Ponic

by Peter Schmidt-Nielsen

Abstract:
Ponic is a stream cipher primitive that can be used to make a hash algorithm in an unusual way. The generalized structure of this stream cipher consists of six circular shift registers (CSR) with irregular clocking, and a non-linear function (NLF) that mixes data between the CSRs. The state function consists of clocking the first CSR, applying an extra clocking to either the next or the previous CSR based on the output, then advancing to the next CSR. In this manner each CSR is clocked anywhere from 1-3 times in a single round. This method of irregular clocking was chosen to achieve both irregularity and deterministic execution time. The non-linear function is a small Feistel cipher that operates with bits from each CSR as its input data. The overall design is highly partitioned so that each part could be shown to be strong, and the parts combined into a strong hash algorithm. The message to be hashed is broken down into 256-bit blocks that are processed consecutively. To process a block, Ponic XORs the block into CSRs 1 and 4, then performs 128 rounds of the state function. To extract the hashsum, Ponic merely runs the cipher and takes the keystream as the hashsum. A useful feature of this way of extracting the hashsum is that there is no upper or lower bound on the hashsum size, other than the size of the internal state (768 bits). This means that Ponic can give hashsums of any size between 1 and 768 bits.

Design:
Simple linear feedback shift register (LFSR) based stream ciphers with no irregularity in clocking can fall to a wide variety of attacks. To improve such structures, one can introduce irregular clocking, such as in the A5/1 algorithm. This can substantially improve the security of the structure, and even simple irregularity (e.g., the Shrinking Generator) can be quite effective. However, this still lacks substantial non-linearity in internal state. To achieve both non-linearity and irregularity, Ponic uses both a strong non-linear function (NLF) and a simple but effective irregular clocking scheme. With every iteration of the state function, each of Ponic's six circular shift registers (CSR) is stepped anywhere from 1-3 times, which introduces substantial irregularity, and contributes to Ponic's avalanche effect.

Ponic is designed as an ordered set of six 128-bit CSRs. From hereon, the individual CSRs will be referred to as csr(0) through csr(5), and let *a*, *b*, *c*, *d*, *e*, and *f* represent the low order bytes of csr(0), csr(1), ... csr(5), respectively. An iteration of the state function, from hereon referred to as a round, is executed as follows:

```
1. for each $n$ from 0 to 5
        a. csr($n$) is stepped, and the output saved.
        b. if the output is 1, csr($n$-1 modulo 6) is stepped, discarding the output.
        c. if the output is 0, csr($n$+1 modulo 6) is stepped, discarding the output.
2. $a, b, c, d, e, f$ = NLF($a, b, c, d, e, f$)
```

The NLF is performed thus:

```
Given six input bytes: $a, b, c, d, e, f$
Repeat the following 6 times:
        1. $a$ = sbox0[sbox1[a]^sbox2[b]^sbox3[c]^sbox4[d]^sbox5[e]^sbox6[f]]
        2. $a, b, c, d, e, f = b, c, d, e, f, a$
Return the six bytes: $a, b, c, d, e, f$ , and (sbox0[a+b+c+d+e+f]>>1)&1 as the keystream bit for this round.
```

Note: + represents addition modulo 256, ^ represents bitwise XOR operation, >> represents bitwise shift left operation, and & represents bitwise AND operation.

To initialize the internal state of Ponic, all of the 128-bit CSRs are set to have 255 in each byte (not zeroing them, "one-ing" them). Next the required hashsum bit length is XORed into the low order two bytes of csr(1). This is to ensure that PonicHash(*m*, 224) isn't just the first 224 bits of PonicHash(*m*, 512) (where PonicHash(*m*, *n*) is the *n*-bit hashsum of *m*).

To process a 256-bit block of data, the low order 128-bits are XORed into csr(0), and the high order 128-bits into csr(3). Next ROUNDS (see below about the ROUNDS constant) rounds are performed to diffuse this data into the internal state.

To finalize the internal state of Ponic and extract the hashsum, the last byte of any excess data is padded with a single one bit, then zeros to fill the excess block out to 256 bits. Next POST_ROUNDS rounds are performed to insure proper diffusion. (See below about ROUNDS and POST_ROUNDS.) Finally, one round is performed per bit in the desired hashsum bit length, and the hashsum is given by the output keystream bits of the NLF during these rounds.

Security Parameters:
Ponic has two tunable security parameters: ROUNDS, and POST_ROUNDS.

