Whirlpool

Stanisław Barański

Podstawy kryptografi

1 Introduction

The goal of this project is to find a preimage for the output of the Whirlpool-like hash function. Formally, finding x for defined upfront $h = H(x)$, where H is a Whirpool-like function—modification of the original Whirlpool [1] function.

2 Whirlpool

Whirlpool is a hash function with a cipher-block-based compression function, meaning that its hash code results from encrypting message blocks using a recursive scheme. The output of encryption becomes an encryption key for the next block, the so-called chaining variable.

Whirlpool compression function is an encryption algorithm based on AES; it takes a 512-bit block of plaintext and a 512-bit key as input and produces a 512-bit block of ciphertext as output. The encryption algorithm involves the use of four transformations: Add Key (AK), Substitute Bytes (SB), Shift Columns (SC), and Mix Rows (MR)

This Whirlpool-like function differs from the original Whirlpool by using different padding, smaller hash code (128-bit over 512-bit), different irreducible polynomial (0x12B over 0x011D), different S-box, Diffusion Matrix, and Round Keys.

3 Usage

In order to compile and execute the program, Rust toolchain (cargo) needs to be installed. The easiest way to install it is through rustup.rs. In order to run the program in debug mode execute

```
cargo run
```

it will expect the input string on stdin. Or just

```
echo -n Hello World | cargo run
```

Alternatively the input can be passed as an argument

```
cargo run -- "Hello World"
```

In order to execute the project goal (finiding the preimages) execute

```
cargo run --bin reverse-hash --release
```

4 Implementation

I decided to implement the program in Rust—relatively new (released on July 7, 2010) programming language. Rust offers C-level performance, a borrow checker, an excellent type system, and a modern toolchain, making it the most loved programming language of 2020 (according to StackOverflow 2020 Developer Survey. It is also a popular choice for new projects where cryptography is involved.

Besides std, I used one external library(Rayon) to achieve easy multi-threaded execution.

5 Multiplication in GF field

Efficient modulo multiplication in GF was the most challenging part of the implementation. I started with the most straightforward implementation of multiplying two polynomials.

$$c = (\sum a_i x^i) \otimes (\sum b_j x^j) = \bigoplus_{i=0} \bigoplus_{j=0} (a_i \otimes b_i) x^{i+j}$$

then calculating $c \pmod{x^8 + x^5 + x^3 + x + 1}$ using Long Polynomial Division and taking the remainder. It worked but required a lot of computations.

First, I noticed multiplying two polynomials of degree up to 7 results with a polynomial of degree up to 14.

For example, multiplying two polynomials (FF) $\otimes$ (FF) equals

$$\begin{aligned}
&(x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0) \otimes (x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0)\\
&= (x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7)\\
&\oplus (x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6)\\
&\oplus (x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5)\\
&\oplus (x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4)\\
&\oplus (x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3)\\
&\oplus (x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2)\\
&\oplus (x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1)\\
&\oplus (x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0)\\
&= x^{14} + x^{12} + x^{10} + x^8 + x^6 + x^4 + x^2 + 1
\end{aligned}$$

Therefore I hardcoded the multiplications reducing the execution time by -50%.

```
// arr and brr are first and second polynomials
encoded as arrays of coefficients.
// out is the resulting polynomial.
```

```
out[14] = arr[7] & brr[7];
out[13] = (arr[7] & brr[6]) ^ (arr[6] & brr[7]);
out[12] = (arr[7] & brr[5]) ^ (arr[5] & brr[7]) ^ (arr[6] & brr[6]);
out[11] = (arr[7] & brr[4]) ^ (arr[4] & brr[7]) ^ (arr[6] & brr[5]) ^
out[10] = (arr[7] & brr[3])
    ^ (arr[3] & brr[7])
    ^ (arr[6] & brr[4])
    ^ (arr[4] & brr[6])
    ^ (arr[5] & brr[5]);
out[9] = (arr[7] & brr[2])
    ^ (arr[2] & brr[7])
    ^ (arr[6] & brr[3])
    ^ (arr[3] & brr[6])
    ^ (arr[5] & brr[4])
    ^ (arr[4] & brr[5]);
out[8] = (arr[7] & brr[1])
    ^ (arr[1] & brr[7])
    ^ (arr[6] & brr[2])
    ^ (arr[2] & brr[6])
    ^ (arr[5] & brr[3])
    ^ (arr[3] & brr[5])
    ^ (arr[4] & brr[4]);
out[7] = (arr[0] & brr[7])
    ^ (arr[7] & brr[0])
    ^ (arr[6] & brr[1])
    ^ (arr[1] & brr[6])
    ^ (arr[5] & brr[2])
    ^ (arr[2] & brr[5])
    ^ (arr[3] & brr[4])
    ^ (arr[4] & brr[3]);
out[6] = (arr[0] & brr[6])
    ^ (arr[6] & brr[0])
    ^ (arr[5] & brr[1])
    ^ (arr[1] & brr[5])
    ^ (arr[4] & brr[2])
    ^ (arr[2] & brr[4])
    ^ (arr[3] & brr[3]);
out[5] = (arr[0] & brr[5])
    ^ (arr[5] & brr[0])
    ^ (arr[4] & brr[1])
    ^ (arr[1] & brr[4])
    ^ (arr[3] & brr[2])
    ^ (arr[2] & brr[3]);
out[4] = (arr[0] & brr[4])
    ^ (arr[4] & brr[0])
```

```
            ^ (arr[3] & brr[1])
            ^ (arr[1] & brr[3])
            ^ (arr[2] & brr[2]);
        out[3] = (arr[0] & brr[3]) ^ (arr[3] & brr[0]) ^ (arr[2] & brr[1]) ^
        out[2] = (arr[0] & brr[2]) ^ (arr[2] & brr[0]) ^ (arr[1] & brr[1]);
        out[1] = (arr[0] & brr[1]) ^ (arr[1] & brr[0]);
        out[0] = arr[0] & brr[0];
```

