21:12 NSA's Backdoor of the PX1000-Cr

by Stefan Marsiske

I was supposed to be doing paid work, porting some silly crypto protocol to browsers. This implied getting dirty with JavaScript, the insanity of fast changing and incompatible browser interfaces, and other nasty beasts. Instead, I remembered an exciting device from a Crypto Museum exhibition. Behold the incredible PX 1000 Cr!

This diabolical pocket telex (an antique peer-to-peer messaging thingy) from 1983 had a unique feature: it came with DES encryption and was marketed toward small companies and journalists. According to some rumors, even the Dutch government used some.

This freaked out the NSA, who sent an emissary to buy up all the stock from the market and pressured Philips to suspend sales of any such infernal devices. In '84, the NSA provided Philips with an alternative encryption algorithm, which they were happy to sell to the public. The astute reader, being knowledgeable about the NSA's backdooring efforts, should immediately suspect that the new firmware might be weird in some ways. I certainly suspected a little mischief.

[32] `unzip pocorgtfo21.pdf brucker-thesis.pdf`

Exposition

Luckily, the fine people of the Crypto Museum have not only dedicated a couple of pages to this device, but they also published ROM dumps of both the original DES-enabled device as well as the agency-tainted device. They also published Ben Brücker's bachelor thesis,[32] in which he reverse engineered much of the DES variant of the device.

Although his thesis did not contain much source code, it was enough of a head start that I could dive directly into the encryption code.

My first steps were a mistake. I could not resist the irony of using a tool from Fort Meade to break their own backdoor, so I loaded the ROM into Ghidra and started annotating memory addresses. Unfortunately, Ghidra does not support this particular CPU. Luckily, IDA Pro has excellent support for this chip.

The CPU in question is a Hitachi HD6303. It's a derivative of the Motorola 6800, which was a simple but, for its time, powerful 8-bit processor. It only has four 16 bit registers: a stack pointer, an instruction pointer, an index register to address memory,

Text Lite PX1000, Photo from the Crypto Museum

and an accumulator. The latter can be accessed as a 16 bit register or as two 8-bit registers. The instruction set is simple, but certainly capable of doing great things. Turns out the same CPU was also used in the venerable Psion II Personal Digital Assistant. This means there are fans of this device who document, for example, the instruction set.[33]

The fine people of the Crypto Museum also published an undated photocopied and scanned version of the CPU datasheet.[34] It took me a few days to progress from realizing that some pages are missing, to realizing that only the even-numbered pages are missing, to realizing that the even-numbered pages start half-way into the document in decreasing order. All hints that the pages have been scanned while holding discordian principles high. Hail Eris, indeed!

Having all this supporting information available made it an easy effort to work my way from the binary dump to an equivalent algorithm written in C. However, there were some weird things. For example, the encryption function starts by decrementing the pointer to the plaintext by one. Why? And where does that preceding byte come from, and will people be offended if I index a C `char` array with -1? Answering these questions meant I had to either reverse engineer other parts of the ROM that were unrelated to the cryptographic algorithm—which I am too lazy to do—or I had to find another way. Turns out the Psion II fans also created an emulator: SIM68xx[35] by Felix Erckenbrecht and Arne Riiber.

One of the most active contributors to this fine piece of software is Mayer Gabor, a Hungarian name. As I've lived and founded a hackerspace in that fine country, it wasn't hard to confirm that this contributor is a regular in such circles. After a friendly chat on IRC, he also became interested in the ROM dumps, but—just like Ben Brücker—he focused on the DES version. I complained about the plaintext array that is indexed by -1, and that it would be easy to figure out with a proper emulator.