As stated previously, after XORing a new block into csr(0) and csr(3), ROUNDS iterations of the state function are performed. The purpose of this is to to diffuse bits before the next round. I recommend a value of 128 for all digest lengths. If any fewer than 120 rounds are used, there is a slight chance (2^{-240}), increasing as the number of rounds decrease, that not all bits will have entered the NLF. Using 128 rounds guarantees both: a) that all bits enter the NLF, and b) on average all bits go through the NLF twice.

Also as stated previously, after the last block is processed, POST_ROUNDS iterations of the state function are performed. I recommend a value of 256 for POST_ROUNDS for all digest lengths. To weaken this algorithm for cryptanalysis, one may reduce POST_ROUNDS to lower values. However, because each block does cause 128 rounds to be performed, a large number of post rounds may not be necessary to guarantee diffusion and security. Nevertheless, to protect against advances in cryptanalysis, it is prudent to set POST_ROUNDS to 256 for all digest lengths. Since 256 rounds are relatively inexpensive, and become trivial when hashing large messages, this decision costs very little, and presumably improves security greatly.

Design rationale:
When designing an NLF, one would wish it to be as close to a random oracle as possible. For the task of designing an 8-bit NLF, an 8-bit lookup table mapping all inputs to outputs (an sbox) is both practical, and, assuming the sbox was randomized appropriately, optimal. However, this design of building a lookup table of inputs to outputs doesn't scale, as it takes 2^n entries in an *n*-bit sbox. To get around this problem, larger NLFs are built out of numerous smaller non-linear components in the hope that no hidden linearity or other weakness will accidentally be introduced. Both SHA-0 and SHA-1 fell to such attacks based on such weaknesses. To avoid the same issue in Ponic, the NLF was designed to have no complicated bit shuffling, but merely a simple compression function of five bytes, used as the Feistel function in a small Feistel cipher. This is guaranteed to have every bit in all 6 bytes dependent on every other bit in just 6 rounds. By simplifying the domain of the NLF function in Ponic to just 6 bytes, it came within reach of using sboxes. However, this alone wouldn't be sufficient as a hash algorithm, as it would only have 96 bits of internal state which is woefully inadequate in modern times. Therefore, this small function was combined with a large internal state of six 128-bit CSRs.

This implementation of Ponic uses 7 sboxes, referred to as sbox0 through sbox6. Sbox0 is the Rijndael sbox, and sboxes 1 through 6 were randomized with the following Python program using Python 2.5.2's Mersenne Twister. The random seed was chosen to be obviously arbitrary:

```
import random
random.seed(123456789)
for i in xrange(6):
    sbox = range(256)
    random.shuffle(sbox)
    print sbox
```

The exact constants for the sboxes can be found in the Constants section on the final page. Hashing an empty string using 128 ROUNDS, 256 POST_ROUNDS (see below), and a 512-bit hashsum accesses every table entry. No input has been found which does not access all entries.

Ponic is designed to have a very strong avalanche effect after just a few rounds. To facilitate this, the state function is highly data dependent, with irregular clocking based on the output of the CSRs. This particular combination of NLF and irregular clocking results in almost complete diffusion in as few as 64 rounds. In a round, each CSR is stepped on average 2 times. Therefore 128 rounds is sufficient to allow each bit in the CSRs to go through the NLF twice. To show that diffusion is sufficient after 128 rounds, imagine two Ponic ciphers run concurrently, with a one bit difference in the low order bits of one CSR. At first, no diffusion occurs, because the different bit is not being used in the NLF. However, once the different bit enters the NLF, it will cause all bytes in all six CSRs to differ in 50% of their constituent bits. Next this will cause the clocking between the two algorithms to differ. This method of irregular clocking combined with an NLF results in a very strong avalanche, but only if each bit goes through the NLF. Figure 1 shows the XOR of the internal states of two concurrently running Ponic ciphers, with a one bit difference in one CSR. As the result is an XOR of the two, a black pixel corresponds to a bit that differs, and a white pixel corresponds to identical bits. Each column is a CSR, with each row representing the XOR of the states of the two CSRs. Red lines separate the CSRs.

