Crypto TP : Multicollisions for narrow-pipe Merkle-Damgård hash functions

Elliot Renel

April 29th 2022

1 Theoretical Study

Q.1: $\mathcal{F}$ has an output space of size 2^n. Considering that the average time to find a collision for a perfect hash function (as per the birthday paradox) is $\sqrt{N}$ for an output space of size N, we therefor have an average time cost of $\sqrt{2^n}$ for each pair $\{m_i^{(0)}, m_i^{(1)}\}$. As each d pairs are computed separately and sequentially, the average time cost of our attack (given the time to compute $\mathcal{F}$ and storing the results $t_{\mathcal{F}}$) is $\Omega(t_{\mathcal{F}} \times d \times \sqrt{2^n})$, with a worst case complexity $O(t_{\mathcal{F}} \times d \times 2^n)$.

Q.2: $\mathcal{H}$ has an output space of size 2^n, and we want 2^d collisions. If we use a buffer of size 2^n, each containing 2^d spaces for input, the complexity of filling one buffer space up to 2^d is $O(t_{\mathcal{F}} \times 2^n \times 2^d) = O(t_{\mathcal{F}} \times 2^{dn})$.

Q.3: No it does not, because of the recursive nature of the Merkle-Damgård hash function, the attack shows that using the compression function, finding the 2^d collisions is equivalent to finding d collisions of an ideal hash functions, which, as we saw, is much more efficient than with $\mathcal{H}$ being ideal.

Q.4: Considering we use the same attack and model as question 1, the only difference being the input and output space sizes of $\mathcal{F}$, we end up with the complexities $\Omega(t_{\mathcal{F}} \times d \times \sqrt{2^{2n}})$ and $O(t_{\mathcal{F}} \times d \times 2^{2n})$. We can note that the average complexity $\Omega(t_{\mathcal{F}} \times d \times \sqrt{2^{2n}}) = \Omega(t_{\mathcal{F}} \times d \times 2^n)$ is equivalent to the worst case complexity of the narrow pipe attack, while the worst case is close to the ideal hash attack complexity with $d = 2$. But for higher values of d the hash function is still worse.

2 Implementation

To run the code, you need to compile with *make* (you can add some extra comment by doing *make -B VERBOSE=1*), then you just have to launch it by typing *./attack d*, with d the number of sections. The code is in the folder *code* present in the archive.

2.1 The Data Structure

```c
typedef struct hmap{
    uint8_t *value;
    uint32_t size;
    uint8_t elem_size;
};
```

2.2 Collision of $\mathcal{F}$

```c
void find_col(uint8_t h[6], uint8_t m1[16], uint8_t m2[16]){
    hmap *map = malloc(sizeof(hmap));
    map->elem_size = 16;
    map->size = UINT32_MAX;
    map->value = malloc(map->size*map->elem_size*sizeof(uint8_t));

    uint8_t th[6];
    uint64_t c = 0;
    while(++c<16777216*3){ // 16777216 is 2^24, aka sqrt(2^48)
        copy(h,th,6);
        make_random(m1);
        tcz48_dm(m1,th);
        if(map->value[((*(uint64_t*)th)*map->elem_size)%UINT32_MAX]!=0 && equals(m1,map->value+(((*(uint64_t*)th)*map->elem_size)%UINT32_MAX),16)==-1 && check_collision(map->value+(((*(uint64_t*)th)*map->elem_size)%UINT32_MAX),h,th)==1){
            copy(map->value+(((*(uint64_t*)th)*map->elem_size)%UINT32_MAX),m2,16);
            copy(th,h,6);
            break;
        }
        copy(m1,map->value+(((*(uint64_t*)th)*map->elem_size)%UINT32_MAX),16);
    }
    free(map->value);
    free(map);
}
```

2.3 The full attack

```c
void attack(int d){
    uint8_t **m1 = (uint8_t **)malloc(d*sizeof(uint8_t*));
    uint8_t **m2 = (uint8_t **)malloc(d*sizeof(uint8_t*));
    uint8_t *h = malloc(sizeof(uint8_t)*6);
    h[0] = IVB0;h[1] = IVB1;h[2] = IVB2;h[3] = IVB3;h[4] = IVB4;h[5] = IVB5;

    for(int i=0; i<d; i++){
        m1[i] = malloc(sizeof(uint8_t)*16);
        m2[i] = malloc(sizeof(uint8_t)*16);
        find_col(h,m1[i],m2[i]);
    }
    print_2powN(m1,m2,d); // prints all the combinations of m1 and m2
```

```
    for(int i=0; i<d; i++){
        free(m1[i]);
        free(m2[i]);
    }
    free(m1);
    free(m2);
}
```

2.4 Performance compared the theoretical complexity

I am running the code on my laptop with a ryzen 5 amd cpu (6 cores 12 threads), 8 GB of ram, on an ubuntu 20.04.

Doing one loop in *find_col* lasts 1.2 microseconds on average, so we'll use that as a base for $t_{\mathcal{F}}$. If we take the average complexity $\Omega(t_{\mathcal{F}} \times d \times \sqrt{2^n})$ and use $d = 1$, knowing that $n = 48$, we arrive at the average time being $1.2 \times 1 \times 2^{\frac{48}{2}} = 20132659.2\mu s \approx 20.13s$. Using the same values, can calculate the worst case complexity to be $1.2 \times 1 \times 2^{48} \approx 337769972s \approx 11\,years$. From here, we can just multiply by d as the other values don't change.

We are going to set the seed for simplicity and compare. Running the attack with $d = 1$, we obtain a time of around 17.65 seconds, which is better than the average predicted. Now if we try $d = 4$, we get a time of 91.67 seconds, slightly higher than the expected average of 80 seconds. Increasing d to 16, we end up with a time of 368 seconds, or just above 6 minutes, which is again slightly higher than the expected average of 320 seconds (5 minutes 20 seconds).