IZNOGOOD : How to exploit a weak linear layer?

Augustin Bariant

May 9, 2022

1 My personal life and CTF experience

I am Augustin Bariant, 24 years old, and this is my first solo CTF. I once participated in a 10 hours CTF in groups of 3 (Aviation ISAC Collegiate CTF Competition), though back then I was a beginner in most CTF fields, excluding cryptography. We still managed to rank 3rd.

On the academic point of view, I was really interested in mathematics very early on, and I started to take part of the French Mathematical Olympiads in the beginning of high school. Following my interests, I went to the Louis Le Grand preparatory class in Mathematics and Physics and finally got accepted to Ecole Polytechnique. There, I heavily gained interest in computer science and decided to specialize in cryptography, a meeting point of my two loving fields. I performed a double degree with KTH Royal Institute of Technology in Stockholm during my last year of Ecole Polytechnique. I am currently a Ph.D student at INRIA de Paris in symmetric cryptography. The core subject of my Ph.D is the cryptanalysis of symmetric schemes, including AES. I was therefore in good conditions to solve this challenge, that requires non-trivial symmetric cryptanalysis.

For more information, check out my personal page here.

2 IZNOGOOD: The challenge

Your intern does not trust AES and wrote his own blockcipher algorithm. Can you remind him that constructing its own crypto should be forbidden?

For this challenge, we have access to:

- The specification of the blockcipher.
- One plaintext-ciphertext pair under a random key K.
- Three other ciphertexts under a key K. Their decryption yields the end of the flag.

Objective Recover the key K and decrypt the ciphertexts corresponding to the flag.

3 IZNOGOOD Specification

Let us look at the specification of IZNOGOOD, in the file IZNOGOOD.py. IZNOGOOD seems to be a standard Substitution-Permutation-Network (SPN) block cipher. SPN ciphers are very popular, as shown for instance by the standardization by the National Institute of Standards and Technology of Rjindael and Keccak into respectively the names of AES and SHA3. They are solid for various reasons:

- The substitution layer provides non-linearity (or confusion): output bits gain non-linear dependance on the input bits.
- The linear layer provides diffusion: output bits depend on many input bits.
- The SPN schemes are often relatively easy to cryptanalyse. For instance, the simplicity of the linear layer of AES makes the diffusion of AES very understandeable.

IZNOGOOD operates on a state of 32 nibbles of 4 bits. The cipher is composed of 8 rounds of the following operations:

- **AddKey:** The secret key is XORed to the state.
- **AddRConstant:** A 128-bit round constant is XORed to the state.
- **SubBytes:** A 4-bit Sbox is applied to each nibble of the state.
- **LLayer:** The state is transformed linearly.

The linear layer of the last round is omitted.

4 Cryptanalysis of IZNOGOOD

4.1 Overview of IZNOGOOD

I performed a little overview of the operations of IZNOGOOD:

- **AddKey:** The secret key seems to be correctly generated.
- **AddRConstant:** The round constants look randomly generated.
- **SubBytes:** The Sbox seems strong. For instance, I computed the Difference Distribution Table (DDT) of the Sbox, and the table only contained values of 2 and 4. This indicates that the Sbox has strong confusion properties. **Post-CTF remark**: the Sbox is actually PRESENT's Sbox.

- **LLayer:** The linear layer is odd and seems to have weak properties. A first observation is that for all $0 \geq i, j < 32$, $S[i] \oplus S[j] = \text{LLayer}(S)[i] \oplus \text{LLayer}(S)[j]$, which is an interesting property that might hide some even weaker ones. Therefore, we should study this linear layer more in depth.

First, we can represent the LinearLayer as a 32-32 Matrix L operating on the full state:

$$L = \begin{bmatrix} 0 & 1 & \dots & 1 \\ 1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \dots & 1 & 0 \end{bmatrix}$$

L can be expressed as such:

$$L = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \dots & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{bmatrix} \equiv J + I$$

Where the + denotes a bitwise XOR.
If we denote the state before the linear layer as a 32-nibble vector S:

$$L \circ S = J \circ S + S$$

We remark that all the lines of J are equal, so $J \circ S$ is a vector of 32 equal values (equal to the XOR sum of the nibbles of S).

