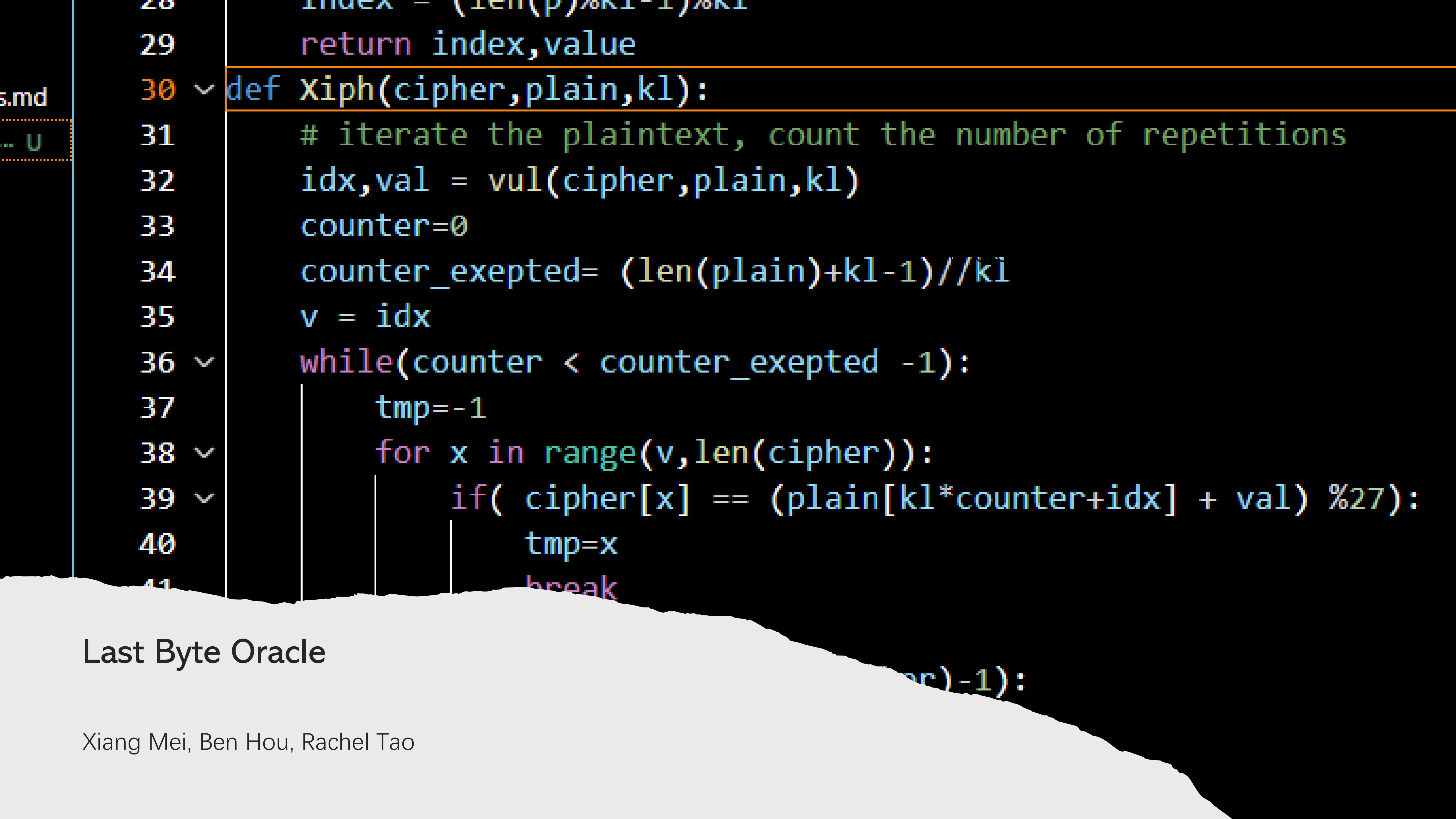



# Last Byte Oracle

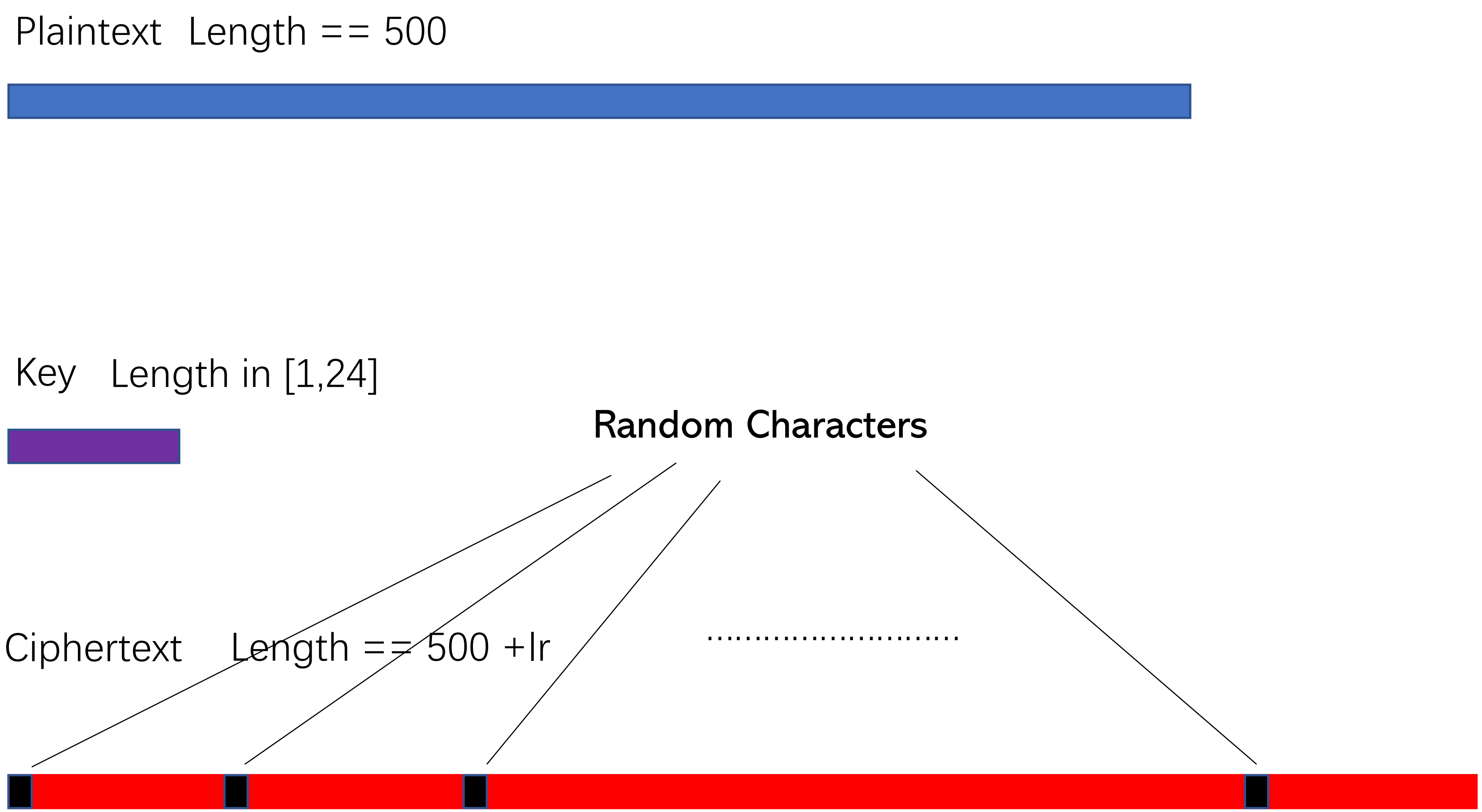