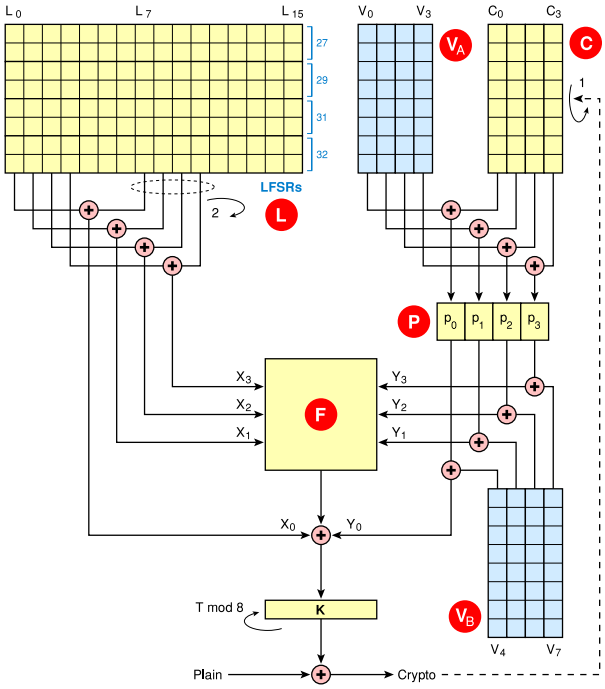
One day later, Gabor shared a branch of SIM68xx that adds support for running the PX1000 firmware, given a few minor patches for compatibility.[36]

There it was, a working emulator with a display and keyboard. It turned out it is possible to set the text-width. The -1 character is indeed encoding the text-width, which is limited by the size of the display to 40. It also turned out that the plaintext is also post-fixed with another character, `0x8d`, before encryption. Here, the most significant bit, which is never set in ASCII, marks the end of the string. Thus `0x8d` encodes both a newline character and the EOS.

With the working emulator I was able to verify my C interpretation of the encryption algorithm. It was finally time to start breaking the crypto!

Dramatis Personae

The algorithm itself can be shown in a simplified block diagram, helpfully provided by the Crypto Museum.



[33] `https://www.jaapsch.net/psion/mcmnemal.htm`

[34] `unzip pocorgtfo21.pdf hd6303rp.pdf`

[35] `git clone https://github.com/dg1yfe/sim68xx || unzip pocorgtfo21.pdf sim68xx.zip`

[36] `unzip pocorgtfo21.pdf sim68xx-px1000.zip`
`git clone https://github.com/iddq/sim68xx.git; cd sim68xx; git checkout origin/px1000`

The Mysterious Key

Remember, this device is a 7-bit ASCII input device. How can someone enter an encryption key without much hassle? The engineers came up with a nice idea: Take an arbitrary 16 byte string, zero out the top nibble of each byte, and only use the lower (and slightly higher entropy) nibble, providing with a 64-bit key, which is stronger than the measly 56 bit key of DES.

Let's introduce our other main characters. In the schema, on the top left, the 16 byte block denoted L is supposed to be a set of four linear feedback shift registers. This is the bad guy, the end level boss. He is elusive and changes like a chameleon.

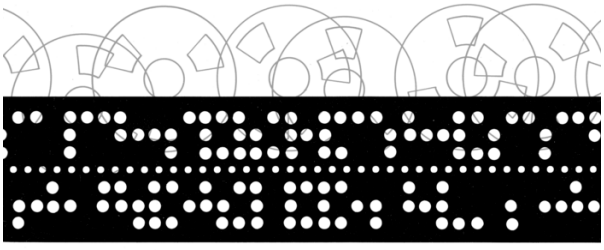
To the right we have two blue blocks, V_A and V_B, of four bytes each which contain some transformation of the encryption key. This is a supporting character, mostly stays in the background, and has little character development.

Right of V_A we have the four byte C block, which is a FIFO initially containing a transformation of parts of the encryption key, but it later becomes a cipher-feedback buffer containing the last four bytes of ciphertext. Another supporting character, this guy looks strange in the beginning, but later on becomes a familiar face we know and recognize.

The block denoted by P is really just a transformation which replaces each 4-bit nybble with another 4-bit nybble based on a lookup-table. This young lady is the sister of F, but is mostly staying predictable.

