Uvajda Stream Cipher

Karl Zander – pvial00@gmail.com

Abstract.

Uvajda is an encryption algorithm that obfuscates data so that no one is able to read it without the key. It is small and fast, maintaining a 512-bit internal key stream state and 256 bit key length.

1. Introduction

Uvajda is a fast encryption algorithm aimed at providing no attack vector other than brute force.

Uvajda consists of a key setup function which produces a 512-bit key stream state, and a key stream generator which outputs blocks of 8 pseudorandom bytes (64-bits) that are XORed with the input plaintext.

The cipher accepts a single key length, 256-bits. A nonce or initialization vector is required for encryption. The length of the nonce must be 128-bits.

2. Design goals

While designing Uvajda, I wanted to take advantage of the speed of Dark's design but add extra security. I developed the following requirements for Uvajda:

The algorithm must not use a counter in it's update function.

The algorithm must not directly expose the state/state.

The algorithm must operate on all 256 bits of the state/state per encryption round and all 256 bits of the key must be required to decrypt even a single byte of data.

The algorithm must employ a public nonce or initialization vector that must not give the attacker more than 50% advantage in the key stream generation process.

The algorithm must produce a non-repeating stream (without period) of bytes with uniform characteristics.

The algorithm must pass known statistical testing for random number generators. The algorithm must be extremely fast to implement in software.

3. Key Setup

Uvajda's key setup function prepares the 512-bit state for encryption. The state is represented as a 8 x 64-bit word array. First the key is loaded 8 bytes per state into the state array. Then the nonce is XORed with the first two words in the state. Next, J (a pseudorandom incrementor) is computed by summing J and each word of the state. Afterwards, the encryption update function is applied to the state twice and

J is computed once again by summing itself with the sum of every state word. Lastly, the update function is run 62 times.

To illustrate in more detail, see the following C code:

```
void uvajda_keysetup(struct uvajda_state *state, unsigned char *key, unsigned char *nonce) {
   memset(state->r, 0, 8*(sizeof(uint64_t)));
   uint64_t n[4];
   int i;
   int m = 0;
   int inc = 8;
   for (i = 0; i < (keylen / 8); i++) {
      state->r[i] = 0;
      state->r[i] = ((uint64_t)(key[m]) << 56) + ((uint64_t)key[m+1] << 48) + ((uint64_t)key[m+2] << 40) +
((uint64_t)key[m+3] << 32) + ((uint64_t)key[m+4] << 24) + ((uint64_t)key[m+5] << 16) + ((uint64_t)key[m+6] <<
8) + (uint64_t)key[m+7];
      m += inc;
   }

   n[0] = ((uint64_t)nonce[0] << 56) + ((uint64_t)nonce[1] << 48) + ((uint64_t)nonce[2] << 40) + ((uint64_t)nonce[3]
<< 32) + ((uint64_t)nonce[4] << 24) + ((uint64_t)nonce[5] << 16) + ((uint64_t)nonce[6] << 8) + (uint64_t)nonce[7];
   n[1] = ((uint64_t)nonce[8] << 56) + ((uint64_t)nonce[9] << 48) + ((uint64_t)nonce[10] << 40) +
((uint64_t)nonce[11] << 32) + ((uint64_t)nonce[12] << 24) + ((uint64_t)nonce[13] << 16) + ((uint64_t)nonce[14] <<
8) + (uint64_t)nonce[15];

   state->r[0] = state->r[0] ^ n[0];
   state->r[1] = state->r[1] ^ n[1];

   state->j = 0;

   for (int i = 0; i < 8; i++) {
      state->j = (state->j + state->r[i]);
   }
   for (int i = 0; i < 2; i++) {
      uvajda_F(state);
   }
   for (int i = 0; i < 8; i++) {
      state->j = (state->j + state->r[i]);
   }
   for (int i = 0; i < 62; i++) {
      uvajda_F(state);
   }
}
```

4. Encryption update function

The encryption round function F() permutes the state/state before processing the next 64-bit block of plaintext. This is done using the following steps:

First, we establish that we will run this function on every array word of the state. Second, we reserve a 512-bit space in memory where the a copy of the state is stored for later use. We also reserve a 64 bit temporary word for later use in the F() function. The following C code illustrates this:

```
uint64_t x;
```

```
uint64_t y[8];
for (i = 0; i < 8; i++) {
   y[i] = state->r[i];
}
```

Next, we take the first word of the state and save it to our temporary 64 bit word, x.

```
x = state→r[i];
```

Then the state word is added to the word to the right of it and J (the pseudorandom incrementer) mod 2^64.

```
state->r[i] = (state->r[i] + state->r[(i + 1) & 0x07] + state→j);
```

Next the word is XORed with the original value we saved in x.

```
state->r[i] = state->r[i] ^ x;
```

Then the word is rotated left 9 bits.

```
state->r[i] = rotateleft64(state->r[i], 9);
```

J is then recomputed by adding itself to the current state word.

```
state->j = (state->j + state→r[i]);
```

Lastly, each state word is added to itself prior to running the update function.

```
for (i = 0; i < 8; i++) {
   state->r[i] = state->r[i] + y[i];
}
```

5. Output function

Now that the state has been prepared by the round function, we are ready to encrypt data. To output 64-bits (8 bytes) from the state, we add states 0 and 6, the result is XORed with state 1, that result is added by the 5th state, that result is XORed with the 2nd state, that result is added to the 4th word, that result is XORed with the 3rd state and finally, the final result is added by state 7 mod 2^64.

The output function could best be expressed by the following code:

```
output = (((((((r[0] + r[6]) ^ r[1]) + r[5]) ^ r[2]) + r[4]) ^ r[3]) + r[7]);
```

The 64 output bits are unpacked to make 8 output bytes that XORed with the 8 input plaintext bytes. Extra unused bytes of the key stream are discarded once all input bytes have been processed.

6. Cryptanalysis

The output of Uvajda has not been able to be distinguished from a random bit sequence nor can it be differentiated from an ideal cipher. Nothing can be learned from frequency analysis.

Under a known plaintext attack of n bytes, one may not invert the output function to regain the key state bytes. One may know n bytes of the plaintext and the remainder of the message will remain secure.

7. Statistical Properties

Uvajda was subjected to the Diehard battery of tests (dieharder). Overall, Uvajda passed the tests. Tests were conducted on streams of 1 gigabyte of data with 100 different key, nonce pairs. Uvajda was also subjected to NIST Statistical Test Suite battery of tests and passed.