082bddc
rk[10]=ea417e40         rk[21]=0ca19efc

Simon6496Encrypt(Pt,Ct,rk);

Pt_1=(8283acb0,6f722067)      Pt_15=(445ffb2d,f7fe608e)      Pt_29=(4dd10cb7,9f33d106)
Pt_2=(6f728bad,8283acb0)      Pt_16=(3ede4de3,445ffb2d)      Pt_30=(5997a431,4dd10cb7)
Pt_3=(7eda965f,6f728bad)      Pt_17=(00c24cc4,3ede4de3)      Pt_31=(4e034cc6,5997a431)
Pt_4=(b3224320,7eda965f)      Pt_18=(0823de12,00c24cc4)      Pt_32=(f5a49ee5,4e034cc6)
Pt_5=(54e9564d,b3224320)      Pt_19=(3af138fa,0823de12)      Pt_33=(8c42426e,f5a49ee5)
Pt_6=(81e80db3,54e9564d)      Pt_20=(2c20531c,3af138fa)      Pt_34=(f02c63cb,8c42426e)
Pt_7=(e69c637d,81e80db3)      Pt_21=(fc0d3d88,2c20531c)      Pt_35=(82ae59ee,f02c63cb)
Pt_8=(a15b04da,e69c637d)      Pt_22=(d8ad33d3,fc0d3d88)      Pt_36=(b6cb7666,82ae59ee)
Pt_9=(5384e8f5,a15b04da)      Pt_23=(3fed7a5c,d8ad33d3)      Pt_37=(466b5c0f,b6cb7666)
Pt_10=(78038243,5384e8f5)     Pt_24=(138c899f,3fed7a5c)      Pt_38=(a9609a0f,466b5c0f)
Pt_11=(59c99fb8,78038243)     Pt_25=(482d0ea3,138c899f)      Pt_39=(75bf649c,a9609a0f)
Pt_12=(00d59361,59c99fb8)     Pt_26=(f8eb5a9e,482d0ea3)      Pt_40=(aa4f6893,75bf649c)
Pt_13=(fd8aa644,00d59361)     Pt_27=(b925ab8d,f8eb5a9e)      Pt_41=(111a8fc8,aa4f6893)
Pt_14=(f7fe608e,fd8aa644)     Pt_28=(9f33d106,b925ab8d)      Pt_42=(5ca2e27f,111a8fc8)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_42=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(5ca2e27f,111a8fc8)

WordsToBytes32(Ct,ct,2);
ct = c8 8f 1a 11 7f e2 a2 5c
```

10 SIMON64/128 Test Vectors

```
pt = 75 6e 64 20 6c 69 6b 65
k = 00 01 02 03 08 09 0a 0b 10 11 12 13 18 19 1a 1b

BytesToWords32(pt,Pt,8);
Pt=(Pt[1],Pt[0])=(656b696c,20646e75)

BytesToWords32(k,K,16);
K=(K[3],K[2],K[1],K[0])=(1b1a1918,13121110,0b0a0908,03020100)

Simon64128KeySchedule(K,rk);
```

rk[0]=03020100	rk[11]=dbf4a863	rk[22]=ab52df0a	rk[33]=4ce4d2ff
rk[1]=0b0a0908	rk[12]=cd0c28fc	rk[23]=247f66a8	rk[34]=32b7ebef
rk[2]=13121110	rk[13]=5cb69911	rk[24]=53587ca6	rk[35]=c47505c1
rk[3]=1b1a1918	rk[14]=79f112a5	rk[25]=d25c13f1	rk[36]=d0e929e8
rk[4]=70a011c3	rk[15]=77205863	rk[26]=4583b64b	rk[37]=8fe484b9
rk[5]=b770ec49	rk[16]=99880c12	rk[27]=7d9c960d	rk[38]=42054bee
rk[6]=57e3e835	rk[17]=1ce97c58	rk[28]=efbfc2f3	rk[39]=af77bae2
rk[7]=d397bc42	rk[18]=c8ed2145	rk[29]=89ed8513	rk[40]=18199c02
rk[8]=94dcf81f	rk[19]=b800dbb8	rk[30]=308dfc4e	rk[41]=719e3f1c
rk[9]=bf4b5f18	rk[20]=e86a2756	rk[31]=bf1a2a36	rk[42]=0c1cf793
rk[10]=8e5dabb9	rk[21]=7c06d4dd	rk[32]=e1499d70	rk[43]=15df4696

```
Simon64128Encrypt(Pt,Ct,rk);
```

Pt_1=(fc8b8a84,656b696c)	Pt_16=(153449bf,488c5c04)	Pt_31=(1e656c93,e203cf04)
Pt_2=(154d4e7f,fc8b8a84)	Pt_17=(a59de5fe,153449bf)	Pt_32=(00c4c678,1e656c93)
Pt_3=(b2a6be7c,154d4e7f)	Pt_18=(968b68b8,a59de5fe)	Pt_33=(fcbfe003,00c4c678)
Pt_4=(e0c1d225,b2a6be7c)	Pt_19=(3e5df649,968b68b8)	Pt_34=(07bf948c,fcbfe003)
Pt_5=(8083c368,e0c1d225)	Pt_20=(8b4e2236,3e5df649)	Pt_35=(dfe251dc,07bf948c)
Pt_6=(54bd334e,8083c368)	Pt_21=(fd0f5dcc,8b4e2236)	Pt_36=(1e0356a7,dfe251dc)
Pt_7=(2ca6a070,54bd334e)	Pt_22=(09690941,fd0f5dcc)	Pt_37=(770087a6,1e0356a7)
Pt_8=(35b04eec,2ca6a070)	Pt_23=(73f9a7c2,09690941)	Pt_38=(4de4cac3,770087a6)
Pt_9=(4efbefcf,35b04eec)	Pt_24=(0353b2e0,73f9a7c2)	Pt_39=(825e6641,4de4cac3)
Pt_10=(28f361c7,4efbefcf)	Pt_25=(2f4d70e4,0353b2e0)	Pt_40=(efcea9a5,825e6641)
Pt_11=(320b0062,28f361c7)	Pt_26=(202a8289,2f4d70e4)	Pt_41=(ebf45d9f,efcea9a5)
Pt_12=(3b2bc82c,320b0062)	Pt_27=(ea64cd8b,202a8289)	Pt_42=(e5c97bed,ebf45d9f)
Pt_13=(31e80836,3b2bc82c)	Pt_28=(b0eca9a9,ea64cd8b)	Pt_43=(b9dfa07a,e5c97bed)
Pt_14=(c03d61c5,31e80836)	Pt_29=(a6e0a8ce,b0eca9a9)	Pt_44=(44c8fc20,b9dfa07a)
Pt_15=(488c5c04,c03d61c5)	Pt_30=(e203cf04,a6e0a8ce)	

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_44=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(44c8fc20,b9dfa07a)

WordsToBytes32(Ct,ct,2);
ct = 7a a0 df b9 20 fc c8 44
```