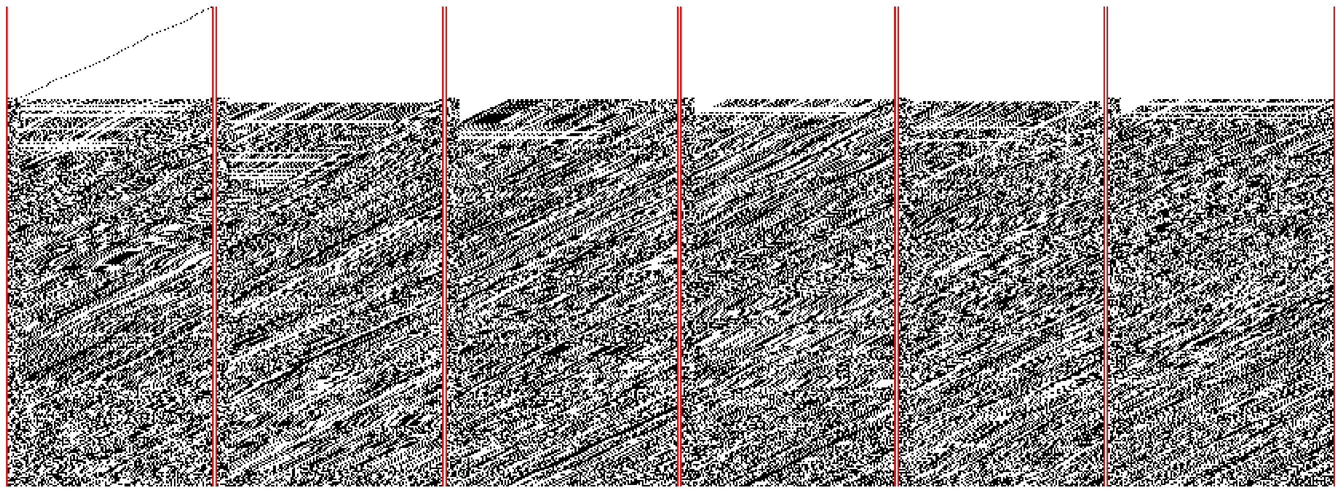

Fig 1: A single bit has an almost instant avalanche effect once it enters the NLF

The single differing bit wobbles (due to the irregular clocking) until it enters the NLF. On average, within 64 rounds every bit has interacted with every other bit in internal state. Imagine Alice is an observer who can only see one of the two ciphers and knows that one bit is different from her cipher to the hidden other cipher. From this perspective, each round after the unknown bit enters the NLF introduces at least 12 new unknown bits — the bits that the NLF outputs, that then get shifted out.

Next, the positions of the remaining bits will be unknown because the clocking is now unknown. From this point, even if only 12 new bits are introduced in each round, and even if the clocking is miraculously identical, a mere 64 rounds (on average) is all it will take to render every bit unknown. Given this level of diffusion, 128 rounds should be more than sufficient. For other reasons why 128 rounds was chosen, see below in the Cryptanalysis section.

Performance:
Due to limited time, I was unable to optimize the algorithm much at all, giving these results:

Performance:	Processor	Operating System	Compiler	Cycles/Byte
Optimized-32bit	Athlon 2x 2.0GHz	Ubuntu 8.04	Gcc 4.2.3	7,000-7,500
Optimized-64bit	Athlon 1.5GHz	Ubuntu 8.04	Gcc 4.2.3	3,000-3,500
Optimized-8bit	Estimate	Estimate	Estimate	~24,000
Setup for 32bit	Athlon 2x 2.0GHz	Ubuntu 8.04	Gcc 4.2.3	700,000 cycles
Setup for 64bit	Athlon 1.5GHz	Ubuntu 8.04	Gcc 4.2.3	600,000 cycles

On an 8-bit machine the algorithm will run in approximately 8 times as many cycles as the 64-bit versions. Tests have shown that the majority of the runtime is taken in the round function, which is divided into two time-consuming functions: stepping the CSRs, and performing the NLF. CSR stepping will take almost exactly 8 times as many cycles in an 8-bit implementation, because it will take 16 memory accesses to the CSR, compared with 2 in a 64-bit version. As for the NLF, the time taken will be almost the same, because all operations (table lookups) are 8-bit.