The big yellow block F in the middle is eight non-linear transforms that converts 6 input bits into one output bit, more on this later. This lady is another trouble-maker; she's the femme fatale of this play, working with the evil guy, making things difficult.

And last the small block K is a transformation of the keystream byte, that rotates the keystream byte left by the number of byte being currently encrypted modulo 8. Just another supporting character without much depth.



Act I

It is very important to see how these blocks are initialized—this is the part where the alarm bells start getting louder. During initialization, one operation comes up everywhere: the low nibble gets complemented and set as the high nibble.

```
1 //Invert low nybble into the high nybble.
uint8_t invert2hi(uint8_t x) {
3   return ((~x) << 4) | x;
}
```

In fact this is how 15 of the 16 bytes of the LFSR are initialized: Each low nibble of the key is taken and inflated into a byte. The last byte is set to `0xff`. As code:

```
for(i=0; i<15; i++) {
2   lfsr[i] = invert2hi(key[i]);
}
4 lfsr[15]=0xff;
```

Now, if you happen to somehow know the internal state of the LFSR and also know how to reverse it, then it becomes trivial to check if any state has the special structure of the initial state, from which the key can be trivially recovered. I'm not sure if that actually helps, but it's ugly anyway.

Blocks V_A, V_B, and C are similarly initialized:

```
for(i=0; i<4; i++) {
2   V[i]   = invert2hi(key[i]   ^ key[i+4]);
    V[i+4] = invert2hi(key[i+8] ^ key[i+12]);
4   C[i]   = V[i] ^ V[i+4] ^ 0xf0;
}
```

It's not obvious at first, but if you expand `V[i]` and `V[i+4]` when setting `C[i]` and do the math, you will come to the conclusion that the values of C can only be one of these sixteen legal values: `0x0f`, `0x1e`, `0x2d`, `0x3c`, `0x4b`, `0x5a`, `0x69`, `0x78`, `0x87`, `0x96`, `0xa5`, `0xb4`, `0xc3`, `0xd2`, `0xe1`, or `0xf0`.

My alarm bells are kinda deafening by now, how are yours?

After the initialization, the stream cipher is ready to be used. For each key-stream byte the LFSR is mutated, then combined with the V and C blocks, fed into the F function, and then XOR'd into the plaintext. Let's have a look first at the mutation of the LFSR block:

```
for(round=0x1f; round>=0; round--) {
  acc = 0;
  // FAC7 in the code this loop is unrolled
  for(i=0; i<16; i++) {
    acc ^= lfsr[i]lookupTab[(round+i)%16];
  } // FB43

  // FB45..FB4A
  acc = ((acc >> 1) ^ acc) & 0x55;

  // tmp is twice the sequence 15..0
  tmp=(round ^ 0xff) & 0xf;
  lfsr[tmp] = ((lfsr[tmp]<<1)&0xAA)|acc;
} // FB63
```

Doesn't really look like a traditional LFSR to me, or even a set of them. But if the Crypto Museum people say so, I'm going with their insights. *Nota bene*: Those 16-bit hex numbers in the comments mark the addresses for where in the ROM this code can be found.

Normally an LFSR emits a bit after each advancement. In this code it is not obvious how this is done. The following snippet shows how four bytes are extracted from the LFSR after it has been mutated:

```
for(i=0; i<4; i++) {
  tmp = lfsr[i+7]; // FB68..FB6C

  // 2x rotate left FB6E..FB72
  tmp = (tmp << 2) | (tmp >> 6);

  // FB74 .. FB7A
  lfsr_out[i] = tmp ^ lfsr[i];
}
```

If you squint you might imagine that there are four LFSRs but, as you will see, this doesn't matter much for our final attack. This concludes the left side of the schema before being fed into the non-linear function F.