11 SIMON128/128 Test Vectors

```
pt = 20 74 72 61 76 65 6c 6c 65 72 73 20 64 65 73 63
k = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

BytesToWords64(pt,Pt,16);
Pt=(Pt[1],Pt[0])=(6373656420737265,6c6c657661727420)

BytesToWords64(k,K,16);
K=(K[1],K[0])=(0f0e0d0c0b0a0908,0706050403020100)

Simon128128KeySchedule(K,rk);

rk[0]=0706050403020100          rk[34]=0722ac03cbc5dc93
rk[1]=0f0e0d0c0b0a0908          rk[35]=e80419b74902192d
rk[2]=79e8db8abd2c1f4c          rk[36]=9b5d9151a98a41d9
rk[3]=b852643a0882b4e9          rk[37]=bd258d7799540af7
rk[4]=2a984eb1a34b9d62          rk[38]=e8d487d7dcca7fd4
rk[5]=205716f8d920816f          rk[39]=914daa0fe0fe1d0c
rk[6]=c36822feca027aa4          rk[40]=4c1617c92125a259
rk[7]=0bf36f77133f166c          rk[41]=d3f1f675a9370c9f
rk[8]=7d568618a6a996f2          rk[42]=b4a8c99f316fccff
rk[9]=9c73282a727f4220          rk[43]=21d19c2043f3f633
rk[10]=98e02e60b03e7568         rk[44]=1d701c26c2d172a6
rk[11]=f91ed0ff90849426         rk[45]=7c576199087b30b0
rk[12]=d7ade68f44d95153         rk[46]=eac079f38ca65846
rk[13]=416e4db973add4e4         rk[47]=305c96c7beda61c1
rk[14]=e469341c0269c97d         rk[48]=2031ddb8ff6f4d9f
rk[15]=dc5a0784cc398ea0         rk[49]=d9a64f54d13e8356
rk[16]=0dd82b6b28d29fbf         rk[50]=69608f58d7a48a3d
rk[17]=32c37fa0e4510b50         rk[51]=5de3a9b5b94fa54e
rk[18]=f7538c9ac5e2519c         rk[52]=b8fd3f0ac4e67b3c
rk[19]=9ca3c905ef4c9b86         rk[53]=fe8c225bb262f267
rk[20]=b2f236d5d900f4a9         rk[54]=c73b869bf673f5aa
rk[21]=ce2d534df60375a4         rk[55]=f5e7151e8c344c75
rk[22]=982ab677475f52bb         rk[56]=d926ea9630c9c71e
rk[23]=fb55511b95629525         rk[57]=3cae591ad6de171b
rk[24]=870ab6ba735ad6b1         rk[58]=f2863bdb38801b71
rk[25]=2c3b535883c39d64         rk[59]=e2290223fdb9ea3f
rk[26]=bfb1961b14e16336         rk[60]=0f1e744287134769
rk[27]=6fc98705afee58cd         rk[61]=acc4d4107ad54958
rk[28]=3b4bc174041db25f         rk[62]=6fb55c7e701b452a
rk[29]=84ebbcc390d3ca5e         rk[63]=b836d5a7ac286a52
rk[30]=fc67f2dfb0f5084c         rk[64]=eccf146f00a33138
rk[31]=2b5ec24a623dc42f         rk[65]=d49c399343c9c09a
rk[32]=1446394da56e9377         rk[66]=e4eaaf3ba3196adf
rk[33]=476d9908f339008a         rk[67]=29b0397872648490

Simon128128Encrypt(Pt,Ct,rk);

Pt_1=(a4c3b5e2a3dfd8f7,6373656420737265)     Pt_35=(c58a465ddbe36e3e,905d296691164823)
Pt_2=(bef6dd63e39ea917,a4c3b5e2a3dfd8f7)     Pt_36=(e474253d14dff5b0,c58a465ddbe36e3e)
```

```
Pt_3=(523d392416a071cb,bef6dd63e39ea917)
Pt_4=(6a687dc991dd19c1,523d392416a071cb)
Pt_5=(91544922f3878aae,6a687dc991dd19c1)
Pt_6=(0f664dfb01e9b605,91544922f3878aae)
Pt_7=(69e9c7303fb12c14,0f664dfb01e9b605)
Pt_8=(62f13e6cdd320010,69e9c7303fb12c14)
Pt_9=(5e59d442dfd0ba87,62f13e6cdd320010)
Pt_10=(9f7547c840afad23,5e59d442dfd0ba87)
Pt_11=(8f2e6d03ec5d7966,9f7547c840afad23)
Pt_12=(549e213c3966be12,8f2e6d03ec5d7966)
Pt_13=(82db0f442f92c078,549e213c3966be12)
Pt_14=(1f9a559de6806b94,82db0f442f92c078)
Pt_15=(02cfe40d37fa335d,1f9a559de6806b94)
Pt_16=(cd7bca3f9f616c42,02cfe40d37fa335d)
Pt_17=(203af38742ed5d6c,cd7bca3f9f616c42)
Pt_18=(7f22fc80f5dd3aa2,203af38742ed5d6c)
Pt_19=(09a60d1f9941c63d,7f22fc80f5dd3aa2)
Pt_20=(c7151be27f14b5d8,09a60d1f9941c63d)
Pt_21=(a30a7607a832ad76,c7151be27f14b5d8)
Pt_22=(870594b939f82707,a30a7607a832ad76)
Pt_23=(2336bba478ad65d4,870594b939f82707)
Pt_24=(f6a30f7bef6fe552,2336bba478ad65d4)
Pt_25=(dfb62a16f88d648a,f6a30f7bef6fe552)
Pt_26=(1268e0500f996208,dfb62a16f88d648a)
Pt_27=(09647d4dcb2b8f8c,1268e0500f996208)
Pt_28=(5878dae98ede08fd,09647d4dcb2b8f8c)
Pt_29=(631c761de8460f7e,5878dae98ede08fd)
Pt_30=(54d2b275ff19e13a,631c761de8460f7e)
Pt_31=(4c9129febcf5818f,54d2b275ff19e13a)
Pt_32=(dce885791f732024,4c9129febcf5818f)
Pt_33=(83f40d4556779223,dce885791f732024)
Pt_34=(905d296691164823,83f40d4556779223)
Pt_37=(8f274be828a35944,e474253d14dff5b0)
Pt_38=(638607ea2e469adc,8f274be828a35944)
Pt_39=(6fefd99309fb59c1,638607ea2e469adc)
Pt_40=(82ad58a9fa0561d7,6fefd99309fb59c1)
Pt_41=(2c140dafc4cbbf44,82ad58a9fa0561d7)
Pt_42=(f1049327c98bd450,2c140dafc4cbbf44)
Pt_43=(5cafaee6509f2259,f1049327c98bd450)
Pt_44=(0b65f0de4926eb16,5cafaee6509f2259)
Pt_45=(6888b1b1b49ceaaf,0b65f0de4926eb16)
Pt_46=(550176a19b06f453,6888b1b1b49ceaaf)
Pt_47=(d64fb3c7522523a0,550176a19b06f453)
Pt_48=(30f06879494a1b51,d64fb3c7522523a0)
Pt_49=(55df9fda8a72135b,30f06879494a1b51)
Pt_50=(35b742c7a1bcd77e,55df9fda8a72135b)
Pt_51=(c9209e1ddb74eaaa,35b742c7a1bcd77e)
Pt_52=(4cd68f1e41c858da,c9209e1ddb74eaaa)
Pt_53=(d20a836e98a362fb,4cd68f1e41c858da)
Pt_54=(fa71a667b065e080,d20a836e98a362fb)
Pt_55=(8c55d8ebcf879552,fa71a667b065e080)
Pt_56=(2e497111854afb3b,8c55d8ebcf879552)
Pt_57=(a446f63ae0f48c86,2e497111854afb3b)
Pt_58=(c378d88010cede3e,a446f63ae0f48c86)
Pt_59=(5bf32fe19bd3d34d,c378d88010cede3e)
Pt_60=(fdbb24a691bb7d2e,5bf32fe19bd3d34d)
Pt_61=(1925c93879594ac2,fdbb24a691bb7d2e)
Pt_62=(15a1c4275e099f7e,1925c93879594ac2)
Pt_63=(0157859579774c04,15a1c4275e099f7e)
Pt_64=(aa4c06fd65b0c53c,0157859579774c04)
Pt_65=(00a8876f6f5661e6,aa4c06fd65b0c53c)
Pt_66=(7c732c9dcd00403e,00a8876f6f5661e6)
Pt_67=(65aa832af84e0bbc,7c732c9dcd00403e)
Pt_68=(49681b1e1e54fe3f,65aa832af84e0bbc)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_68=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(49681b1e1e54fe3f,65aa832af84e0bbc)

WordsToBytes64(Ct,ct,2);
ct = bc 0b 4e f8 2a 83 aa 65 3f fe 54 1e 1e 1b 68 49
```

