Cracker

A searching framework for CTF

Haijiang Xie
Department of Computer Science and Engineering
Shanghai JiaoTong University
hjxie04@gmail.com

Pan Chen
Department of Computer Science and Engineering
Shanghai JiaoTong University
chenpan5762337@126.com

Yuzhu Wang
Department of Computer Science and Engineering
Shanghai JiaoTong University
hfut0830@sjtu.edu.cn

ABSTRACT

To crack the software and write a keygen in CTF(*Capture The Flag*), we need to understand the encryption (or decryption) algorithms in the binary by both static disassembling and dynamic analysis.

In this paper, we construct a parallel searching framework to solve this problem in a more efficient way. We choose the CRC64 as the encrytion algorithm here. Based on the already known outputs, we need to design a parallel searching algorithm to find the original input of this algorithm.

Keywords

CRC, CTF, MITM, OpenMP, MPI, Parallel

1. INTRODUCTION

Capture the Flag (CTF)[1] is a special kind of information security competitions. T-here are three common types of CTFs: Jeopardy, Attack-Defence and mixed.Jeopardy-style CTFs has a couple of questions (tasks) in range of categories. For example, Web, Forensic, Crypto, Binary or something else. Team can gain some points for every solved task. More points for more complicated tasks usually. The next task in chain can be opened only after some team solve previous task. Then the game time is over sum of points shows you a CTF winer. Famous example of such CTF is Defcon CTF quals. Well, attack-defence is another interesting kind of competitions. Here every team has own network(or only one host) with vulnarable services. Your team has time for patching your services and developing exploits usually. So, then organizers connects participants of competition and the wargame starts! You should protect own services for defence points and hack opponents for attack points. Historically this is a first type of CTFs, everybody knows about DEF CON CTF - something like a World Cup of all other competitions.

CTF games often touch on many other aspects of information security: cryptography, stego, binary analysis, reverse engeneering, mobile security and others. Good teams generally have strong skills and experience in all these issues.

Reverse engineering[7] is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation. It often involves disassembling something (a mechanical device, electronic component, computer program, or biological, chemical, or organic matter) and analyzing its components and workings in detail, just to re-create it. Reverse engineering is done for maintenance or to create a new device or program that does the same thing, without using original, or simply to duplicate it.

A crackme[3] (often abbreviated by cm) is a small program designed to test a programmer's reverse engineering skills. They are programmed by other reversers as a legal way to "crack" software, since no company is being infringed upon. Crackmes, Reversemes and Keygenmes generally have similar protection schemes and algorithms to those found in commercial protections. However due to the wide use of packers/protectors in commercial software, many crackmes are actually more difficult as the algorithm is harder to find and track than in commercial software.

A Keygenme specifically is designed for the reverser to not only find the algorithm used in the application, but also write a small Keygen in the programming language of their choice. Although, most keygenmes properly manipulated can be self-keygenning.

In this paper, we choose one of the typical type of tasks in CTF Jeopardy competition as our goal to solve. Usually, we need to use brute-force to get the flag after we understand the encryption(decryption) algorithms by statically and dynamically reversing, and also have got the ciphertext. Those algorithms must be cracked later usually have a few of special mathematical or other kinds of disadvantages, which can be choose as the breakthrough points. More narrowly, we have known the encryption function the program used here is a CRC64[1] algorithm with a special crc table.[2]

A cyclic redundancy check (CRC)[4] is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents;

Parallel Computing and Parallel Algorithms (Graduate) Spring 2014, Shanghai

[1]A CRC is called an n-bit CRC when its check value is n bits. For a given n, multiple CRCs are possible, each with a different polynomial. Such a polynomial has highest degree n, which means it has n + 1 terms.

[2]Instead of calculate a bit every step, programmers usually use precomputed CRC table to calculate a byte at a time. In this case, the speedup is nearly up to 8.

on retrieval the calculation is repeated, and corrective action can be taken against presumed data corruption if the check values do not match.

CRCs are so called because the check (data verification) value is a redundancy (it expands the message without adding information) and the algorithm is based on cyclic codes. CRCs are popular because they are simple to implement in binary hardware, easy to analyze mathematically, and particularly good at detecting common errors caused by noise in transmission channels. Because the check value has a fixed length, the function that generates it is occasionally used as a hash function.

The remainder of this paper is organized as follows: We give an overview of our approach in Section 2, and then describe the design and implementation of our framework in Section 3. Section 4 gives our evaluation of performance. Finally Section 5 concludes.

