KRYPTOS RESEARCH
ADVANCING THE STATE OF SECURE COMPUTING



# A Brief Look At North Korean Cryptography

Jul 3, 2018 • Kryptos Logic

With much attention lately over North Korea and its evolving cybersecurity capabilities, we thought to cover a somewhat related topic. A couple of years back, the North Korean Red Star OS was described at the Chaos Computer Club conference. Among other things, they described the watermarking mechanism used by the OS to keep track of media files.

Along with the OS, three kernel modules were identified that appeared to contain homemade encryption algorithms specific to Red Star OS. We will name them after their kernel module names—Jipsam1, Jipsam2, and Pilsung. The former two are present in Red Star OS 2.0, whereas Pilsung is present only in Red Star OS 3.0. We are going to take a look at these, and comment on possible rationales for their design. We will only analyze the algorithms in isolation, as there is not a lot of information on how (or if) they are used. To our knowledge, this is the first time these algorithms are described.

## An AES Refresher

Since the ciphers described below are largely based around the AES structure, it might be worth to recall how the AES works. If you're familiar with the AES, you can safely skip ahead. We provide a simple implementation from the specification here. For a complete description, we refer to Wikipedia or FIPS-197.

The 128-bit block is laid out as a $4 \times 4$ grid, with the following byte order:

$$\begin{pmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{pmatrix}.$$

Encryption consists of a number of rounds, each of which consisting of the following sequence of operations:

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

### SubBytes

SubBytes is the most complex operation of AES, and the one that imbues it with any nonlinearity. Each byte of the state is replaced by the transformation

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix},$$

where the vector $v = [v_0, \ldots, v_7] = u^{254} \bmod (x^8 + x^4 + x^3 + x + 1)$ is the multiplicative inverse of the input $u$ (and 0 maps to 0). This operation is generally implemented as a lookup table, but the specifics of its structure will become important later.

### ShiftRows

ShiftRows does exactly what you think it does—it shifts the rows. In particular, row number $i$ is shifted by $i$ positions, as such:

$$\begin{pmatrix} 0 & 4 & 8 & 12 \\ 5 & 9 & 13 & 1 \\ 10 & 14 & 2 & 6 \\ 15 & 3 & 7 & 11 \end{pmatrix}.$$

### MixColumns

MixColumns is a little more complicated; it consists of a matrix-vector multiplication, in which each column is multiplied by the circulant matrix

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}.$$

Note that the elements of the above matrix are not integers. They are standins for polynomial multiplications over the same field used in SubBytes. *2* corresponds to multiplication by $x$ modulo $x^8 + x^4 + x^3 + x + 1$, and *3* corresponds to multiplication by $x + 1$ modulo $x^8 + x^4 + x^3 + x + 1$.

### AddRoundKey

AddRoundKey, once again as you would expect, xors the respective round key into the state.

## Jipsam1

Jipsam1 is just like AES-256, down to every last detail. In particular, the code used in the kernel module is clearly borrowed from an old Rijndael implementation by Brian Gladman, warts and all. Thus, it is a 128-bit block cipher with a 256-bit key.

The only difference between Jipsam1 and AES-256 is the S-box. Whereas in AES the S-box is public and constant, namely

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix},$$

where the vector $v = [v_0, \ldots, v_7] = u^{254} \bmod (x^8 + x^4 + x^3 + x + 1)$ is the multiplicative inverse of the input byte $u$. In Jipsam1, the constant component of the affine transformation changes from `0x63` = $[1, 1, 0, 0, 0, 1, 1, 0]$ to

$$c_r = ((k_r \oplus k_{r+3}) \wedge (k_{r+17} \oplus k_{r+15})) \oplus (k_{r+7} \wedge 15) \oplus (k_{r+11} \wedge 240)$$

at round $r$. This means that each round has its own custom S-box, and this also impacts the key schedule for each round key.

This does not improve security compared to the AES. Since only the affine component of the AES is affected, the nonlinearity and differential characteristics of the cipher are unchanged. It might be slightly stronger against integral attacks, but not by much. On the other hand, the scheme is now key-dependent and harder to analyze; there may be related-key attacks made possible by this tweak now. The AES with secret S-boxes has been studied recently, and shown to not be significantly superior to the AES with a fixed public S-box.