Xiang Mei, Ben Hou, Rachel Tao

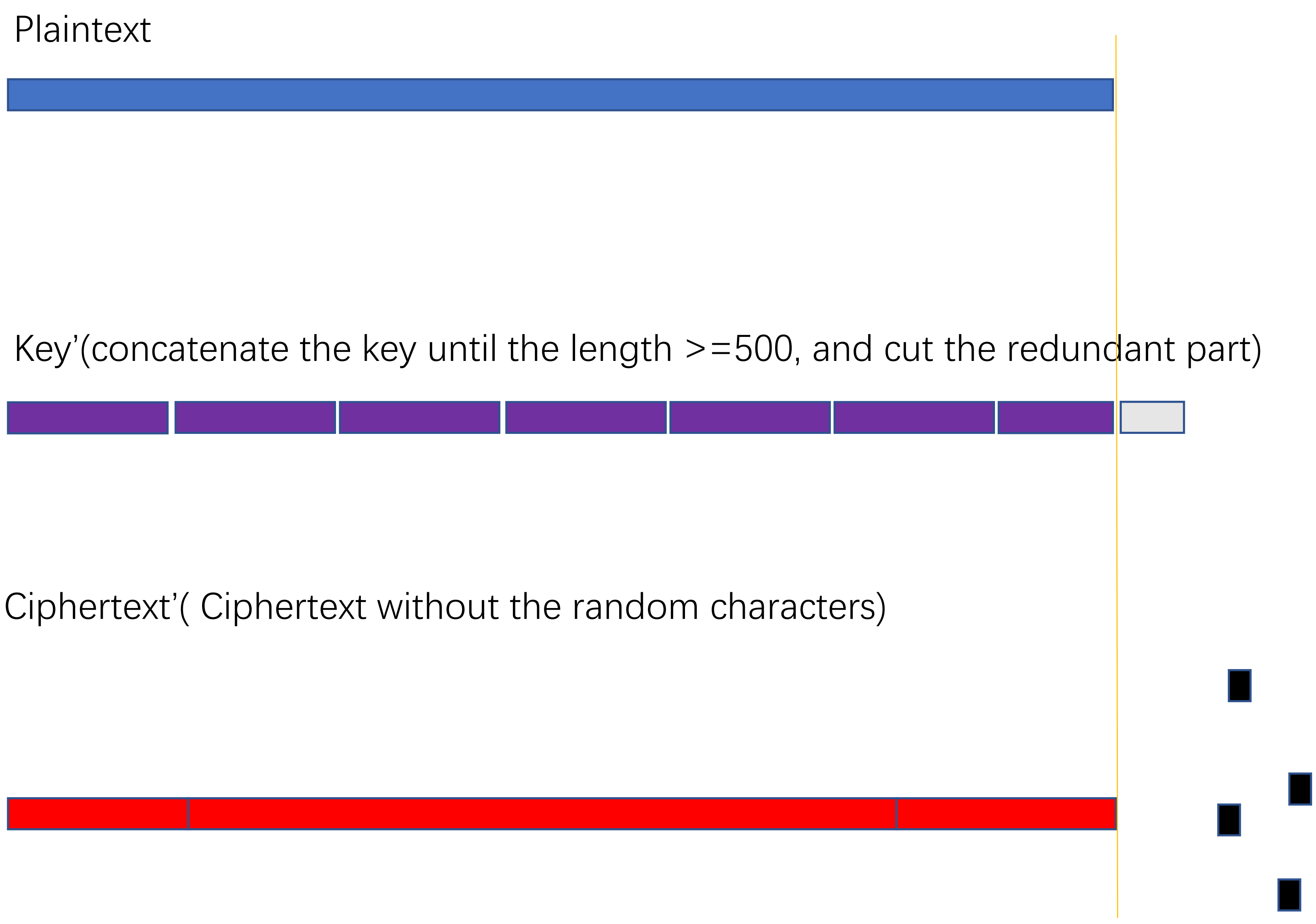
# Outline

- Project Recall