12 SIMON128/192 Test Vectors

```
pt = 72 69 62 65 20 77 68 65 6e 20 74 68 65 72 65 20
k = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17

BytesToWords64(pt,Pt,16);
Pt=(Pt[1],Pt[0])=(206572656874206e,6568772065626972)

BytesToWords64(k,K,24);
K=(K[2],K[1],K[0])=(1716151413121110,0f0e0d0c0b0a0908,0706050403020100)

Simon128192KeySchedule(K,rk);

rk[0]=0706050403020100          rk[35]=d7fa476311c952df
rk[1]=0f0e0d0c0b0a0908          rk[36]=4a1d4de516b635b7
rk[2]=1716151413121110          rk[37]=0ec58c4f4de076ec
rk[3]=fb6a59083fae9dce          rk[38]=693151d1f354a490
rk[4]=c02a1c4270facc90          rk[39]=be578d3dc8161491
rk[5]=fcee08a785fd1bb7          rk[40]=dd15fac4d79c2acb
rk[6]=84c6876948b110fe          rk[41]=50bd4edadb211c99
rk[7]=07015b263298007d          rk[42]=feb40ff4c13fd937
rk[8]=7381c98edf786443          rk[43]=a2d7c43afc57d383
rk[9]=22715d3f815665cf          rk[44]=e13535e1d491642d
rk[10]=ee979aadc5585565         rk[45]=637e05e9191b1c0d
rk[11]=6f45bc8e44791444         rk[46]=27701b26b11afe3e
rk[12]=16924e99526129ff         rk[47]=3853c8c8965c6bf5
rk[13]=02d3086885cd9d38         rk[48]=680ebf837d4aa831
rk[14]=10cd12fa236381cc         rk[49]=e30ed8d11698fe46
rk[15]=aa78c2164bc49e27         rk[50]=75fd01a05a190424
rk[16]=72c463f4d7f6b8e3         rk[51]=5e1130528c57e70b
rk[17]=b64627440b1dc2a2         rk[52]=c2d212219028832b
rk[18]=282b9b7575e905a5         rk[53]=4e759d398ee1638e
rk[19]=7abc56d2d1eaf7f1         rk[54]=8cc725d9da9a22bf
rk[20]=7145970c83c1ccde         rk[55]=34797b38c92d9aab
rk[21]=fee88f1bd252df0e         rk[56]=6442da12e4a9f68c
rk[22]=b570303fe97a7e1c         rk[57]=39f4acc557ba7cfa
rk[23]=d3436df77f86db00         rk[58]=2f275b93c95e8d07
rk[24]=164b2b85b5a59623         rk[59]=0cabbb265ee83000
rk[25]=19221808fb6b6a46         rk[60]=c7549fec06760606
rk[26]=8e0af08910a2ff11         rk[61]=64477e6f7608d25a
rk[27]=c0952563f94419cd         rk[62]=1998dcf2b8b65893
rk[28]=82c6110d4428571e         rk[63]=6a01f64285145760
rk[29]=4981ac47939a8f7e         rk[64]=9058a03cf104e23d
rk[30]=12c2f5d08df11f29         rk[65]=8d69bd091679750a
rk[31]=ce4d9f15a2b69af5         rk[66]=6c85a5ccc943116c
rk[32]=535339ab8218ca73         rk[67]=24ffb1265b474efa
rk[33]=b2625e80ba6c753c         rk[68]=94464fc0475a17c7
rk[34]=6cd80ed241a22cfd
```

```
Simon128192Encrypt(Pt,Ct,rk);

Pt_1=(a3b9dff99790a9ca,206572656874206e)
Pt_2=(a0dfb91c3d1dcccd,a3b9dff99790a9ca)
Pt_3=(76683ea568fd026c,a0dfb91c3d1dcccd)
Pt_4=(ea053fc970455ce2,76683ea568fd026c)
Pt_5=(1a5c94d2991a1db7,ea053fc970455ce2)
Pt_6=(6b0964a583c40283,1a5c94d2991a1db7)
Pt_7=(32bf002edabb0646,6b0964a583c40283)
Pt_8=(83f43f606ab61fe6,32bf002edabb0646)
Pt_9=(4ac654613b173b1e,83f43f606ab61fe6)
Pt_10=(0e9813d911968058,4ac654613b173b1e)
Pt_11=(862180b8ba156f1b,0e9813d911968058)
Pt_12=(795bad85a9903274,862180b8ba156f1b)
Pt_13=(2778793e5e14eb5d,795bad85a9903274)
Pt_14=(ae1973484027561a,2778793e5e14eb5d)
Pt_15=(97e2e6e57dac3adf,ae1973484027561a)
Pt_16=(792eef83554b76d4,97e2e6e57dac3adf)
Pt_17=(23d0b818f5619d45,792eef83554b76d4)
Pt_18=(008a3895eb51c160,23d0b818f5619d45)
Pt_19=(09c3d0117d4e9d60,008a3895eb51c160)
Pt_20=(5eb92e22851c6311,09c3d0117d4e9d60)
Pt_21=(bb40ff92e2deddf9,5eb92e22851c6311)
Pt_22=(0dd3cd5218a8724a,bb40ff92e2deddf9)
Pt_23=(2afae8e549552ac9,0dd3cd5218a8724a)
Pt_24=(259bc2785250436c,2afae8e549552ac9)
Pt_25=(a1dccad1b5b1b55a,259bc2785250436c)
Pt_26=(fb4260975edcb6e1,a1dccad1b5b1b55a)
Pt_27=(80df390b42d1f00f,fb4260975edcb6e1)
Pt_28=(3993a3db2d7f6f12,80df390b42d1f00f)
Pt_29=(f774e44ee96a0979,3993a3db2d7f6f12)
Pt_30=(c921d62e594dd578,f774e44ee96a0979)
Pt_31=(c173657f013d6b72,c921d62e594dd578)
Pt_32=(00c596c7ff64b087,c173657f013d6b72)
Pt_33=(90b402441a37621d,00c596c7ff64b087)
Pt_34=(d077c55f19b749dd,90b402441a37621d)
Pt_35=(9d7613f20e00f807,d077c55f19b749dd)
Pt_36=(4055eff0307dfb11,9d7613f20e00f807)
Pt_37=(d69731f7b9ba31f5,4055eff0307dfb11)
Pt_38=(91ecc7c8a9452be8,d69731f7b9ba31f5)
Pt_39=(d8d4f785aff07a56,91ecc7c8a9452be8)
Pt_40=(dc4911e88ef282aa,d8d4f785aff07a56)
Pt_41=(7cf56a6353265a62,dc4911e88ef282aa)
Pt_42=(8e4bb6fd3f02d7fe,7cf56a6353265a62)
Pt_43=(b3f9d3596c177223,8e4bb6fd3f02d7fe)
Pt_44=(82a83f82632aecf0,b3f9d3596c177223)
Pt_45=(587c1ab13669754c,82a83f82632aecf0)
Pt_46=(b03e618dcbc46dd4,587c1ab13669754c)
Pt_47=(9f9506ab2c6aec80,b03e618dcbc46dd4)
Pt_48=(e33bbaeda4f73422,9f9506ab2c6aec80)
Pt_49=(7947371e83d8b47b,e33bbaeda4f73422)
Pt_50=(a72fb047bdbd73f9,7947371e83d8b47b)
Pt_51=(9e14b72d16469e1a,a72fb047bdbd73f9)
Pt_52=(954d70b36c7cf48c,9e14b72d16469e1a)
Pt_53=(01e3c6a56f6d4712,954d70b36c7cf48c)
Pt_54=(df717255136a894a,01e3c6a56f6d4712)
Pt_55=(c0836e2adadc4213,df717255136a894a)
Pt_56=(6803b99625361bae,c0836e2adadc4213)
Pt_57=(04ce4044a8a5fc6e,6803b99625361bae)
Pt_58=(4ace14c9d153fee8,04ce4044a8a5fc6e)
Pt_59=(84c5416126126288,4ace14c9d153fee8)
Pt_60=(5470aa6917d2c4ca,84c5416126126288)
Pt_61=(32f3373b7dabffb3,5470aa6917d2c4ca)
Pt_62=(9add229f3c225b7e,32f3373b7dabffb3)
Pt_63=(553d648915d4fc42,9add229f3c225b7e)
Pt_64=(8c49cfe9eecdbd13,553d648915d4fc42)
Pt_65=(fcc172d0927ff835,8c49cfe9eecdbd13)
Pt_66=(3327792295b318a6,fcc172d0927ff835)
Pt_67=(7a9111932ef0abc1,3327792295b318a6)
Pt_68=(6c9c8d6e2597b85b,7a9111932ef0abc1)
Pt_69=(c4ac61effcdc0d4f,6c9c8d6e2597b85b)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_69=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(c4ac61effcdc0d4f,6c9c8d6e2597b85b)

Words64ToBytes(Ct,ct,2);
ct = 5b b8 97 25 6e 8d 9c 6c 4f 0d dc fc ef 61 ac c4
```