We have a simple implementation of Jipsam1, minimally changed from AES, here. The difference with respect to the AES code is essentially just the following:

```
--- a/rijndael.cpp
+++ b/jipsam1.cpp

-static uint8_t SubByte(const uint8_t x0) {
+static uint8_t SubByte(const jipsam1_ctx * ctx, size_t r, const uint8
+  const uint8_t   k = ((ctx->k[r+0] ^ ctx->k[r+3]) & (ctx->k[r+17] ^
+                 (ctx->k[r+7] & 0x0F) ^ (ctx->k[r+11] & 0xF0);
   const uint8_t  x1 = multiply( x0,  x0); // x^2
   const uint8_t  x2 = multiply( x1,  x0); // x^3
   const uint8_t  x3 = multiply( x2,  x2); // x^6
@@ -49,7 +51,7 @@ static inline uint8_t SubByte(const uint8_t x0) {
   const uint8_t x10 = multiply( x9,  x0); // x^127
   const uint8_t x11 = multiply(x10, x10); // x^254 = x^-1
   return x11 ^ rotate_left(x11, 1) ^ rotate_left(x11, 2) ^
-         rotate_left(x11, 3) ^ rotate_left(x11, 4) ^ 0x63;
+         rotate_left(x11, 3) ^ rotate_left(x11, 4) ^ k;
 }
```

## Jipsam2

With Jipsam2, the designers went in a different direction—instead of tweaking an existing algorithm, they went for composition and statefulness instead. The components used now are AES-256—unchanged—and RC4. It is implemented here.

The 256-bit key is split into two 128-bit halves. Each half is used to initialize two RC4 states $S_0$ and $S_1$, after which 32 bytes of keystream are discarded on both.

To encrypt a new 128-bit block, each state $S_i$ generates 32 new bytes of stream $s_0$ and $s_1$. The encryption then proceeds as

$$\text{AES-256}_{s_0}(\text{input}) \oplus s_1 \,.$$

In other words, Jipsam2 is a stateful mode of operation combining RC4 and AES, with each block using a fresh 256-bit AES key. It appears reasonably secure despite the use of RC4, though it's a rather convoluted way to achieve its goal. In particular, RC4 is

useless here, and everything could be accomplished using AES alone. It is possible the authors are hedging their bets against some unknown attack or backdoor on the AES.

## Pilsung

Pilsung is the newest and more complex algorithm found in Red Star OS 3.0. Like Jipsam1 it is a 128-bit block cipher, and like Jipsam1 it is based on the AES. But Pilsung makes more profound changes to the cipher than its predecessors. In fact, Pilsung somewhat resembles the "provably secure" cipher of Baignères and Finiasz, based on the insight of Baignères and Vaudenay that the AES with actual random S-boxes is provably secure against large classes of attacks, given enough rounds.

Pilsung goes further than having random S-boxes; it also randomizes the ShiftRows step of the AES with a random permutation of the 16-byte state. It is unclear what the purpose of this step is. It makes the scheme more akin to the SASAS scheme, in which both substitution and linear layers are secret.

Pilsung is implemented here. The encryption is fundamentally like the AES—a substitution-permutation network with 10 rounds of SubBytes followed by ShiftRows, MixColumns and AddRoundKey. The last round does not include MixColumns. As with the AES, we now describe each of these functions as implemented in Pilsung.

### SubBytes

Pilsung's SubBytes proceeds exactly like the AES. However, after the AES S-box is applied, the resulting byte goes through a random bit permutation, different for each byte of the state, followed by an xor with the constant 3.

### ShiftRows

Pilsung's ShiftRows does not shift rows at all (with high probability). Instead, it applies a random permutation to the 16 bytes of the state.

### MixColumns

MixColumns is unaltered compared to the AES.

### AddRoundKey

AddRoundKey is unaltered compared to the AES.

### Key Derivation

SubBytes and ShiftRows, as already mentioned, make use of a large number of permutations. In SubBytes's case, these are permutations of the 8 bits of each byte, and in ShiftRows, of the 16 bytes of the state. Where do these permutations come from? They are derived from the key.