Cryptanalysis:
Ponic is vulnerable to an attack creating hash collisions if fewer than a critical number of rounds are used. If 63 or fewer rounds are used, this attack becomes guaranteed. If 64-100 rounds are used, this attack becomes merely likely (faster than brute force, at any rate). If 100-120 rounds are used, this attack becomes exceedingly unlikely (2^{-240}). If 121 or more rounds are used, this attack becomes impossible.
To explain this attack, I will simplify the algorithm to have 8-bit CSRs, and have 8-bit blocks.
Also, let csr(n, x)$_m$ be the state of csr(n) after x blocks of message m are hashed.

First our attacker hashes a sample message, *m1*: 0b00000000 0b00000000, and finds it has a hashsum of: 0xDEADBEEF. Next our attacker notes that the last bit of the first CSR never reaches the NLF after the first block is hashed, but merely ends up in bit 3 of the first CSR. So our attacker constructs the following message, *m2*: 0b00000001 0b001000000.
Therefore, csr(0, 1)$_{m1}$ ^ csr(0, 1)$_{m2}$ = 0b00100000 (^ represents the bitwise XOR operation.)
However, this time our attacker has put 0b001000000 in the second block, and when this is XORed into csr(0), it cancels out the changed bit. Our attacker has successfully found a collision between messages: 0b00000000 0b00000000, and 0b00000001 0b001000000. The only way to preclude this attack is to force each bit to enter the NLF. 121 rounds suffices to do just that.

At the cost of performance, the CSRs could be made to have relatively prime lengths. This would mitigate this attack if the key CSRs are shortened, and might also increase the period of the internal state.

Constants:

The sboxes of Ponic are defined as such:

sbox0 = the Rijndael sbox

sbox1 = [218, 87, 92, 101, 242, 44, 221, 84, 28, 155, 55, 43, 158, 246, 223, 16, 120, 116, 202, 54, 214, 211, 219, 194, 197, 60, 172, 65, 199, 25, 152, 61, 121, 104, 82, 191, 135, 36, 249, 68, 239, 13, 63, 110, 165, 136, 71, 67, 125, 127, 114, 73, 7, 11, 210, 235, 212, 4, 163, 175, 201, 47, 134, 62, 243, 107, 122, 70, 146, 167, 200, 248, 29, 253, 21, 156, 139, 66, 228, 226, 96, 220, 123, 149, 23, 192, 12, 118, 166, 184, 217, 90, 17, 69, 49, 222, 237, 229, 181, 115, 141, 190, 37, 208, 24, 124, 159, 111, 14, 129, 244, 77, 193, 161, 0, 30, 207, 9, 245, 251, 102, 2, 41, 130, 34, 106, 187, 182, 75, 140, 142, 203, 51, 64, 1, 40, 81, 56, 225, 35, 177, 15, 31, 153, 100, 83, 97, 174, 113, 59, 131, 179, 18, 224, 8, 78, 32, 95, 19, 170, 38, 132, 79, 195, 109, 189, 46, 173, 183, 45, 52, 215, 196, 33, 103, 22, 216, 233, 144, 119, 91, 80, 168, 143, 232, 137, 94, 231, 205, 198, 85, 93, 180, 162, 178, 76, 186, 185, 108, 26, 128, 42, 209, 89, 169, 112, 188, 151, 27, 105, 39, 157, 150, 236, 227, 58, 48, 230, 53, 57, 206, 255, 88, 250, 117, 241, 126, 240, 3, 10, 154, 72, 20, 238, 254, 160, 74, 176, 148, 6, 147, 5, 145, 50, 133, 247, 86, 98, 171, 234, 99, 204, 213, 252, 138, 164]

sbox2 = [162, 198, 227, 166, 25, 154, 90, 6, 115, 218, 210, 120, 187, 52, 188 215, 224, 205, 91, 185, 62, 43, 110, 153, 117, 17, 253, 67, 3, 42, 216, 251, 112, 163, 0, 4, 113, 242, 36, 28, 46, 223, 30, 89, 219, 39, 55, 142, 7, 60, 87, 234, 102, 190, 130, 139, 209, 69, 175, 93, 141, 104, 174, 232, 50, 212, 13, 131, 38, 180, 168, 70, 204, 40, 246, 235, 201, 11, 33, 94, 44, 214, 233, 231, 118, 68, 161, 244, 125, 122, 140, 155, 238, 250, 41, 27, 192, 8, 137, 86, 220, 189, 178, 165, 182, 177, 167, 213, 121, 51, 138, 134, 128, 169, 96, 202, 225, 239, 18, 101, 56, 157, 82, 176, 237, 181, 245, 199, 100, 59, 184, 158, 228, 24, 99, 240, 66, 73, 200, 119, 152, 123, 20, 151, 106, 164, 114, 9, 207, 255, 74, 173, 57, 16, 109, 243, 111, 186, 105, 129, 47, 241, 92, 45, 26, 127, 171, 48, 29, 1, 147, 31, 80, 84, 208, 54, 252, 193, 49, 156, 98, 85, 103, 195, 79, 21, 145, 236, 14, 230, 203, 23, 150, 247, 194, 211, 159, 191, 22, 64, 126, 143, 249, 179, 160, 206, 217, 124, 229, 77, 88, 132, 37, 183, 12, 172, 83, 146, 97, 136, 35, 95, 144, 10, 135, 5, 19, 32, 63, 71, 78, 61, 148, 196, 15, 133, 72, 76, 107, 2, 58, 170, 149, 75, 116, 81, 65, 254, 248, 53, 108, 222, 197, 34, 221, 226]