13 SIMON128/256 Test Vectors

```
pt = 69 73 20 61 20 73 69 6d 6f 6f 6d 20 69 6e 20 74
k = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
                           10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

BytesToWords64(pt,Pt,16);
Pt=(Pt[1],Pt[0])=(74206e69206d6f6f,6d69732061207369)

BytesToWords64(k,K,24);
K=(K[3],K[2],K[1],K[0])
 =(1f1e1d1c1b1a1918,1716151413121110,0f0e0d0c0b0a0908,0706050403020100)

Simon128256KeySchedule(K,rk);
```

```
rk[0]=0706050403020100          rk[36]=a31b3abe8076f610
rk[1]=0f0e0d0c0b0a0908          rk[37]=3dfb6c156f4fe4b6
rk[2]=1716151413121110          rk[38]=623eba5ca89b0f90
rk[3]=1f1e1d1c1b1a1918          rk[39]=d1897281dd94af28
rk[4]=7262d303b0a011c3          rk[40]=e8cba626810ac017
rk[5]=b5069a3da370ec49          rk[41]=12b0ba3ed4576710
rk[6]=5588439d9423e835          rk[42]=27f1928653f59243
rk[7]=d45bd42be7d3bc42          rk[43]=6459d3334ca51ba4
rk[8]=95d63c18ff10381e          rk[44]=c61219ad9dd719bd
rk[9]=8e52a3d392ae0b18          rk[45]=3de57cab3bc05122
rk[10]=8d2efc1b34995478         rk[46]=ea19af8c9239f4fe
rk[11]=6dee39824d6ed866         rk[47]=8d5f96bebb02d0ea
rk[12]=08311575366dfffd         rk[48]=e3842f10f8d8e8e1
rk[13]=cb91edc3362f6aa1         rk[49]=7f473c0d0f8c33cd
rk[14]=0c0305c3f518042e         rk[50]=a60a99d3fa4cf6d9
rk[15]=3f7809565e38a75e         rk[51]=ce47ab7e80a4e231
rk[16]=fd8f7017ca8e416e         rk[52]=f9b3fd3cb773ef52
rk[17]=0e058c1d43bdbf5e         rk[53]=851a56bcaa80029b
rk[18]=f2181e83078f633c         rk[54]=b863f7519abd9a76
rk[19]=12adbc8dde87232f         rk[55]=085816bcb96d6fde
rk[20]=1808a9a2b12a4835         rk[56]=6055fc9debf7687e
rk[21]=886ffd8e453d7f85         rk[57]=b4b845e0615d31a2
rk[22]=7f9783dee5405671         rk[58]=363499ae8aa7c894
rk[23]=41553647fa3bbf54         rk[59]=627d313f6160c9f9
rk[24]=6790a0b8a992bb91         rk[60]=cb2611465fc12b50
rk[25]=8d475e5cb28981fa         rk[61]=73bfcc5aff52d55f
rk[26]=18eb4f6b40e06152         rk[62]=03c4cabf5497ed92
rk[27]=0861e447248dacce         rk[63]=537302d9f1650b97
rk[28]=72018cf92b796a3f         rk[64]=61e094b881ff9a96
rk[29]=7f46602d1a3e9419         rk[65]=2444879966513b02
rk[30]=53590cf77b30dfc4         rk[66]=603d6e7f1810a8e3
rk[31]=f3c1e06cfd629a16         rk[67]=a79858eadd19bc25
rk[32]=fc5f013613dd3135         rk[68]=46f321815d0b6faf
rk[33]=1a02056b802faef6         rk[69]=1729f70e33c7e362
rk[34]=846783a9cf1bf814         rk[70]=8801448e713eb772
rk[35]=d6841671bdbc0cc6         rk[71]=c46d2a8df85ca638
```

```
Simon128256Encrypt(Pt,Ct,rk);
```

```
Pt_1=(9aae8780a3dd8180,74206e69206d6f6f)
Pt_2=(35917d66e1906065,9aae8780a3dd8180)
Pt_3=(5add05ceb6ae5104,35917d66e1906065)
Pt_4=(d4fb7dd40c633d64,5add05ceb6ae5104)
Pt_5=(d226f1953786d594,d4fb7dd40c633d64)
Pt_6=(0d27a09f770d077e,d226f1953786d594)
Pt_7=(b130314373932e55,0d27a09f770d077e)
Pt_8=(3d9cf3bbddb456cb,b130314373932e55)
Pt_9=(caa460e14a12c473,3d9cf3bbddb456cb)
Pt_10=(1d1f12af77554cde,caa460e14a12c473)
Pt_11=(29e4f311e7d63b6f,1d1f12af77554cde)
Pt_12=(97a3e749634b1f0c,29e4f311e7d63b6f)
Pt_13=(5c1d33431e81b4b1,97a3e749634b1f0c)
Pt_14=(347485802e628628,5c1d33431e81b4b1)
Pt_15=(e14d20801297a02f,347485802e628628)
Pt_16=(ce380ed63f24a188,e14d20801297a02f)
Pt_17=(3c227fe3008a6762,ce380ed63f24a188)
Pt_18=(10f09e477eb4c15a,3c227fe3008a6762)
Pt_19=(ad781c7349970326,10f09e477eb4c15a)
Pt_20=(efad6347156de8e2,ad781c7349970326)
Pt_21=(86877ec985c2285d,efad6347156de8e2)
Pt_22=(78d1ac6e45586693,86877ec985c2285d)
Pt_23=(cbf604ea7dc36541,78d1ac6e45586693)
Pt_24=(805881d48b6a0c43,cbf604ea7dc36541)
Pt_25=(ad85a389fbfdef5e,805881d48b6a0c43)
Pt_26=(ba0a50bc23ff6e6e,ad85a389fbfdef5e)
Pt_27=(5d570e32738e7b2e,ba0a50bc23ff6e6e)
Pt_28=(d5399c524f530845,5d570e32738e7b2e)
Pt_29=(53a0e38677bb3087,d5399c524f530845)
Pt_30=(44bdf46220b15f43,53a0e38677bb3087)
Pt_31=(9b7e5ef98f0c904a,44bdf46220b15f43)
Pt_32=(ecd9d66bedf184ee,9b7e5ef98f0c904a)
Pt_33=(0dd42ea5fa97ba08,ecd9d66bedf184ee)
Pt_34=(d1a36cdd12aac238,0dd42ea5fa97ba08)
Pt_35=(6c7ac76a5f674aae,d1a36cdd12aac238)
Pt_36=(ee096d51f4c1600b,6c7ac76a5f674aae)
Pt_37=(7f441833cd143c97,ee096d51f4c1600b)
Pt_38=(6aea51cebff667ce,7f441833cd143c97)
Pt_39=(768367c8ec326236,6aea51cebff667ce)
Pt_40=(e06874ecc2cb445b,768367c8ec326236)
Pt_41=(5fb9fa9de711bbee,e06874ecc2cb445b)
Pt_42=(354db1868af9aaae,5fb9fa9de711bbee)
Pt_43=(e5efac098ea08701,354db1868af9aaae)
Pt_44=(0d26da91fcdfad0c,e5efac098ea08701)
Pt_45=(152e4ec339a42284,0d26da91fcdfad0c)
Pt_46=(4e361c36018f723e,152e4ec339a42284)
Pt_47=(d3e3a197aeb23acf,4e361c36018f723e)
Pt_48=(2f660ff810650c78,d3e3a197aeb23acf)
Pt_49=(cbf3a97737f6fbee,2f660ff810650c78)
Pt_50=(ec4ec40fa6db36c7,cbf3a97737f6fbee)
Pt_51=(9446289c1fe093a4,ec4ec40fa6db36c7)
Pt_52=(7319dd19797cbe64,9446289c1fe093a4)
Pt_53=(a183b9f53dd9e127,7319dd19797cbe64)
Pt_54=(730c1d597d3a3a60,a183b9f53dd9e127)
Pt_55=(d1c823f169bcf290,730c1d597d3a3a60)
Pt_56=(bc74c540f2d41ffc,d1c823f169bcf290)
Pt_57=(308fcaef8d13d9a4,bc74c540f2d41ffc)
Pt_58=(cbf92e93b5c7e8ce,308fcaef8d13d9a4)
Pt_59=(b87df82a93237282,cbf92e93b5c7e8ce)
Pt_60=(388bdf17ba686b3d,b87df82a93237282)
Pt_61=(906383184503e11e,388bdf17ba686b3d)
Pt_62=(2a391f2c53343808,906383184503e11e)
Pt_63=(2b511946796cecac,2a391f2c53343808)
Pt_64=(840e7ae4272a0927,2b511946796cecac)
Pt_65=(5290826e6e3b50a0,840e7ae4272a0927)
Pt_66=(6a08f088e1c6d0e4,5290826e6e3b50a0)
Pt_67=(9a9eae3333b01b9a,6a08f088e1c6d0e4)
Pt_68=(b3c6008cd21f10bb,9a9eae3333b01b9a)
Pt_69=(55758d9122d717e8,b3c6008cd21f10bb)
Pt_70=(d1b0d0e42f828428,55758d9122d717e8)
Pt_71=(3bf72a87efe7b868,d1b0d0e42f828428)
Pt_72=(8d2b5579afc8a3a0,3bf72a87efe7b868)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_72=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(8d2b5579afc8a3a0,3bf72a87efe7b868)
```

```
Words64ToBytes(Ct,ct,2);
ct = 68 b8 e7 ef 87 2a f7 3b a0 a3 c8 af 79 55 2b 8d
```