On the right side of the schema you can see how V_A and the ciphertext FIFO are being XOR'd and mapped through P. It looks like this in code.

```
for(i=0; i<4; i++) {
  tmp = V[i] ^ CiphertextFifo[i];
  acc = map4to4bit[i][tmp >> 4] << 4;
  acc |= map4to4bit[i][tmp & 0xf];
  pbuf[i] = acc ^ V[i+4];
}
```

Looks straightforward, but if we unpack this in the context of encrypting the very first character (which is probably "(", but this is irrelevant here), then we can unpack:

```
tmp = V[0] ^ CiphertextFifo[0]
```

where

```
CiphertextFifo[0] = V[0] ^ V[4] ^ 0xf0
```

which drops out `V[0]`, and thus:

```
tmp == V[4] ^ 0xf0
```

and we know that all values of `V` are values where the high nibble is just the inversion of the low nibble, and if we XOR that with `0xf0`, we conclude that `tmp` can only be one of these 16 values: `0x00`, `0x11`, `0x22`, `0x33`, `0x44`, `0x55`, `0x66`, `0x77`, `0x88`, `0x99`, `0xaa`, `0xbb`, `0xcc`, `0xdd`, `0xee`, or `0xff`.

Strange, huh? This loop runs four times, with similar results for each output byte. Later, when the Ciphertext FIFO is filled with real ciphertext, this doesn't apply anymore, but then the contents of this buffer are known, since it's the ciphertext. The mapping itself of four bits to four bits was relatively uninteresting ... or at least I couldn't immediately see anything wrong with it. And also XORing that with V_B was also much less exciting.

Now that we have the inputs to the F function, we can analyze what happens there. The code is a bit dense, we'll unpack it later:

```
for(i=8,acc=0; i>0; i--) {
  // FBB9
  for(j=1,tmp=0; j<4; j++) {
    tmp = (tmp << 1) | (lfsr_out[j] >> 7);
    lfsr_out[j]<<=1;
    tmp = (tmp << 1) | (pbuf[j] >> 7);
    pbuf[j]<<=1;
  }
  tmp=lookupTab6To1bit[tmp];

  acc=(acc<<1) + ((tmp>>(i-1)) & 1);
} // 0xfbd9
```

The outer loop takes care that all eight bits of each input of the six input bytes get used in F and that the output of F is being assembled back into one byte. The inner loop interleaves the 6 input bits from `lfsr[1]`, `pbuf[1]`, `lfsr[2]`, `pbuf[2]`, `lfsr[3]` and finally `pbuf[3]`. The lookup table produces one bit, which in the last line is put into the correct bit-position of the accumulator. It's a pretty straightforward bit-sliced 6-byte-to-1-byte mapping. The lookup table is neat, it's 64 bytes, which is indexed by the six-bit interleaved value, and from the resulting byte the i^{th} bit is extracted. Very compact, neat.

The next steps are unspectacular, keeping in mind that `curChar` starts with −1:

```
acc ^= pbuf[0] ^ lfsr_out[0];

// FBDF
tmp = (curChar + 1) & 7;
// rotate left by tmp
acc = (acc << tmp) | (acc >> (8-tmp));

ciphertext[curChar]=plaintext[curChar]^acc;
```

For decryption, note that only this last line needs to swapped be around.

One last step is needed before we can loop back to mutating the LFSR, and that is advancing the ciphertext FIFO, now that there is a ciphertext byte. Again, this is pretty straightforward, and after four ciphertext bytes, the peculiar structure noted above of the initial four bytes in this FIFO is lost:

```
// FC05
CiphertextFifo[4] = ciphertext[curChar];
for(i=0; i<4; i++) {
  // rotate left
  CiphertextFifo[i] =
    (CiphertextFifo[i+1] << 1) |
    (CiphertextFifo[i+1] >> 7);
} // FC15
```

A small optimization is that the array holding the FIFO is actually five bytes, and the newest ciphertext byte is always added to the fifth position, which enables this compact loop updating the four effective items in this FIFO.