2. THE EXHAUSTIVE SEARCH APPROACH

As we have known both the detail of the algorithms and the ciphertext, what we care more about is the plaintext which become the ciphertext we have got by applying the encryption algorithm. One way to find out the plaintext is to reverse the encryption algorithm functionally. However, it is not a silver bullet all the time. One typical example is one-way function[3], more specifically and commonly, the encryption algorithm is a hash function(e.g., MD5, SHA1, SHA256, etc.).In this case, we can do nothing but guess all of the possible plaintext in a brute-force way and verify our guess.

Now we take the CRC64 as the one-way encryption function and develop a efficient searching algorithm to solve this problem. Then we also can apply it to other functions also have the similar property.

2.1 Assumption

Without any information about the algorithms and the plaintext/ciphertext, we can no nothing but give up. It is however critical that three assumptions hold. These three assumptions are:

2.1.1 Length of text

As the main goal to use CRC is to check the correctness of the plaintext in transmission or compression, only got the length of the plaintext can we go on the follow checking work.

2.1.2 CRC value

Since we aim to find the plaintext corresponding to the ciphertext, it is essential to get the CRC value first. In reality, we can get both the length and the CRC value of the plaintext can be find in protocol packet header and compression software easily.

2.1.3 Visible character

[3]In computer science, a one-way function is a function that is easy to compute on every input, but hard to invert given the image of a random input. Here, "easy" and "hard" are to be understood in the sense of computational complexity theory, specifically the theory of polynomial time problems. Not being one-to-one is not considered sufficient of a function for it to be called one-way.[6]

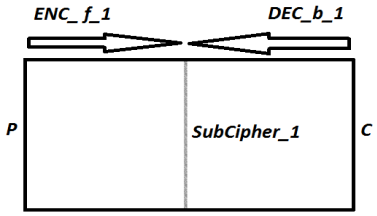

Figure 1: 1D-MITM attack

To simplify the problem, we care about the content we can read directly. That is, we only pay our attention to the visible ASCII characters.

2.1.4 Bytes sum of text(Optional)

One special case we meet is that we have known some information about the plaintext. We have known the sum of all characters in plaintext here, which is useful for us to prune some branch when searching.

2.2 Basic Searching Scheme

Suppose we have got a CRC value, which is the result of one plaintext with n bytes length. Without any information about the plaintext, so we must guess every bit in a brute-force way. The time complexity is $O(2^{8*n})$, where 8 is the length of one byte and n is the number of bytes. As n increase, time complexity increases exponentially. It is hard for us to accept this overhead if $n \geq 5$.

However, we can use some space to exchange the time. We always do this because time is more precious than space. It is a trade-off. Here we have a method called "Meet-in-the-middle".[5]

MITM is a generic attack, applicable on several cryptographic systems. The internal structure of a specific system is therefore unimportant to this attack. An attacker requires the ability to encrypt and decrypt, and the possession of pairs of plaintexts and corresponding ciphertexts. When trying to improve the security of a block cipher, a tempting idea is to simply use several independent keys to encrypt the data several times using a sequence of functions (encryptions). Then one might think that this doubles or even n-tuples the security of the multiple-encryption scheme, depending on the number of encryptions the data must go through. Figure 1 illustrate 1D-MITM attack.

The Meet-in-the-Middle attack attempts to find a value using both of the range (ciphertext) and domain (plaintext) of the composition of several functions (or block ciphers) such that the forward mapping through the first functions is the same as the backward mapping (inverse image) through the last functions, quite literally meeting in the middle of the composed function.

An exhaustive search on all possible combination of keys (simple brute-force) would take $2^{k \cdot j}$ attempts if j encryptions has been used with different keys in each encryption, where each key is k bits long. MITM or MD-MITM improves on this performance. Now the time complexity decrease to $O(2 \cdot 2^{\frac{n}{2}})$. Instead, the space complexity increase to $O(c \cdot 2^{\frac{n}{2}})$, where c is a small constant.