14 SPECK64/96 Test Vectors

```
pt = 65 61 6e 73 20 46 61 74
k = 00 01 02 03 08 09 0a 0b 10 11 12 13

BytesToWords32(pt,Pt,8);
Pt=(Pt[1],Pt[0])=(74614620,736e6165)

BytesToWords32(k,K,12);
K=(K[3],K[1],K[0])=(13121110,0b0a0908,03020100)

Speck6496KeySchedule(K,rk);

rk[0]=03020100          rk[9]=9812aac8          rk[18]=71a9351e
rk[1]=131d0309          rk[10]=16796373         rk[19]=8eff59e3
rk[2]=bbd80d53          rk[11]=ff72647b         rk[20]=498ff996
rk[3]=1a2370c1          rk[12]=ccda7364         rk[21]=15ec7c21
rk[4]=e45d26dd          rk[13]=d6f4b7c9         rk[22]=0f49104a
rk[5]=63cb3f1c          rk[14]=2589bf5a         rk[23]=d8ea21bc
rk[6]=27597d5a          rk[15]=39741c59         rk[24]=dcdb415c
rk[7]=205175b4          rk[16]=85a6aa9c         rk[25]=2fa7e901
rk[8]=db01db9f          rk[17]=208eb076

Speck6496Encrypt(Pt,Ct,rk);

Pt_1=(90e0c3ab,0b93c880)        Pt_14=(dfab738b,b1e1dca4)
Pt_2=(a439aa4a,f8a7ee4a)        Pt_15=(1848374d,9746d268)
Pt_3=(f8942aa7,3dab58f0)        Pt_16=(dd2b06c6,671d9582)
Pt_4=(ff809ddb,12da5a5a)        Pt_17=(a85c6a14,90b0c607)
Pt_5=(0a84fc2a,9c562efa)        Pt_18=(85d79207,0051a23b)
Pt_6=(a5ab8cea,471afb3e)        Pt_19=(767e4cd3,74f35d0b)
Pt_7=(1699db90,2e4e0262)        Pt_20=(c69682b4,610c6aef)
Pt_8=(9e35e989,ec45fa98)        Pt_21=(5c5cf8e7,543faf9c)
Pt_9=(aee5eb1e,ccca3fd9)        Pt_22=(2e7070b5,8f8d0c57)
Pt_10=(736b8f0c,153a71c2)       Pt_23=(4bf26c8d,379a0e31)
Pt_11=(37d4be22,9e073032)       Pt_24=(1c0c2121,a0dc50a8)
Pt_12=(3f4d608b,cf74e11f)       Pt_25=(1d231d95,1bc198d0)
Pt_13=(966e5d1b,edc955e5)       Pt_26=(9f7952ec,4175946c)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_26=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(9f7952ec,4175946c)

Words32ToBytes(Ct,ct,2);
ct = 6c 94 75 41 ec 52 79 9f
```