If there are more plaintext bytes to encrypt, then the algorithm loops back to mutating the LFSR. Otherwise, everything is done

ACT II: Climax

This all looks a bit fishy, but how does one actually *break* this scheme? Well for a long time I focused on somehow figuring out the LFSR and how it can be decomposed in four LFSRs of 32, 31, 29, and 27 bit lengths as indicated on the Crypto Museum schema. Many hours were wasted into slicing and dicing the LFSR, mutating it, slicing and dicing it again, writing bit level differs, staring at colored bits, throwing Berlekamp-Massey at it, trying to write my own 32/31/29/27 bit LFSRs and seeing if I could somehow slice-'n-dice a state from the big one into the ones I implemented. It was a nightmare of dead ends, failure, and despair. Boredom started to set in, and I started to ask friends if maybe they could figure out how this works. They said it's easy, but they do not have time for this now. Anyway, maybe this is an LFSR or even four, but I was unable to figure out how.

I also started to consult the bible of cryptanalysis, Antoine Joux's masterpiece: *Algorithmic Cryptanalysis*. It has a chapter, *Attacks on Stream Ciphers*, about LFSRs hidden behind a non-linear function F. Antoine calls these *filtered generators*:

> The *filtered generator* tries to hide the linearity of a core LFSR by using a complicated non-linear output function on a few bits. At each step, the output function takes as input t bits from the inner state of the LFSR. These bits are usually neither consecutive, nor evenly spaced within the register.

Bingo! Exactly what I've been staring at for days now: the *big guy* and the *femme fatale*. The chapter mostly covers correlation attacks, but at the end there is also mention of algebraic attacks, the latter giving me a warm fuzzy feeling. Algebra is elementary school stuff, I can do that!

Antoine goes on:

> The function f is usually described either as a table of values or as a polynomial. Note that, using the techniques of Section 9.2, f can always be expressed as a multivariate polynomial over $\mathbb{F}_2$.

The technique in section 9.2 is is called the *Möbius transform* which is used to calculate the *Algebraic Normal Form* (ANF) of a boolean function. I tried to implement the *Möbius transform* as given

in algorithm 9.6 in Joux' masterpiece, but the results were not providing the expected outputs as the lookup table. After reading a bunch of papers on algebraic normal forms, I learned that different disciplines call this different names, such as

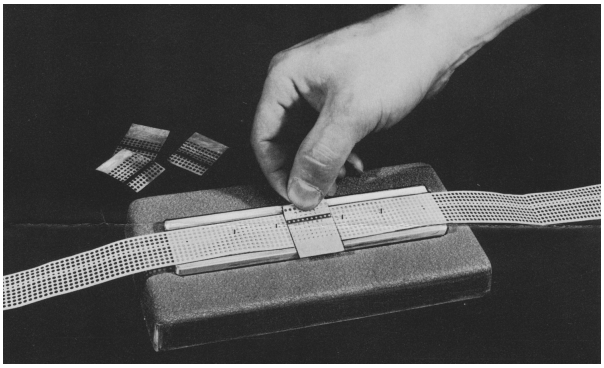
- ANF Transform (ANFT),
- Fast Möbius Transform,
- Zhegalkin Transform, and
- Positive Polarity Reed–Muller Transform.

Valentin Bakoev's excellent paper *Fast Bitwise Implementation of the Algebraic Normal Form Transform*[37] went into much more detail than Joux on this topic, and an implementation of Bakoev's Algorithm 1 gave the expected results.

```
void moebius(uint8_t *f, int n) {
  int blocksize=1;
  for(int step=1; step<=n; step++) {
    int source=0;
    while(source < (1<<n)) {
      int target = source + blocksize;
      for(int i=0; i<blocksize; i++) {
        f[target+i]^=f[source+i];
      }
      source+=2*blocksize;
    }
    blocksize*=2;
  }
}
```