sbox3 = [71, 47, 90, 46, 1, 123, 33, 4, 117, 219, 157, 252, 173, 95, 182, 176, 89, 223, 208, 249, 37, 45, 247, 100, 128, 48, 131, 196, 232, 233, 20, 65, 14, 16, 224, 91, 59, 11, 82, 202, 30, 127, 2, 141, 225, 72, 158, 175, 84, 77, 43, 103, 166, 114, 228, 149, 169, 73, 246, 167, 36, 164, 129, 220, 122, 107, 142, 201, 85, 17, 148, 53, 9, 162, 156, 40, 111, 178, 116, 210, 199, 124, 108, 163, 251, 8, 61, 184, 52, 119, 106, 203, 83, 76, 143, 62, 186, 44, 66, 190, 54, 165, 222, 150, 242, 216, 238, 41, 248, 137, 69, 187, 153, 172, 19, 155, 104, 160, 49, 51, 255, 93, 170, 125, 185, 214, 34, 161, 78, 177, 146, 236, 204, 189, 179, 38, 254, 97, 133, 87, 174, 212, 113, 139, 94, 120, 211, 171, 88, 205, 206, 13, 135, 24, 226, 243, 102, 192, 86, 145, 29, 7, 134, 96, 50, 98, 63, 12, 101, 81, 35, 250, 234, 231, 144, 56, 207, 193, 64, 21, 92, 240, 57, 168, 18, 27, 74, 213, 217, 147, 237, 28, 183, 152, 132, 25, 15, 221, 31, 60, 109, 67, 200, 3, 229, 99, 112, 244, 22, 215, 253, 239, 55, 75, 191, 121, 126, 227, 197, 198, 194, 26, 195, 136, 70, 241, 42, 159, 235, 110, 154, 23, 209, 6, 80, 115, 105, 218, 118, 39, 10, 188, 32, 245, 0, 5, 138, 181, 79, 68, 58, 230, 140, 180, 130, 151]

sbox4 = [59, 232, 192, 227, 22, 7, 188, 72, 44, 122, 222, 32, 210, 16, 199, 221, 166, 37, 213, 64, 83, 27, 208, 123, 106, 254, 84, 118, 180, 67, 110, 82, 165, 172, 28, 103, 157, 241, 62, 212, 24, 1, 235, 203, 201, 231, 142, 9, 237, 255, 109, 197, 238, 159, 70, 152, 190, 135, 144, 94, 252, 214, 243, 21, 15, 168, 26, 95, 215, 248, 11, 150, 251, 207, 85, 236, 121, 36, 140, 223, 99, 209, 31, 158, 20, 205, 89, 51, 130, 71, 53, 154, 225, 184, 151, 6, 73, 155, 240, 77, 160, 198, 52, 242, 90, 133, 169, 0, 136, 148, 129, 185, 156, 195, 93, 107, 74, 120, 167, 4, 76, 246, 200, 253, 117, 38, 153, 63, 78, 113, 174, 128, 149, 138, 3, 87, 55, 96, 132, 211, 186, 23, 196, 233, 217, 226, 250, 49, 13, 54, 41, 179, 100, 68, 175, 194, 141, 164, 60, 247, 134, 229, 147, 124, 102, 101, 12, 119, 61, 69, 143, 187, 75, 80, 177, 81, 33, 46, 98, 14, 25, 234, 40, 218, 108, 56, 48, 224, 88, 202, 228, 45, 127, 79, 191, 2, 183, 91, 10, 189, 193, 244, 239, 17, 181, 146, 65, 173, 34, 115, 18, 145, 161, 57, 170, 66, 19, 104, 35, 131, 5, 171, 216, 8, 139, 220, 137, 126, 29, 116, 92, 176, 47, 219, 230, 105, 178, 111, 249, 162, 112, 97, 206, 42, 204, 182, 86, 39, 245, 58, 163, 30, 50, 43, 114, 125]