- A 20 lines solution
- Performance
- New Challenges

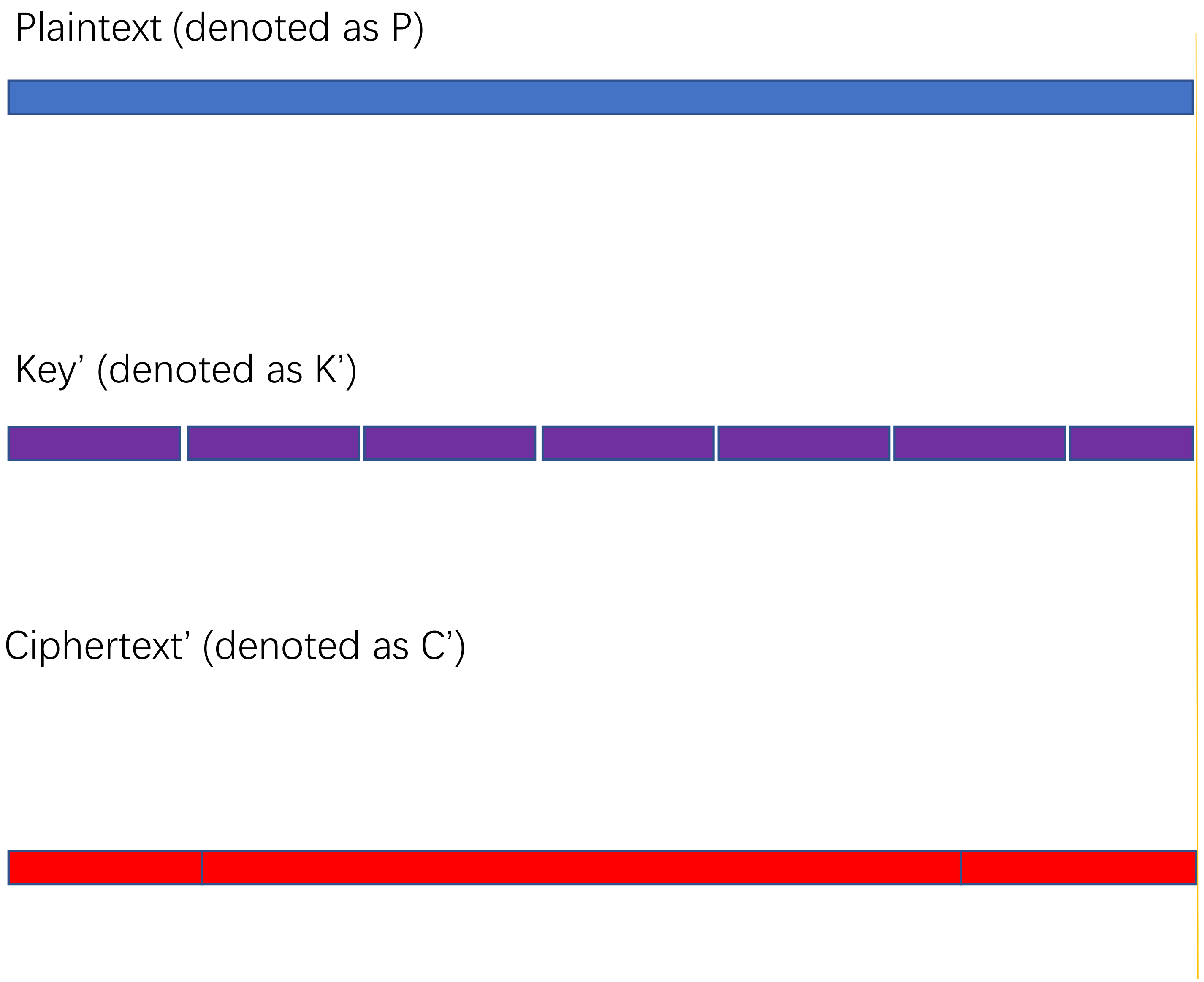
# What do we know?

We know 5 plaintext candidates, the encryption scheme, and the ciphertext.