15 SPECK64/128 Test Vectors

```
pt = 2d 43 75 74 74 65 72 3b
k = 00 01 02 03 08 09 0a 0b 10 11 12 13 18 19 1a 1b

BytesToWords32(pt,Pt,8);
Pt=(Pt[1],Pt[0])=(3b726574,7475432d)

BytesToWords32(k,K,16);
K=(K[3],K[2],K[1],K[0])=(1b1a1918,13121110,0b0a0908,03020100)

Speck64128KeySchedule(K,rk);

rk[0]=03020100          rk[9]=03fa82c2          rk[18]=12d76c17
rk[1]=131d0309          rk[10]=313533ad         rk[19]=6eaccd6c
rk[2]=bbd80d53          rk[11]=dff70882         rk[20]=6a1ab912
rk[3]=0d334df3          rk[12]=9e487c93         rk[21]=10bc6bca
rk[4]=7fa43565          rk[13]=a934b928         rk[22]=6057dd32
rk[5]=67e6ce55          rk[14]=dd2edef5         rk[23]=d3c9b381
rk[6]=e98cb3d2          rk[15]=8be6388d         rk[24]=b347813d
rk[7]=aac76cbd          rk[16]=1f706b89         rk[25]=8c113c35
rk[8]=7f5951c8          rk[17]=2b87aaf8         rk[26]=fe6b523a

Speck64128Encrypt(Pt,Ct,rk);

Pt_1=(ebb2b492,4818adf9)          Pt_15=(4b0ec157,7687e119)
Pt_2=(c81963a4,88dc0c6e)          Pt_16=(4634d757,f20bdf9c)
Pt_3=(967c2882,d09c4bf6)          Pt_17=(56227ffa,c67c831d)
Pt_4=(5e0185ed,dae3da5b)          Pt_18=(eb550f64,d8b1178a)
Pt_5=(b7e5ee85,60fb3c5b)          Pt_19=(2f4b008e,eac3bcd8)
Pt_6=(8155ec1c,868c0ec7)          Pt_20=(165fcab4,40422c73)
Pt_7=(4a81d761,7ee1a15d)          Pt_21=(9e42352f,9c5356b5)
Pt_8=(4aeb4f89,bde64562)          Pt_22=(db4df320,39d7468c)
Pt_9=(38686179,d75a4a6c)          Pt_23=(3ae5494d,f45f7d2c)
Pt_10=(5368300f,e9ba6369)         Pt_24=(9253d1f4,30a83893)
Pt_11=(c838f834,85ebe37b)         Pt_25=(967d0d59,133cc9c0)
Pt_12=(654314f1,4a1c0f2d)         Pt_26=(e0c27af8,792434f8)
Pt_13=(a5c92ed2,f52957b8)         Pt_27=(8c6fa548,454e028b)
Pt_14=(6efb99ce,c7b12409)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_27=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(8c6fa548,454e028b)

Words32ToBytes(Ct,ct,2);
ct = 8b 02 4e 45 48 a5 6f 8c
```