We can split up the original F lookup-table bit-by-bit.

```
static uint8_t lookupTab6To1bit[64]={ // at 0xFE9B
    0x96, 0x4b, 0x65, 0x3a, 0xac, 0x6c, 0x53, 0x74,
    0x78, 0xa5, 0x47, 0xb2, 0x4d, 0xa6, 0x59, 0x5a,
    0x8d, 0x56, 0x2b, 0xc3, 0x71, 0xd2, 0x66, 0x3c,
    0x1d, 0xc9, 0x93, 0x2e, 0xa9, 0x72, 0x17, 0xb1,
    0xb4, 0xe4, 0xa3, 0x4e, 0x27, 0x5c, 0x8b, 0xc5,
    0xe8, 0x95, 0xe1, 0xd1, 0x87, 0xb8, 0x1e, 0xca,
    0x1b, 0x63, 0xd8, 0x2d, 0xd4, 0x9a, 0x99, 0x36,
    0x8e, 0xc6, 0x69, 0xe2, 0x39, 0x35, 0x6a, 0x9c
};
```



[37] `unzip pocorgtfo21.pdf bakoev-afn.pdf`

Feeding it into the `moebius` function, we get this.

```
f0= 01100010011010101011100011101011
    00101011011100011010010001011100
g0= 01100101000011111011011010010001
    01011011010010010011000110001110

f1= 11010010001101010111011000110110
    00111010000010111100010111010010
g1= 10111000100111101100100111001011
    10010110101111010100100111111110

f2= 10101101011011001100001110010010
    11011101010010100001100111000101
g2= 11000111101010110110111111101110
    01110111010000101100000010110110

f3= 01011100100010111010000111011000
    00010110100001111011011010101011
g3= 01001110101111001000001111000101
    01011001010100110100111100110000

f4= 10010011100100110100110110100111
    10000100010101101010111100001101
g4= 11101100000000001011001010010101
    00010110101110001000110011000010

f5= 00111101110101000010101100011101
    11101000101001000101000100111110
g5= 00101001100101110001011110110011
    10111111110010001100010010000010

f6= 01100111101010110101111001000100
    01010101101100010110100001110010
g6= 01100001101000000010110010111101
    00100001001111000000010100111100

f7= 10001000010101001001010001101001
    11100011111111010010111011010001
g7= 11110000101100010001101100110010
    01101011101010011011101010101010
```

The output of the Möbius transform is just another lookup table, a boolean function with exactly the same amount of input parameters as the the original non-linear function. Using this it is possible to create the ANF of the non-linear function:

$$f(x_0,\ldots,x_{n-1}) = \bigoplus_{(a_0,\ldots,a_{n-1})\in\mathbb{F}_2^n} g(a_0,\ldots,a_{n-1}) \prod_i x_i^{a_i}$$

In this equation the $g(\ldots)$ coefficient is the output of the Möbius transform, and since these bits are either 0 or 1, we can eliminate around half of all terms.

By inputting the `fx`/`gx` pairs we obtained from the `moebius` function into the following Python beauty, we can construct the ANF.

```
' ^ '.join(
  '('+c+')'
   for c in [
     '&'.join(
         f"x[{i}]"
         for i, x in enumerate(reversed(f'{a:06b}'))
           if x == "1")
         for a in range(64) if moebius[a]=='1']
    if c)
```

This can then be evaluated for all values between 0 and 63 and should produce the same result as the corresponding `fx`. If the result is the exact inverse of `fx`, then the ANF has an odd number of constant 1 terms, and the ANF must be fixed by prefixing it with `1 ^`.

For illustration, behold the ANF of `f4`:

```
1 + (x0) + (x1) + (x2) + (x0&x2) + (x4)
  + (x1&x4) + (x0&x1&x4) + (x1&x2&x4)
3 + (x3&x4) + (x0&x1&x3&x4) + (x0&x2&x3&x4)
  + (x0&x1&x2&x3&x4) + (x0&x1&x5)
5 + (x0&x2&x5) + (x1&x2&x5) + (x3&x5)
  + (x1&x3&x5) + (x0&x1&x3&x5) + (x2&x3&x5)
7 + (x4&x5) + (x2&x4&x5) + (x0&x2&x4&x5)
  + (x3&x4&x5) + (x0&x3&x4&x5)
9 + (x1&x3&x4&x5) + (x1&x2&x3&x4&x5)
```

Woohooo, look ma, I converted a lookup-table into algebra! I mean, I defeated the evil temptress, the femme fatale! After a few days of pondering, I also converted the lookup table marked 🅟 in the schema to its ANFs. Erm, I mean, I defeated the younger sister. The path to this victory was not immediately obvious, since 🅟 is a 4-bit to 4-bit table, and the Möbius transform only applies to boolean functions with one output bit.