While the Pilsung cipher itself is pretty straightforward, its key schedule is pretty elaborate. More so than necessary. Firstly, the 256-bit key is passed through a generic-

looking SHA-1-based key derivation function that derives 32 bytes of key material. Then, this key material goes through the AES key schedule, but mysteriously only 160 bits of key material (Nk = 5) are used here. (Is this a bug?) This naturally results in 11 * 16 = 176 bytes of expanded key.

The SHA-1 implementation for the key derivation above is slightly tweaked: bytes $4i + 3$ of the output digest are bitwise negated before returning.

### Generating Permutations

To generate permutations, Pilsung uses a relatively obscure method. Instead of using a Fisher-Yates shuffle, as one would expect, it uses a variant of the Rao-Sandelius shuffle, which closely resembles the radix sort. The idea is quite simple:

- Split the array into two groups, chosen at random using coin flips;
- Recursively split each of the resulting groups, until every group has a single element.

This technique has been rediscovered multiple times over time. One drawback of this method, as originally described, is that the groups are not necessarily balanced. One group may have more elements than another. This usually results in a necessary $n \log n + O(n)$ coin flips needed, instead of exactly $n \log n$. The Pilsung method, however, ensures that this number is exactly $n \log n$ by the "randomness" always having the same number of zeroes and ones.

Each permutation of 8 elements uses one byte of the AES key schedule mentioned above, and expands it into 24 bits of randomness as shown below.

```
// Distribution sort: sort array p of size n according to 0-1 array s
// Assumes s has n / 2 zeros and n / 2 ones
void Get_One(const uint8_t * s, uint8_t * p, size_t n, uint8_t * buf)
  size_t a = 0, b = 0;
  for(size_t i = 0; i < n; ++i) {
    if( s[i] )
      buf[n / 2 + a++] = p[i];
    else
      buf[b++] = p[i];
  }
  memcpy(p, buf, n);
}

// Generate a random permutation of [0..7]
void Get_P8forSEnc(uint8_t perm[8], uint8_t random_bits) {
  uint8_t coinflips[24];

  // Random 4-bit subset
  uint8_t v0 = Tree_Integer8[random_bits % 70];
  for(size_t i = 0; i < 8; ++i) {
```

```
    coinflips[i] = (v0 >> (7 - i)) & 1;
  }

  uint8_t v1 = (Tree_Integer4[(random_bits & 0x0F) % 6] << 0) |
               (Tree_Integer4[(random_bits >> 4) % 6] << 4);
  for(size_t i = 0; i < 8; ++i) {
    coinflips[i+8] = (v1 >> (7 - i)) & 1;
  }

  for(size_t i = 0; i < 8; i += 2) {
    if( random_bits & (3 << i) ) {
      coinflips[16+i+0] = 1;
      coinflips[16+i+1] = 0;
    } else {
      coinflips[16+i+0] = 0;
      coinflips[16+i+1] = 1;
    }
  }

  // Initialize permutation with identity
  for(size_t i = 0; i < 8; ++i)
    perm[i] = i;

  // Iterative version of the Rao-Sandelius shuffle
  uint8_t scratch[32];
  for(size_t i = 0; i < 3; ++i) {
    const size_t bins = 1 << i;       // number of subgroups
    const size_t size = 1 << (3 - i); // size of each subgroup
    for(size_t j = 0; j < bins; ++j) {
      Get_One(&coinflips[i * 8 + j * size], &perm[j * size], size, scr
    }
  }
}
```

`Tree_Integer{4,8}` above is simply a table with all possible combinations of balanced words of 4 and 8 bits respectively. There are $\binom{4}{2} = 6$ combinations for the former, and $\binom{8}{4} = 70$ of the latter. We can see that the choice is somewhat biased—given a random key byte, some combinations will be more common than others. Furthermore, the test `if( random_bits & (3 << i) )` is heavily biased towards true, and therefore the permutations obtained here won't be perfect. It would make much more sense to use a single bit for this purpose, and that is what the 16-byte permutation does.