Attack idea: For each round, guess the XOR sum of nibbles: $2^4 = 16$ possible values per round, for 7 rounds (since the last linear layer is omitted). We need to guess a total of 2^{28} bits, that we denote g. Let us also denote the known plaintext-ciphertext pair (P, C). The mapping from $P[i]$ to $C[i]$ only depends on $K[i]$, g and i (since the round constants depend on the position). i.e

$$C[i] = f(P[i], K[i], g, i)$$

where f is a known function. Formulating this, we reduced the dependancy of $C[i]$ on all $P[j]$ and $K[j]$ with $i \neq j$ to only a 28-bit unkown value g. After guessing g, we detect a right key nibble candidate $K[i]$ if $C[i] = f(P[i], K[i], g, i)$ holds. If for a guess g, there exists at least one key nibble candidate $K[i]$ for all $i = 0 \dots 31$, we retain the corresponding key candidates (potentially multiple candidates if there are multiple nibble key candidates for certain indexes). Each key candidate induced by the array of lists of nibble candidates can be confirmed or rejected directly depending on if $C = E(P, K)$ holds. This idea can be exploited in the following attack.

4.2 Attack on IZNOGOOD

I implemented the attack in python, which took around ten hours to return the key. Why recode everything in C/C++ if it works in python with the orginal implementation?

```python
def brute():
    for i in range(1,1<<28):
        if i%(1<<15) == 0:
            print(i)
        #prk stores the guessed value
        prk = [(i>>(4*j))&0xf for j in range(7)]
        K = [[] for j in range(32)]
        candidate = True
        # For all nibble position kp
        for kp in range(32):
            # For all possible values of K[kp]
            for j in range(1<<4):
                # Compute f(input[kp],K[kp],g,kp)
                res = singleNibbleIZ(plaintext[kp],j,prk,kp)
                if res == ciphertext[kp]:
                    K[kp].append(j)
            # There exist a position with no candidate
            # break and try the next guess;
            if K[kp]==[]:
                candidate=False
                break
        #If there is a candidate for each position
        if candidate:
            print("Testing some keys")
            # The following function encapsulates a loop over all keys
            # induced by K and a quick verification test.
            k,b = testValidity(prk,K)
            if b:
                with open("sol.txt",'w') as f:
                    f.write(str(k))

            print("Go back to bruteforce")
```

Complexity Analysis: First, we need to loop over $2^{28} \times 32 \times 16 = 2^{39}$ values, and for each of them perform 8 Sbox calls. This can be reduced to an average of $2^{28} \times 2 \times 16 = 2^{35}$ values by exiting early the loop on kp, considering that $K[kp] == 0$ happens with probability $1/2$ (which is not totally accurate, the real probability is equal to $(1 - 1/16)^{16} \approx 1/e$). Secondly, from time to time some key candidates need to be tested. Most of the time, the number of induced keys from the array of nibble key candidates is not too large (around 2^{15}), as verified experimentally. For each of them, a full state encryption needs to be performed ($32 \times 8 = 256$ Sbox calls). However, for the right key candidate, I found around 2^{29} potential key candidates in the list of nibble candidates. This can be explained because for each position kp, $K[kp]$ contains the right key candidates plus random candidates (in average 1 other). So with a very rough

approximation, we expect 2^{32} ($= \Pi_{kp=0}^{31}|K[kp]|$) candidates. Due to the inequal repartition of the nibble candidates in the list K, $\Pi_{kp=0}^{31}|K[kp]|$ is in practical a bit smaller. 2^{29} encryptions is very costly, so I had to optimize the search for the potential keys.

Final search optimization: I used a meet-in-the-middle trick: first, I split the list K into two lists LK and RK with K = $LK||RK$ of size 16 each. For I loop through all possible values of g. Then, for each 16-nibble key candidate induced by LK, I computed the partial nibble sums (16-nibble sum) at each round. It can be done without knowing RK since the internal nibbles of index i only depend on $P[i]$ and $K[i]$. I concatenated the 7 obtained partial nibble sums (yielding a 28-bit guess), XORed g to it, and stored that into a hashmap. For each 16-nibble key candidate induced by RK, I computed the other partial nibble sum and looked for collisions in the hashmap. A collision means that the full nibble sum at each internal state equal the guessed value, therefore that the guessed value is correct, so the two colliding key candidates form a correct full key. This allows to only compute approximately $2^{14.5}$ instead 2^{29} encryptions.

Eventually, this attack leads to several keys that map the plaintext P to C. I decrypted the three other ciphertexts and looked for something that ressembled a flag. Gotcha!!