16 SPECK128/128 Test Vectors

```
pt = 20 6d 61 64 65 20 69 74 20 65 71 75 69 76 61 6c
k = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

BytesToWords64(pt,Pt,16);
Pt=(Pt[1],Pt[0])=(6c61766975716520,7469206564616d20)

BytesToWords64(k,K,16);
K=(K[1],K[0])=(0f0e0d0c0b0a0908,0706050403020100)

Speck128128KeySchedule(K,rk);

rk[0]=0706050403020100          rk[11]=753e7a7c6660459e          rk[22]=d7987684a318b54a
rk[1]=37253b31171d0309          rk[12]=78d648a3a5b0e63b          rk[23]=a22c5282e600d319
rk[2]=f91d89cc90c4085c          rk[13]=87152b23cbc0a8d2          rk[24]=e029d67ebdf90048
rk[3]=c6b1f07852cc7689          rk[14]=a8ff8b8c54a3b6f2          rk[25]=67559234c84efdbf
rk[4]=014fcdf4f9c2d6f0          rk[15]=4873be3c43b3ea79          rk[26]=65173cf0cb01695c
rk[5]=b5fae1e4fe24cfd6          rk[16]=771ebffcbf05cb13          rk[27]=24cf1f1879819519
rk[6]=a36d6954b0737cfe          rk[17]=e8a6bcaf25863d20          rk[28]=38a36ed2dbafb72a
rk[7]=f511691ea02f35f3          rk[18]=e6c2ea8b5c520c93          rk[29]=ded93cfe31bae304
rk[8]=5374abb75a2b455d          rk[19]=4d71b5c1ac5214f5          rk[30]=c53d18b91770b265
rk[9]=8dd5f6204ddcb2a5          rk[20]=dc60b2ae253070dc          rk[31]=2199c870db8ec93f
rk[10]=b243d7c9869cac18         rk[21]=b01d0abbe1fb9741

Speck128128Encrypt(Pt,Ct,rk);

Pt_1=(93d384dfced4df85,309a87f4eddfb686)        Pt_17=(9779e1904fa59aa3,d3c722bc1288daa6)
Pt_2=(810b6048dab3886c,05df5fefb44e3c5d)        Pt_18=(9ff82032875ebd60,01c135d213186856)
Pt_3=(8b7de2836dece7b9,a5871dfecf9d0551)        Pt_19=(84a3c77919cdcb80,8aaa69e9810e8930)
Pt_4=(99a36b9901c684b1,b59b846f7d2eae3c)        Pt_20=(465eb871567a420e,130df73d5e0e0b8a)
Pt_5=(667aea2feff2a230,caa6c9540687d3d5)        Pt_21=(fd34e75bea54f510,655b5eb11a24a940)
Pt_6=(4ef7a5dac85309a1,1bc1ef7afc6d970f)        Pt_22=(c645992397f56974,ec9f6cab46d02377)
Pt_7=(1e7d8e74674696e6,c072f5a3842a2e9e)        Pt_23=(b6fdc4c0c970adaa,d206a19afff1b615)
Pt_8=(53801a2f58be40c7,5017b73379ef3431)        Pt_24=(de91cddd26bbf5db,4ea4c10ad9364575)
Pt_9=(441f9cfaf36cb72c,c4a225613c1516a6)        Pt_25=(caaa84a60ba40122,bf8c8cf0c2162a88)
Pt_10=(7d33b2de7ad431f8,582299d79a7c84ce)       Pt_26=(8502a541a06f3336,7966c2c7b0de6773)
Pt_11=(e2dc1a43fe6bf4e7,23c8d4ff2d8fd295)       Pt_27=(cafcf99c397fbffa,01caefa1bf8c8461)
Pt_12=(7e95cb6517ee7b17,60d36c9c7b90efbe)       Pt_28=(d85af38322479139,d60d8e8ede23b231)
Pt_13=(00844ac445183802,061f2e27999f45f1)       Pt_29=(37468750baea4ee8,872af3264bf7df66)
Pt_14=(8f0a99519624f6fb,bff3e86d5aded973)       Pt_30=(b1bb0553ad082ab0,88ec9c61f2b6d184)
Pt_15=(137d788af8d7489b,ece23be02e218306)       Pt_31=(fca34fde51136bcb,bbc7acd1c4a5e7ef)
Pt_16=(cf860764faa9b037,a897d8658ba5a800)       Pt_32=(a65d985179783265,7860fedf5c570d18)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_32=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(a65d985179783265,7860fedf5c570d18)

Words64ToBytes(Ct,ct,2);
ct = 18 0d 57 5c df fe 60 78 65 32 78 79 51 98 5d a6
```

17 SPECK128/192 Test Vectors

```
pt = 65 6e 74 20 74 6f 20 43 68 69 65 66 20 48 61 72
k = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17

BytesToWords64(pt,Pt,16);
Pt=(Pt[1],Pt[0])=(7261482066656968,43206f7420746e65)

BytesToWords64(k,K,24);
K=(K[2],K[1],K[0])=(1716151413121110,0f0e0d0c0b0a0908,0706050403020100)

Speck128128KeySchedule(K,rk);

rk[0]=0706050403020100          rk[11]=dac09cd5af5e76d6          rk[22]=efd149b0efbbf74e
rk[1]=37253b31171d0309          rk[12]=ca660ce33bd16b9e          rk[23]=02ca8ace24b0cbb4
rk[2]=fe1588ce93d80d52          rk[13]=1d3f349235549e8d          rk[24]=94968fa740eaf782
rk[3]=f788db953a2770c8          rk[14]=aca233ed2931350f          rk[25]=fca6ad548d13daff
rk[4]=ae96cb4f51692699          rk[15]=42e1dc29fc94bca2          rk[26]=fa1c1a8a0be7904f
rk[5]=792bb597b847397a          rk[16]=bd29d68e9dbdb77e          rk[27]=3594f90a254d56d0
rk[6]=9e6329125cfe47cc          rk[17]=751f72a5339f6f77          rk[28]=4913ea2b79da668d
rk[7]=a6698f90adc36bbb          rk[18]=49320cf569e3fc63          rk[29]=b0660f031a87ec17
rk[8]=f68cb29333796e9d          rk[19]=825e391774dd8c3f          rk[30]=3e7bbb3d40e4fc47
rk[9]=6187b7c3ae08eb15          rk[20]=49b1ca0af73ec529          rk[31]=4fa96d719e9eb338
rk[10]=ae70c68cb42115ea         rk[21]=22f3c83e0a8caec6          rk[32]=372dd2b830c61709

Speck128128Encrypt(Pt,Ct,rk);

Pt_1=(ac94d5b843d8d2ce,b597ae19407ba1e4)        Pt_18=(8fff8513d19f6a93,8b0801823ffc553b)
Pt_2=(b36179dfefa279bf,1fdc0915ec7f769a)        Pt_19=(57aa0df23a2e08c6,0fea01e3c5cca11a)
Pt_3=(219ae2415fb71441,df7aaaee3c4ca091)        Pt_20=(541f92e6ccdb431d,2b4f9df8e2be4bcd)
Pt_4=(d7149e45478b276d,2cc1c934a5ee23e3)        Pt_21=(011277813eb5e239,5b6e98462b47bc50)
Pt_5=(340e169dba5c8993,52005f38952d968a)        Pt_22=(b69c6283a60adcf4,6de8a0b2fc373e76)
Pt_6=(9c1fd8d88aa0ca69,0c1d211c23cc7e3b)        Pt_23=(8d4e74a59066be1c,e20b713271df4daf)
Pt_7=(ebda69e6a0a958c9,8b336107becaa911)        Pt_24=(fc52356933df7fd9,ec09bcfabd2512a6)
Pt_8=(f376b4e108a839d2,aaedbcdcfefd715e)        Pt_25=(5290809766b205a7,32dd67428f9a9090)
Pt_9=(8b6d8102d37f770a,dc0067e52494fdff)        Pt_26=(26895a97aa12986a,b0626083d6c61ceb)
Pt_10=(870c62a589609663,670f5d8cadc7799d)       Pt_27=(e094f3546597bfcc,6387f74ad3a75891)
Pt_11=(64e6af63e771cfd9,5c9c4306894a0332)       Pt_28=(05fc75340d41a680,19c3cf62907b620b)
Pt_12=(ecc1b560426f03d7,0823ad54083f1a45)       Pt_29=(d0da21fcbd52c53c,1ec45ae83e89d564)
Pt_13=(2a7663ea5350e2d6,6b6b094a12a830fe)       Pt_30=(ebf33a0921c0c43e,1dd1ed48d58e6f1e)
Pt_14=(5caa4b3fc9af1f6d,07f2016f5cee989e)       Pt_31=(62c65bbf9e54d3a5,8c4931f93227ab55)
Pt_15=(d9ec9857b58972b2,e67c932d52fdb642)       Pt_32=(7e0295256f58b310,1c4b1aecfe65e9bc)
Pt_16=(dbb7a3ec56278316,e8533a86c1ca3101)       Pt_33=(1be4cf3a13135566,f9bc185de03c1886)
Pt_17=(420724a4339deffa,009ef0923dcc67f5)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_33=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(1be4cf3a13135566,f9bc185de03c1886)

Words64ToBytes(Ct,ct,2);
ct = 86 18 3c e0 5d 18 bc f9 66 55 13 13 3a cf e4 1b
```