I noticed that calculating modulo in finite field can be achieved by simple substitutions that hold congruence property over irreducible polynomial p (in my case $x^8 + x^5 + x^3 + x + 1$).

$$x^8 + x^5 + x^3 + x + 1 \equiv 0 \pmod{p}$$
$$x^8 \equiv x^5 + x^3 + x + 1 \pmod{p}$$

Whenever polynomial c contains monomial x^8, it can be replaced with polynomial $x^5 + x^3 + x + 1$ while still holding congruence property.

The same can be done with all monomials of degree $> 8 = degree(p)$

$$x^9 = x^8 * x = (x^5 + x^3 + x + 1) * x \equiv x^6 + x^4 + x^2 + x \pmod{p}$$
$$x^{10} = x^9 * x = (x^6 + x^4 + x^2 + x) * x \equiv x^7 * x^4 * x^3 + x^2 \pmod{p}$$
$$x^{11} = x^{10} * x = (x^7 * x^4 * x^3 + x^2) * x = x^8 + x^5 + x^4 + x^3 \equiv x^4 + 1 \pmod{p}$$
$$x^{12} = x^{11} * x = (x^4 + 1) * x \equiv x^5 + x \pmod{p}$$
$$x^{13} = x^{12} * x = (x^5 + x) * x \equiv x^6 + x^2 \pmod{p}$$
$$x^{14} = (x^6 + x^2) * x \equiv x^7 + x^3 \pmod{p}$$

By using these substitutions, I reduced the execution time dramatically, about -80%.

```
// x^8 = x^5 + x^3 + x + 1
if out[8] {
    out[5] ^= true;
    out[3] ^= true;
    out[1] ^= true;
    out[0] ^= true;
}

// x^9 = x^6 + x^4 + x^2 + x
if out[9] {
    out[6] ^= true;
    out[4] ^= true;
    out[2] ^= true;
    out[1] ^= true;
}
// x^10 = x^7 + x^4 + x^3 + x^2
```

```
if out[10] {
    out[7] ^= true;
    out[4] ^= true;
    out[3] ^= true;
    out[2] ^= true;
}
// x^11 = x^4 + 1
if out[11] {
    out[4] ^= true;
    out[0] ^= true;
}
// x^12 = x^5 + x
if out[12] {
    out[5] ^= true;
    out[0] ^= true;
}
// x^13 = x^6 + x^2
if out[13] {
    out[6] ^= true;
    out[2] ^= true;
}
// x^14 = x^7 + x^3
if out[14] {
    out[7] ^= true;
    out[3] ^= true;
}
```

6 Results

The attacks were performed on my server machine equipped with a processor Intel i7-4790 (8) @ 4.000GHz, 16GB (2*8GB) of DDR3 RAM, running Arch Linux operating system with kernel version 5.11.8. For compilation I used cargo 1.51.0 (43b129a20 2021-03-16).

Input length	Preimage found	Execution time
2	0J	25.635827ms
3	Ss1	1.89898985s
4	@@-8	172.601573451s
5	qDP0Z	10431.405587416s
6	Not Found	Not Found

Unfortunately, I did not manage to find a preimage for six-digit input over five days. Based on the five-digit attack, the average time for calculating one hash took $10431.405587416s/79^5 = 0.00000339005s$. Therefore six-digit attack could take $79^6 * 0.00000339005s = 824078.628589s = 824078.628589s = 9.53794709015days$.

References

1. Stallings, W.: The whirlpool secure hash function. Cryptologia **30**(1), 55–67 (2006)