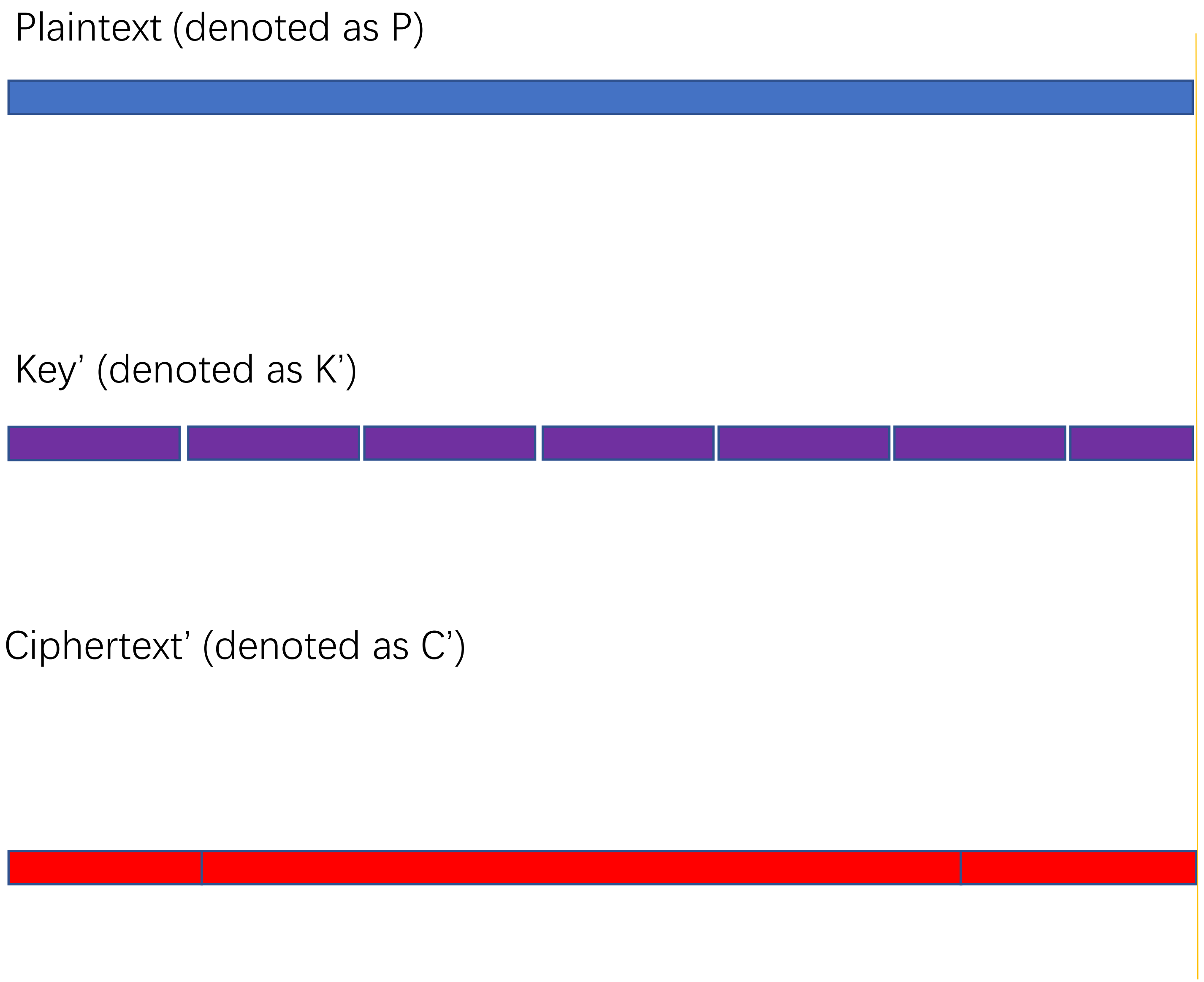

Plaintext Length == 500
Key Length in [1,24]
Random Characters
Ciphertext Length == 500 +lr
..........................




If We have an Oracle which knows the location of each random character, we could eliminate these characters.