18 SPECK128/256 Test Vectors

```
pt = 70 6f 6f 6e 65 72 2e 20 49 6e 20 74 68 6f 73 65
k = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
                           10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

BytesToWords64(pt,Pt,16);
Pt=(Pt[1],Pt[0])=(65736f6874206e49,202e72656e6f6f70)

BytesToWords64(k,K,32);
K=(K[3],K[2],K[1],K[0])
 =(1f1e1d1c1b1a1918,1716151413121110,0f0e0d0c0b0a0908,0706050403020100)

Speck128256KeySchedule(K,rk);

rk[0]=0706050403020100          rk[12]=10419dd1d0b25f29          rk[23]=be77397e9de6bf31
rk[1]=37253b31171d0309          rk[13]=fd71e73b9c69fff6          rk[24]=35177f07af7d9479
rk[2]=fe1588ce93d80d52          rk[14]=8ea922047f976e93          rk[25]=b86971c5e7815ff0
rk[3]=e698e09f31334dfe          rk[15]=2e039afd398cffbc          rk[26]=7d77bfff103b45ea
rk[4]=db60f14bcbd834fd          rk[16]=9c9fcfef22c1072c          rk[27]=9983914c82a1a11e
rk[5]=2dafa7c34cc2c2f8          rk[17]=25fa8973ed55e6c9          rk[28]=1e88e9b26e3307f5
rk[6]=fbb8e2705e64a1db          rk[18]=69819861a6b4280c          rk[29]=7a0068774fc7061b
rk[7]=db6f99e4e383eaef          rk[19]=7b62d87498038f77          rk[30]=1771e55c7df2b16f
rk[8]=291a8d359c8ab92d          rk[20]=f2351ece62e296fe          rk[31]=a2cb5323bbf86418
rk[9]=0b653abee296e282          rk[21]=a6d382d176ba05ff          rk[32]=400303547ff5e38b
rk[10]=604236be5c109d7f         rk[22]=8d96e66745b78726          rk[33]=f4d26f589a56b276
rk[11]=b62528f28e15d89c

Speck128256Encrypt(Pt,Ct,rk);

Pt_1=(6e95e0d0d5e18ede,6fe673fba69af55f)          Pt_18=(d12e6fb3e76e2bb1,d2c5ac449d50cf49)
Pt_2=(797032ed606dd5e4,0643ad3054ba7f1f)          Pt_19=(ed1742d5f78c1578,7b3a20f11d0a6f36)
Pt_3=(14a895add1c2e1a6,26b5fc2f7411195e)          Pt_20=(8f45e0476b02743c,5694e7cf83510d8f)
Pt_4=(2a52445a10d191c1,1ffda521b0595b30)          Pt_21=(61113361a85e86fd,d5b60d1db2d6ea87)
Pt_5=(3a47062dc1b2183c,c5aa2f204378c1bc)          Pt_22=(75c49c8062c54cf2,d874f46df47218cc)
Pt_6=(2c4bd1e53df8b12c,011aa8e7263ebcca)          Pt_23=(477c5f6d3163593e,84dbfc0292f39f58)
Pt_7=(d6fe16c9551814a0,de2b51f064edf2f0)          Pt_24=(7d54411c9dc3bd80,5b8ba1080a5f4744)
Pt_8=(a46dc9e3cdc0e1eb,55374660eaaf766d)          Pt_25=(e91f8a4e89809f78,3542820edb7aa55a)
Pt_9=(69c1391f52f78e63,c07b0a18078c3d09)          Pt_26=(1642d05ccd857a09,bc56c02a165050d8)
Pt_10=(2881f1efc449d615,2b59a12ff8283e5b)         Pt_27=(b81abd05632693b8,5aacbc55d1a4157d)
Pt_11=(20c0159fbbfc154e,7a0d1ce07abde797)         Pt_28=(8ae7465e55a69d0e,5f82a4f0d88636e4)
Pt_12=(7e08f404946c3b30,ae6013074183078b)         Pt_29=(7085658558e8da74,8c9042039cd96d56)
Pt_13=(ce9f862a96a52cef,bd9f1e109abd10b2)         Pt_30=(7b00af1e6df5502b,1f82bf028b3e3a9f)
Pt_14=(501c5aad593a4a28,bce4aa298cd2cfbd)         Pt_31=(5d8c5aedd45e9e80,a199a2f98daf4a78)
Pt_15=(6b9de48045bb6494,8cb8b5cc232d1979)         Pt_32=(833c7c77c07bcd0e,8ff16bbbad019ecb)
Pt_16=(0f27c94d9afe2b61,6ae2672c8396e0ad)         Pt_33=(de77ab6c5b37f913,a1fcf6b1333b0f4f)
Pt_17=(576e411af3f0d9f4,007d787eef47dc9f)         Pt_34=(4109010405c0f53e,4eeeb48d9c188f43)
```

Here, `Pt_i` is the output after performing i rounds of encryption. `Pt_34=Ct`, is the pair of ciphertext words.

```
Ct=(Ct[1],Ct[0])=(4109010405c0f53e,4eeeb48d9c188f43)
```

```
Words64ToBytes(Ct,ct,2);
ct = 43 8f 18 9c 8d b4 ee 4e 3e f5 c0 05 04 01 09 41
```

Acknowledgement The authors appreciate the help provided by Jeff Walton (Crypto++) who provided assistance on the structure of this document, made suggestions for code improvements, and served as an independent reviewer.

References

[1] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan-Treatman Clark, Bryan Weeks, and Louis Wingers, *The* SIMON *and* SPECK *Families of Lightweight Block Ciphers*, Cryptology ePrint Archive, https:/www.eprint.iacr.org/2013/404.pdf, 2013

[2] Crypto++ Test Vectors, https://github.com/weidai11/cryptopp/tree/master/TestVectors.

[3] Eric Biggers, Linux Code, https://github.com/ebiggers/linux.

[4] Supercop Code, https://github.com/nsacyber/simon-speck-supercop.

[5] D.J. Bernstein and T. Lange. eBACS: ECRYPT benchmarking of cryptographic systems.