The trick was to deconstruct the 4-to-4 mapping into four times 4-to-1 mappings, one for each output bit, while of course the input bits will be always the same for the same nibble. Hah! Take that, NSA! Most of your backdoor is now reduced to a bunch of polynomials!

ACT III: The Fall

But what do we do with that big guy, the end level boss, that pesky LFSR block? I kinda gave up on finding the polynomial for the LFSR, but maybe there is a different way to convert this into algebra? I've always been a big fan of Angr and symbolic execution. Maybe if I let Angr consume the loop that mutates the LFSR, I can get some symbolic constraints. Symbolic constraints being nothing other than equations. The trick was to modify the the loop to not run in place, but to output another 16 byte LFSR. Angr can then tell me, symbolically, how the output LFSR depends on the input LFSR. The (much truncated) output is promising.

```
<BV128 state_19_128[87:87]    ^ state_19_128[63:63] ^
       state_19_128[55:55]    ^ state_19_128[31:31] ^
       state_19_128[23:23]    ^ state_19_128[7:7]   ^
       state_19_128[102:102] ^ state_19_128[86:86] ^
       state_19_128[70:70]    ^ state_19_128[62:62] ^
       state_19_128[54:54]    ^ state_19_128[46:46] ^
       state_19_128[30:30]    ^ state_19_128[14:14] ..
```

Notice the trailing `..` in the last line. This signals concatenation of bit vectors. In total, 128 bits are being concatenated! The big guy finally reveals some weakness! Angr gave me the bits I needed to XOR together for each bit in the next state. After running some `sed` magic on this output, I had 128 lists, with only the bit positions contributing to the next state of this bit.[38] Wow, this really looks like algebra, but first lets analyze this list of lists a bit more.

I was very interested how these bits are related. I wrote a recursive function, taking one bit and visiting recursively all bits that this bit depends on. My goal was to figure out if there is loops or islands in this graph. This was my recursive function:

```
def walk(bit, c):
  c.append(bit)
  for b in bits[bit]:
    if b in c: continue
    c=walk(b,c)
  return c
```

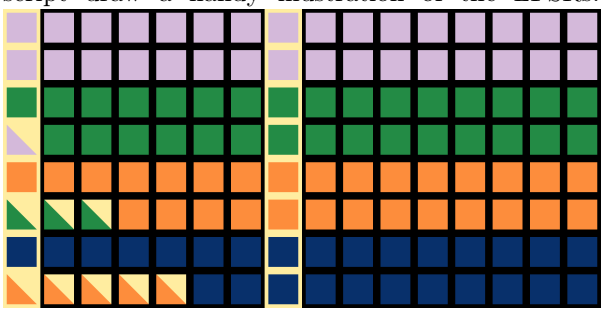
I ran it for all values from 0 to 127, discarding any duplicate results. Here are the bit indices for which I first saw a result, its length, and the result values themselves.

```
0 32   (0, 1, 8, 9, 16, 17, 24, 25, 32, 33, 40, 41, 48,
        49, 56, 57, 64, 65, 72, 73, 80, 81, 88, 89, 96,
        97, 104, 105, 112, 113, 120, 121)
2 31   (2, 3, 10, 11, 18, 19, 26, 27, 34, 35, 42, 43,
        50, 51, 58, 59, 66, 67, 74, 75, 82, 83, 90, 91,
        98, 99, 106, 107, 114, 115, 122)
4 29   (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
        52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
        100, 101, 108, 116, 124)
6 27   (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
        54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
        102, 110, 118, 126)
95 28  (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
        54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
        95, 102, 110, 118, 126)
103 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
        54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
        102, 103, 110, 118, 126)
109 30 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
        52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
        100, 101, 108, 109, 116, 124)
111 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
        54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
        102, 110, 111, 118, 126)
117 30 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
        52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
        100, 101, 108, 116, 117, 124)
119 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
        54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
        102, 110, 118, 119, 126)
123 32 (2, 3, 10, 11, 18, 19, 26, 27, 34, 35, 42, 43,
        50, 51, 58, 59, 66, 67, 74, 75, 82, 83, 90, 91,
        98, 99, 106, 107, 114, 115, 122, 123)
125 30 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
        52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
        100, 101, 108, 116, 124, 125)
127 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
        54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
        102, 110, 118, 126, 127)
```