sbox5 = [200, 186, 59, 152, 249, 39, 65, 2, 124, 239, 174, 188, 66, 214, 238, 128, 145, 191, 44, 112, 42, 172, 4, 240, 163, 27, 45, 199, 6, 147, 108, 102, 134, 197, 132, 109, 92, 14, 138, 25, 26, 49, 114, 18, 60, 175, 149, 135, 130, 235, 43, 106, 111, 164, 50, 194, 201, 142, 126, 185, 9, 212, 224, 253, 206, 84, 3, 189, 165, 118, 180, 144, 158, 69, 29, 242, 161, 34, 79, 121, 51, 234, 82, 71, 183, 198, 56, 125, 208, 207, 218, 176, 13, 205, 184, 169, 76, 195, 219, 22, 32, 156, 223, 113, 15, 110, 94, 209, 38, 215, 137, 204, 90, 136, 33, 182, 107, 20, 168, 225, 127, 53, 23, 155, 226, 80, 139, 154, 177, 178, 81, 246, 232, 122, 78, 222, 119, 116, 146, 217, 97, 30, 87, 221, 150, 148, 67, 91, 173, 196, 115, 8, 129, 74, 228, 16, 68, 213, 237, 48, 52, 123, 159, 245, 62, 210, 254, 101, 233, 47, 248, 120, 24, 12, 241, 229, 100, 181, 5, 70, 95, 61, 83, 166, 0, 117, 21, 89, 10, 236, 98, 63, 64, 99, 141, 131, 37, 143, 157, 19, 230, 104, 88, 179, 251, 151, 216, 41, 35, 250, 75, 96, 133, 170, 36, 57, 193, 77, 46, 231, 85, 93, 54, 40, 160, 162, 187, 255, 73, 227, 247, 11, 171, 103, 202, 220, 55, 211, 86, 243, 31, 17, 1, 190, 203, 244, 72, 140, 105, 252, 28, 192, 167, 7, 153, 58]

sbox6 = [250, 184, 41, 219, 253, 18, 40, 95, 122, 198, 9, 159, 211, 178, 15, 176, 45, 163, 128, 65, 124, 61, 67, 3, 181, 22, 112, 133, 167, 34, 4, 246, 196, 146, 186, 129, 210, 142, 179, 109, 32, 59, 156, 141, 30, 99, 1, 222, 24, 49, 238, 119, 215, 207, 244, 80, 137, 21, 66, 12, 96, 201, 157, 169, 54, 193, 29, 33, 132, 107, 46, 177, 251, 105, 229, 138, 255, 194, 243, 89, 241, 106, 94, 117, 101, 120, 144, 76, 216, 173, 87, 25, 152, 52, 240, 83, 153, 78, 8, 155, 224, 247, 85, 108, 214, 225, 127, 90, 69, 164, 86, 154, 131, 175, 10, 235, 20, 126, 140, 28, 143, 190, 57, 195, 64, 110, 116, 53, 223, 168, 220, 37, 231, 51, 189, 158, 245, 50, 93, 134, 147, 55, 74, 82, 212, 79, 0, 183, 118, 84, 237, 19, 208, 209, 218, 172, 6, 73, 113, 200, 43, 188, 249, 62, 161, 36, 58, 239, 100, 103, 130, 145, 150, 248, 136, 135, 97, 205, 13, 192, 233, 98, 27, 226, 148, 16, 242, 185, 92, 236, 149, 60, 23, 88, 39, 217, 5, 228, 11, 230, 71, 68, 213, 202, 139, 234, 199, 221, 204, 72, 63, 171, 7, 56, 165, 197, 114, 254, 35, 182, 42, 81, 174, 191, 75, 26, 206, 31, 187, 151, 203, 162, 125, 115, 91, 252, 166, 2, 48, 121, 123, 227, 17, 160, 170, 180, 14, 102, 232, 70, 38, 111, 47, 77, 44, 104]