Algorithm 1 shows the basic scheme of search algorithm:

2.3 Parallel Version

Algorithm 1 MITM Search Scheme

```
1: function FORWARD(depth, value)
2:     if depth = HalfLenOfText then
3:         ADDTOTABLE(value)
4:         return
5:     end if
6:     for i ← 1 to LengthOfDict do
7:         newValue ← ENCRYPTROUND(dict[i], value)
8:         FORWARD(depth+1, newValue)
9:     end for
10: end function
11:
12: function BACKWARD(depth, value)
13:     if depth = InputLen − HalfLenOfText then
14:         if FINDVALUEINTABLE(value) = True then
15:             print "Find it!"
16:             return True
17:         else
18:             return False
19:         end if
20:     end if
21:     for i ← 1 to LengthOfDict do
22:         newValue ← DECRYPTROUND(dict[i], value)
23:         if BACKWARD(depth-1, newValue) = True then
24:             return True
25:         end if
26:     end for
27:     return False
28: end function
```

It is obvious to see that each searching procedure with different guess value do not influence each other. That is, each iteration is independent of each other. The *for* loop in $ForwardDFS$ and $BackwardDFS$ can be execute in a parallel way. Given these two algorithms are recursive, so it also means $ForwardDFS$ and $BackwardDFS$ can be parallized. In section 3 there is a detail of parallel implementation.

3. DESIGN & IMPLEMENTATION

Given the large space complexity, our design targets 64-bit x86_64 processors running Linux, and it is the experiment environment here. Combining with the assumptions given in Section 2(Optional assumption also be considered here), Algorithm 2 is a more detailed algorithm than Algorithm 1. Then some implementation aspects discussed in the following.

3.1 Space Overhead

To store the middle results, it must construct a data structure with quickly lookup property. Here we present a simple open hashing structure and the space complexity of each hash table is 128MB. Suppose there are n threads searching simultaneous. Thus, the space overhead is $16 \times 128 = 2GB$.

Since the experiment environment supports 64-bit process, the user-mode virtual address space is larger than 4GB. So we can create more thread or allocate more space for a hash table.

3.2 Parallelization

The searching space can be partition into several parts. Suppose n is the range of seaching space and p is the number

Algorithm 2 Parallel CRC Search Scheme

```
1: function CRACK(nodeId, nodeNum, sum, crcValue, textLen)
2:     gStart ←' 0' ∗ ⌊textLen/2⌋
3:     gEnd ← sum − ('0' × (textLen − ⌊textLen/2⌋))
4:     range ← (gEnd − gStart + 1)
5:     start ← ⌊(nodeId × range)/nodeNum⌋ + gStart
6:     end ← ⌊((nodeId+1) × range)/nodeNum⌋ + gStart
7:
8:     #pragma omp parallel
9:     table ← []
10:    #pragma omp parallel for
11:    for i ← start to end do
12:        FORWARD(0, i, 0)
13:        if BACKWARD(textLen-1, sum-i, crcValue) then
14:            break
15:        end if
16:    end for
17: end function
18:
19: function FORWARD(depth, sum, value)
20:    if depth = HalfLenOfText then
21:        if sum = 0 then
22:            ADDTOTABLE(value)
23:        end if
24:        return
25:    end if
26:    for i ← 1 to LengthOfDict do
27:        newValue ← ENCRYPTROUND(dict[i], value)
28:        newSum ← sum − dict[i]
29:        if newSum < 0 then
30:            return
31:        end if
32:        FORWARD(depth+1, newValue)
33:    end for
34: end function
35:
36: function BACKWARD(depth, sum, value)
37:    if depth = InputLen − HalfLenOfText then
38:        if sum = 0 then
39:            if FINDVALUEINTABLE(value) = True then
40:                print "Find it!"
41:                return True
42:            else
43:                return False
44:            end if
45:        else
46:            return False
47:        end if
48:    end if
49:    for i ← 1 to LengthOfDict do
50:        newValue ← DECRYPTROUND(dict[i], value)
51:        newSum ← sum − dict[i]
52:        if newSum < 0 then
53:            return False
54:        end if
55:        if BACKWARD(depth-1, newValue) = True then
56:            return True
57:        end if
58:    end for
59:    return False
60: end function
```

of processes. The block allocation scheme[2] we use here is that:

The first element controlled by process i is

$$start = \lfloor in/p \rfloor$$

The last element controlled by process i is the element immediately before the first element controlled by process $i+1$:

$$end = \lfloor (i+1)n/p \rfloor - 1$$

The process controlling a particular element j is

$$nid = \lfloor (p(j+1) - 1)/n \rfloor$$

In this decomposition way, we can divide the searching space into p parts. As there are 14 nodes in the cluster, p is 14 at most. In other words, the *start* and *end* arguments in $CRACK$ function signature is determined. With id of each node and total number of nodes in cluster, we can calculate different *start* and *end* for different node in cluster.

