Identifying crypto[1] functions

When reverse engineering programs you might encounter code that make use of various cryptographic functions. These functions can be both large and difficult to to understand making you waste valuable time on reverse engineering them. This article will explain a few methods to more easily identify some of the most popular cryptographic functions which will hopefully save you time in your reverse engineering efforts.

Constants

The first and easiest way to identify some cryptographic functions is to utilize the fact that many of these algorithms make use of specific constants in their caluclations. Identifying and looking up these constans can help you quickly identify some algorithms. For example the MD5 hashing algorithm initializes a state with the following four 32bit values: `0x67452301`, `0xefcdab89`, `0x98badcfe`, `0x10325476`. Be careful though since SHA1 also uses these four values but additionally it uses `0xc3d2e1f0` in its initialization. Another thing to look out for is some optimizations. Several algorithms, including the XTEA block cipher, adds a constant (`0x9e3779b9` in the XTEA case) in each iteration. Since numbers are represented with two's complement, it means that adding a value X is the same as subtracting $\neg X + 1$, that is the bitwise negation of X, plus one. This means that, in the case of XTEA, you will sometimes instead see that the code substracts `0x61c88647` (since `0x61c88647` $=$ `¬0x9e3779b9` $+ 1$), thus if you try to look up a constant and get no results, try searching for the inverse of that constant (plus one) as well.

```
; these two are the same
add edx, 0x9e3779b9
sub edx, 0x61c88647
```

Popular algortihms that make use of specific constants include: MD5, SHA1, SHA2, TEA and XTEA

Tables

Closely related to algorithms that use specific constants are algorithms that use look-up tables for computations. While the individual values in these tables usually are not that special as they are typically indices or permutations of a sequence, the sequence it self is often unique to that specific algorithm. For example, the substitution box, S-box, for AES encryption looks like this:

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
...								...								

Searching for a subset of this table, such as "63 7c 77 7b f2 6b", will reveal that this is the Rijndael (the name of the AES algorithm) S-box. Popular algortihms that make use of look-up tables include: AES, DES and Blowfish.

[1] Crypto stands for cryptography

RC4

Although not recommended anymore due to cryptographical weaknesses, the RC4 chiper still shows up in a lot of places, possibly due to its simplicity. The full key scheduling and stream cipher implemented in Python are shown below. The pattern to look out for here is the two loops in the key scheduling algorithm where the first one creates a sequence of the numbers $[0, 255]$ and the second one swaps them around based on the key.

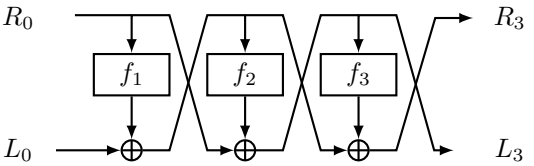
```
S, j = range(256), 0
for i in range(256):
    j = (j + S[i] + key[i % keylength]) % 256
    S[i], S[j] = S[j], S[i]  # swap
```

The actual key stream is then generated by swapping items around in the table and using them to select an element as a key byte.

```
i,j = 0,0
for b in data:
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    S[i], S[j] = S[j], S[i]  # swap
    yield b ^ S[(S[i] + S[j]) % 256]
```

Feistel networks

A popular pattern to look out for in cryptographic code is a Feistel Network. The general idea is that the input is split into two halves, one of them is fed into a function whose output is XOR:ed with the other half before they halves finally swap places. This is repeated a certain number of times, commonly 16, 32 or 64. The diagram below illustrates a three round Feistel Network. Identifying this pattern can help in narrowing down which algorithm you are reversing.



Be careful

Finally, look out for slightly modified algorithms. Techniques described above gives you good heuristics for identifying crypto algorithms. However, sometimes authors make small adjustments to them to waste your time. You might incorrectly identify a piece of code as, for example, SHA1 and just use a SHA1 library function in a unpacker script you are writing separately when in reality a slight adjustment has been made to the algorithm to make it produce completely different outputs. This of course destroys any security guarantees of the algorithm but in some scenarios that is of less importance. This means that if you use these techniques and experience issues, verify the functions by comparing the input and output with an off-the-shelf version of the algorithm you believe to have identified.