The 16-byte permutations use a similar method, but instead use 16 bytes of AES key schedule material to generate 64 coin flips:

```
// Generate a random permutation of [0..15] at output
void Get_P16Enc(uint8_t output[16], const uint8_t input[16]) {
```

```
uint8_t coinflips[64];

// coin flips for first level
for(size_t i = 0; i < 4; ++i) {
  uint8_t v0 = Tree_Integer4[(input[i] ^ input[i+4]) % 6];
  for(size_t j = 0; j < 4; ++j) {
    coinflips[4*i+j] = (v0 >> (7 - j)) & 1;
  }
}

// coin flips for second level
for(size_t i = 0; i < 8; ++i) {
  if((input[i] >> i) & 1) {
    coinflips[16 + 2*i + 0] = 1;
    coinflips[16 + 2*i + 1] = 0;
  } else {
    coinflips[16 + 2*i + 0] = 0;
    coinflips[16 + 2*i + 1] = 1;
  }
}

// coin flips for third level
for(size_t i = 0; i < 4; ++i) {
  uint8_t v1 = Tree_Integer4[(input[i+8] ^ input[i+12]) % 6];
  for(size_t j = 0; j < 4; ++j) {
    coinflips[32+4*i+j] = (v1 >> (7 - j)) & 1;
  }
}

// coin flips for fourth level
for(size_t i = 0; i < 8; ++i) {
  if((input[8+i] >> i) & 1) {
    coinflips[48 + 2*i + 0] = 1;
    coinflips[48 + 2*i + 1] = 0;
  } else {
    coinflips[48 + 2*i + 0] = 0;
    coinflips[48 + 2*i + 1] = 1;
  }
}

// Initialize permutation with identity
for(size_t i = 0; i < 16; ++i)
  output[i] = i;

// Iterative version of the Rao-Sandelius shuffle
uint8_t scratch[32];
for(size_t i = 0; i < 4; ++i) {
  const size_t bins = 1 << i;       // number of subgroups
  const size_t size = 1 << (4 - i); // size of each subgroup
  for(size_t j = 0; j < bins; ++j) {
```

```
      Get_One(&coinflips[i * 16 + j * size], &output[j * size], size,
    }
  }
}
```

It would seem a lot simpler to use directly the 128 bits of key material to serve as coin flips.

### Security

This construction seems reasonably secure, considering the additions are largely done on top of a solid AES base. However, there are some concerning details:

- The permutations are not quite random, given the biases described above;
- The randomness is not very independent, as S-boxes and permutations all derived from the same key bits;
- Given that the S-box randomness is only partial, the Baignères-Vaudenay security result does not quite apply here;
- The random ShiftRows operation is unnecessary, and looks more like a liability than an asset. It can make weak classes of keys possible, by having permutations that do not change columns at all.

All in all, the designers would probably have been better off with Baignères and Finiasz's idea of using S-boxes of the form $A + B/x$, for random $A$ and $B \neq 0$, and touching nothing else. Using the AES key schedule seems pointless as well, given that the key is already derived using a SHA-1 based key derivation function.

## Summary

North Korea has, for some years now, become a notable player in the cybersecurity space. Intelligence agencies have attributed their efforts to the Sony hack, to WannaCry, to the quite recent TYPEFRAME and HIDDEN COBRA; they have proved to have serious capabilities in the field. From the little we have been able to gather, they also seem to have some cryptographic competence—their ciphers are not amateur hour, but neither are they very impressive in some sense, like, say, SIMON and SPECK. They mostly stick with simple variants of the AES, which is not really a bad idea.

In general, we observe that the ciphers are based on existing Western blocks, such as the AES, RC4, and SHA-1, without many innovations besides randomizing internal components. This may have two plausible reasons:

- this was expected to leak to the outside world, and they don't want to leak their capabilities with it, or
- there is not plenty of internal cryptographic research to work with.

We also point out that there was no care whatsoever in protecting these implementations against side-channel attacks. All the implementations are table-based,

with Pilsung consuming a lot of memory in key-derived tables.

We also note that other national ciphers in the region, such as South Korea's ARIA, China's SM4, Japan's Camellia, and Russia's Kuznyechik also borrow from AES in one way or another, though to a lesser extent than the ciphers described here.

---

---

Author | Kryptos Logic

Kryptos Logic is a company of computer security experts which develops cybersecurity solutions. The company's offerings have been developed from years of security and threat intelligence experience in numerous industries including including academic, government, and commercial. Kryptos Logic has developed widely used security products, publicly disclosed vulnerabilities, and regularly participates and is acknowledged by leading industry sponsored events and conferences.

---