However, it is easy to find out that the for loop in $FORWARD$ and $BACKWARD$ can be parallelized, too. So we use OpenMP to parallelize them here. One important thing need to be noticed is the hash table in different threads should be declared as thread-private.

4. EVALUATION

We implement and run this searching framework in a cluster with 14 nodes, where each node runs in CentOS 6.5 with 24 processors.

4.1 Performance

We seperately create 100 random strings with length of 4 (i.e., each string has 4 ASCII characters.). Then we calculate the crc value of each string and use the crc value and the length of string to search the original string. To show the efficiency, the time cost t_i has been recorded during each searching procedure. When all of the 100 strings have been found out successfully, we use the average time cost to evaluate the perfermance.

$$t_{aver} = \frac{1}{100} \sum_{i=1}^{100} t_i$$

By the same way, we also get the average time cost when the length of each random string is 5, 6, 7 and 8. In each case decribed above, there are 100 random strings with corresponding length and use the same scheme to search.

Because the experiment environment is public and testing a longer string may cost more time, we could not get exact result when the length of random string is larger than 9 if other also run their program at the same time. Here we only present the result of 4,5,6,7,8,9.

Here we choose four compiling modes to evaluate the result, record the time cost and compare the result of each mode. Table 1 shows those four modes.

Figure 2 shows the time cost with different string length. As the length of string grow, the average time cost of NONE mode increase faster than others. From the result, we can find that the increasing speed OMP_MPI mode is the slowest and OMP mode and MPI mode is a little higher than OMP_MPI mode. However, all of these three parallel modes increase much slower than NONE modes.

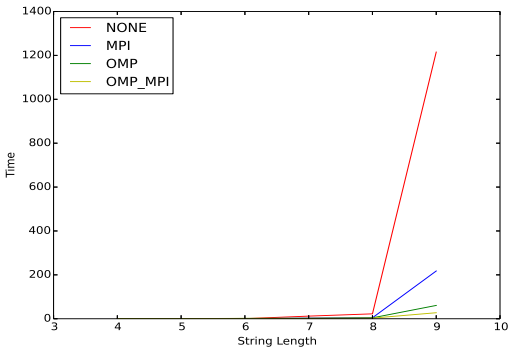

Figure 2: Average time cost with different string length

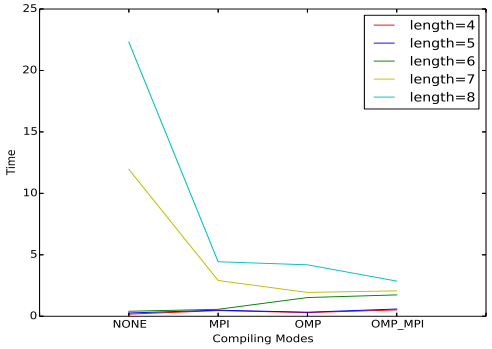

Figure 3: Average time cost in different compiling modes

Figure 3 shows the power of parallelization. When the string length is small, it may be hard to show the powerful ability of parallelization because of the communication overhead accounting for a large proportion. As the string length increase to 7, we can find that the time cost decrease significantly. Similar to the result in Figure 2, the time cost in OMP_MPI mode is lowest.

Mode Type	Compiling Mode
NONE	g++ -O3 -o
MPI	mpic++ -O3 -o
OMP	g++ -O3 -fopenmp -o
OMP_MPI	mpic++ -O3 -fopenmp -o

Table 1: Four Modes

5. CONCLUSIONS

This paper introduce a parallel searching scheme for CTF, which is efficient to apply to the problem that need to be solved in a brute-force way. We have used MPI and OpenMP to implement a search framework. The experiment result shows that the speedup of parallelization is significant.

6. REFERENCES

[1] CTFtime. All about ctf. In *https://ctftime.org/ctf-wtf/*.

[2] M. J. Quinn. Data decomposition options. In *Parallel Programming in C with MPI and OpenMP*, pages 117–119. McGraw-hill Companies, Inc., 2003.

[3] Wikipedia. Crackme. In *http://en.wikipedia.org/wiki/Crackme*.

[4] Wikipedia. Cyclic redundancy check. In *http://en.wikipedia.org/wiki/Cyclic_redundancy_check*.

[5] Wikipedia. Meet-in-the-middle attack. In *http://en.wikipedia.org/wiki/Meet-in-the-middle_attack*.

[6] Wikipedia. One-way function. In *http://en.wikipedia.org/wiki/One-way_function*.

[7] Wikipedia. Reverse engineering. In *http://en.wikipedia.org/wiki/Reverse_engineering*.