Whoa! The first four results are with lengths 32, 31, 29, and 27. That seems to be the source of the Crypto Museum people claiming that there are four small LFSRs hidden in there. There are also nine positions that are not contributing to the first four loops, but which themselves do depend on bits in those.

[38] `unzip pocorgtfo21.pdf px1k.zip; unzip px1k.zip lfsr-next-bits.txt`

To make this all much clearer, I made my script draw a handy illustration of the LFSRs.



Bytes of the char array are horizontal increasing indexes left to right, bits vertical with increasing indexed top-down. The homogeneous squares constitute the four LFSRs, and the squares half-yellow depend on the LFSR of their other color. Just for reference I also framed with yellow border byte 0 and byte 7 of each LFSR block, as these are used when extracting bits as described above in the discussion of the encryption algorithm.

Interestingly, some of the orphan bits are included in extraction of entropy from the LFSR. Just to add a bit of confusion, `claripy`'s bitvector considers such char arrays as one big-endian value, which means that bit 127 is the bottom bit of byte 0 (the bottom left-most bit) and the least significant bit of the bitvector is bit 0 of byte 15, thus the top right corner of the diagram.

ACT IV: Revelation

I had everything converted to polynomials and constraints, so I started to try to feed it all directly into Z3, but Z3 seems to be geared more toward non-boolean equations. Working with vectors of booleans was quite tedious. After some long nights, I gave up and started anew in `claripy`, a wrapper around Z3 from the fine Angr people.

With `claripy`, everything went well, I had the first solution! It took nearly two minutes but, alas, it was incorrect! After a few days of debugging my constraints, I finally had the correct solution, and it only took 50 seconds! I defeated the beast! What a symbolic execution!

All you need to do is feed the solver 17 bytes of ciphertext, and the solver will either declare that the ciphertext cannot be the output of the PX1000cr algorithm, or it outputs the encryption key and the decrypted 17 bytes of plaintext. The rest of the plaintext can be recovered by decrypting the ciphertext with the recovered key. With a little change it is also possible to solve keys for shorter ciphertexts, but then there will be multiple key candidates which must be tested by the user. The number of key candidates in that case is $2^{17-\min(\text{len(ciphertext)},17)}$.

Looking at my script I realized I could keep everything symbolic, pre-computing all constraints, and with this change a speed-run is possible. With this, calculating the solution takes now less than four seconds!

I invite everyone to download the emulator and run the ROM themselves and plug the ciphertext into the solution. With a few changes you can even calculate things backwards, like what plaintext and key combination generates the following ciphertext: "`(NSA backdoor fun`".

I do not know if the NSA had a SAT solver like Z3 back in 1983, but 40 years later the fact is I can recover a key within seconds in a single thread on a laptop CPU. I am far from being able to do so if DES were used. This lets me confirm that the PX1000cr algorithm is indeed a backdoor.

Finally I would like to thank Ben, Phr3ak, the Crypto Museum people, Jonathan, Antoine, the Angr devs, Asciimoo and Dnet for their support!