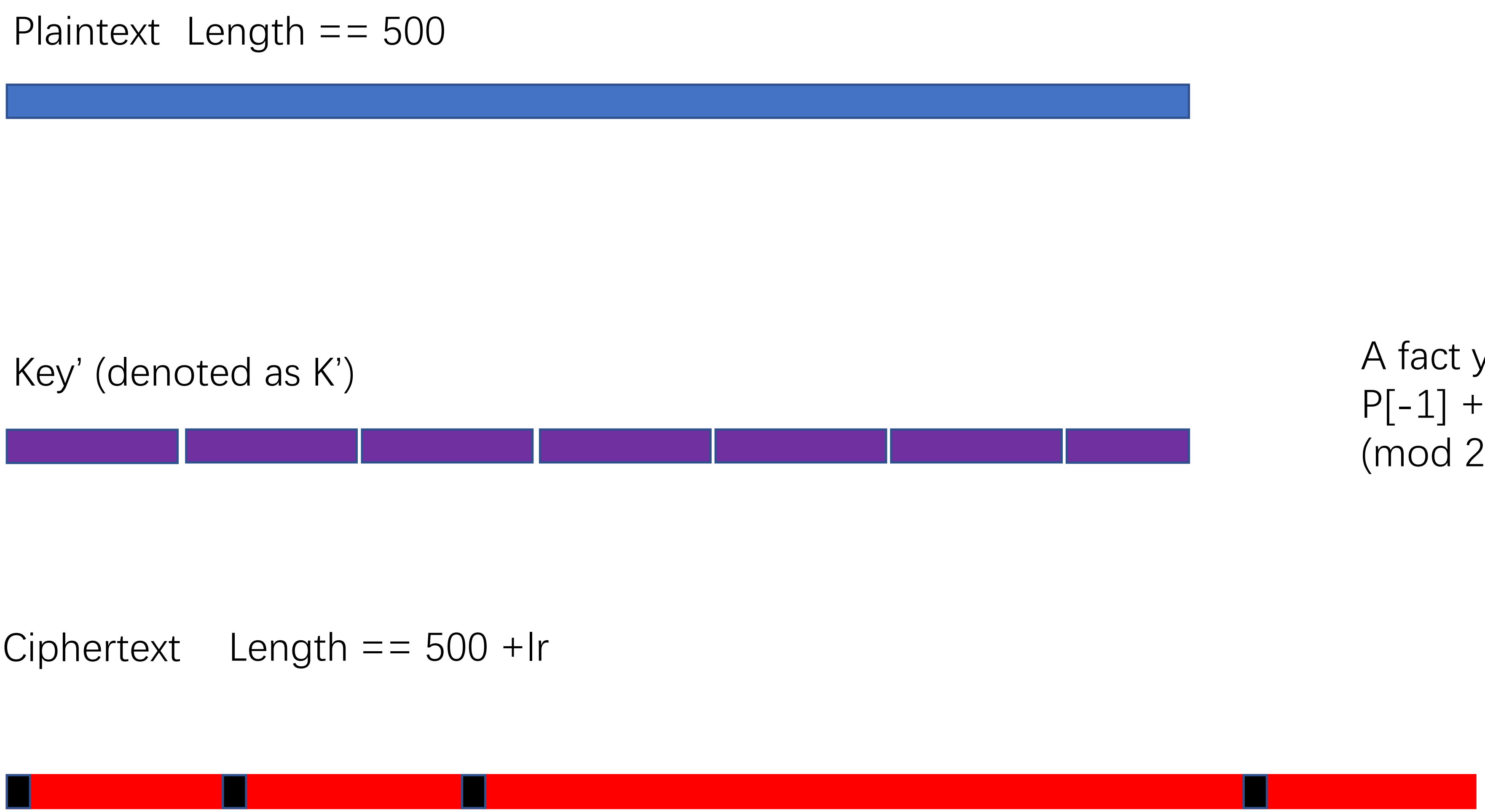

- Shift(P[x],K’[x])=C’[x]
- It’s easy to revers if there is no randomness




- **If randomness exists**
- **Statistical (It’s hard because of randomness)**
- **Brute force: Combination(500+lr,lr)**

How can I make advantages ?

Vulnerability Discovery and Exploit




A fact you did not notice:
$P[-1] + K'[-1] = C[-1] \pmod{27}$

$$P[-1] + K'[-1] = C[-1]$$

---

$$K'[-1] = K[IDX], IDX = (500-1) \bmod Len(K)$$
$$K'[IDX] = K'[-1] = Val, Val = (C[-1] - P[-1]) \bmod 27$$

*How do we **eliminate the 4 bad candidates**?*

*In the scheme, the original key is repeatedly used.(We could take advantage from it!)*

---

**How many times** has the **key** been used?

$(500 + len(k) - 1)//len(k)$ , for example, $len(k) = 24 \rightarrow (500+23)//24 = 21$

---

**How many times** *has the $K[IDX]$ been used?*

$(500 + len(k) - 1)//len(k)$ or $(500 + len(k) - 1)//len(k) - 1$

An exploit comes out!

**We could use the "additional" information to build a filter**

-----------------------------------------

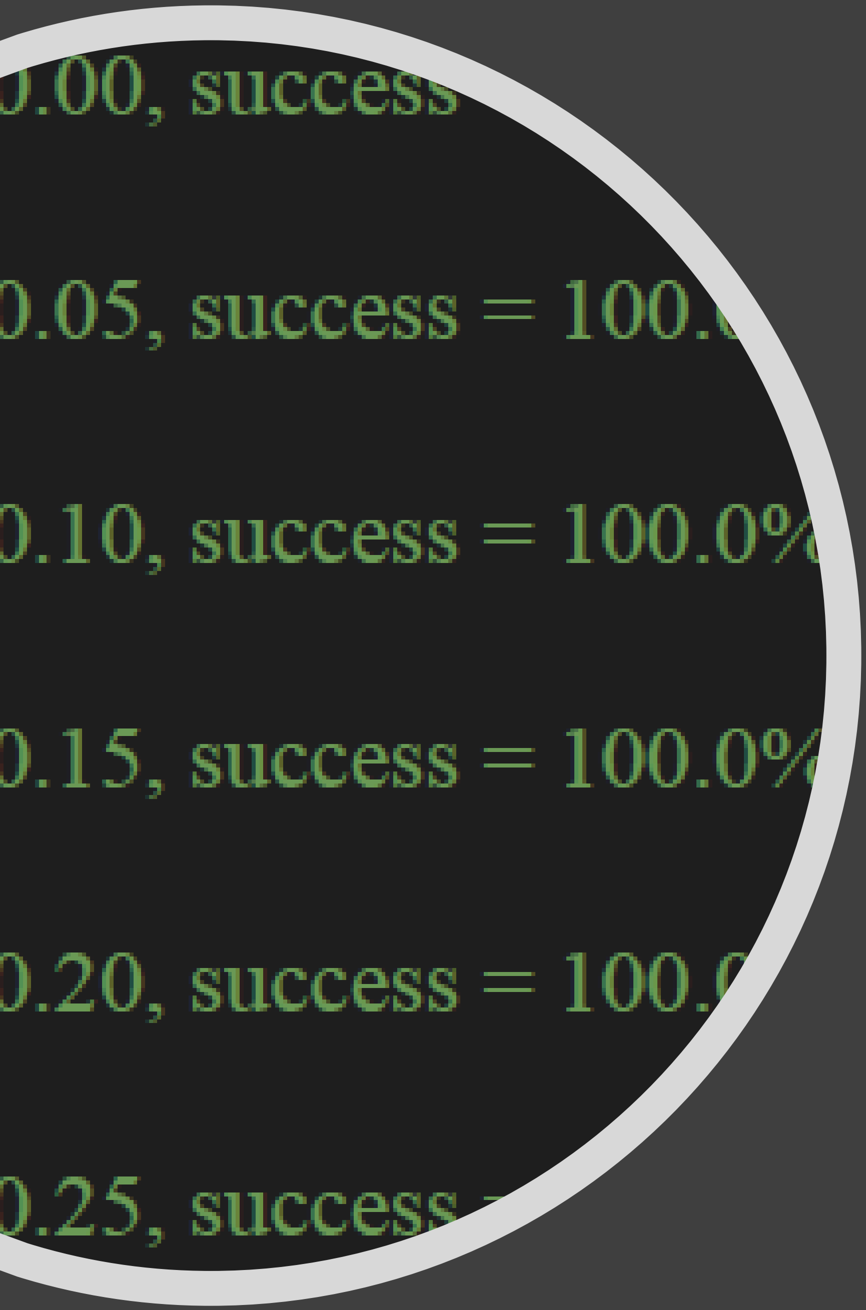
Compare the $expected\_counter$ to the $counter$.

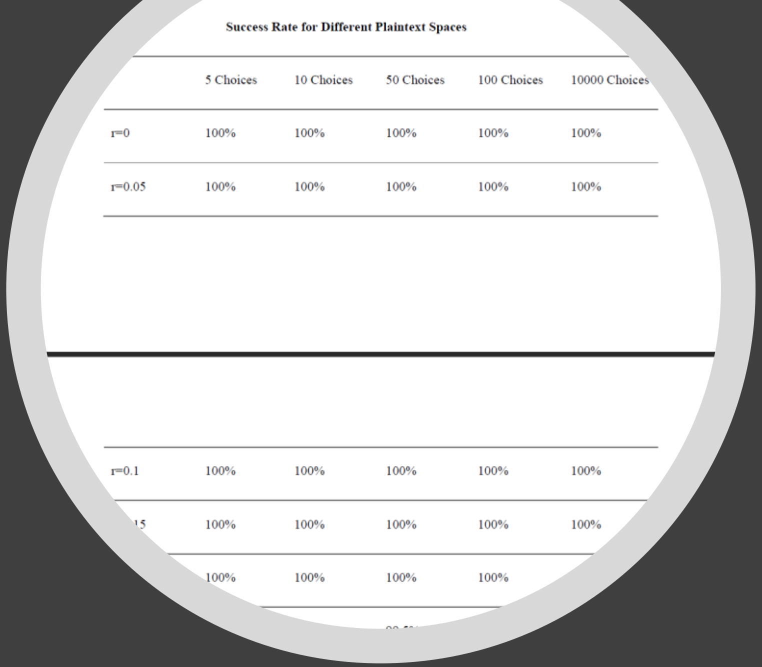
If a candidate can pass the filter, it maybe the plaintext
If can't, pass the filter. It's NOT the plaintext (100%)

-----------------------------------------

What about the possibility of fail.
~=0.015

-----------------------------------------





How about the performance?
Fast and precise!
I can do it all day!

# Result For Part One

```
# r = 0.00, success = 100.0%
# r = 0.05, success = 100.0%
# r = 0.10, success = 100.0%
# r = 0.15, success = 100.0%
# r = 0.20, success = 99.96%
# r = 0.25, success = 99.60%
```

```
# r = 0.00, success = 100.0%
# r = 0.05, success = 100.0%
# r = 0.10, success = 100.0%
# r = 0.15, success = 100.0%
# r = 0.20, success = 100.0%
# r = 0.25, success = 99.76%
```

- The longest time for our program is 281.1690323352814 seconds for testing 10000 test cases.
- No matter if the last byte of the ciphertext is a random character, I can give the correct answer with success rate ~=100%

# Extra Credit

Success Rate for Different Plaintext Spaces

| | 5 Choices | 10 Choices | 50 Choices | 100 Choices | 10000 Choices |
|---|---|---|---|---|---|
| r=0 | 100% | 100% | 100% | 100% | 100% |
| r=0.05 | 100% | 100% | 100% | 100% | 100% |
| r=0.1 | 100% | 100% | 100% | 100% | 100% |
| r=0.15 | 100% | 100% | 100% | 100% | 100% |
| r=0.2 | 100% | 100% | 100% | 100% | 100% |
| r=0.25 | 99.7% | 100% | 99.5% | 99.0% | 80.0% |

Time Cost for Per Run

| | 5 Choices | 10 Choices | 50 Choices | 100 Choices | 10000 Choices |
|---|---|---|---|---|---|
| r=0 | 0.003930019s | 0.007775409s | 0.038110003s | 0.09424866s | 7.435941219s |
| r=0.05 | 0.003968138s | 0.00903297s | 0.042688169s | 0.094910076s | 8.461290359s |
| r=0.1 | 0.004464785s | 0.008744225s | 0.045663726s | 0.097951498s | 8.462076902s |
| r=0.15 | 0.004602315s | 0.008976836s | 0.049627956s | 0.095542486s | 8.945510149s |
| r=0.2 | 0.004800907s | 0.009758241s | 0.051130914s | 0.097419448s | 9.582294941s |
| r=0.25 | 0.005057936s | 0.010028378s | 0.04962585s | 0.099855525s | 9.969549894s |

# Thank You!

What if the last byte is random?

What if random=40%?

What if the length of key is 60?

---

What really helps me to attack?

The “extra” information is not important,

What really matters is

the key is repeatedly used → Build filters to win!

# X-Turbo-Fan Filter Array for different index

- A simple idea to make full use of "Repeated use of the key"
- Concatenate Filters
- TurboFan & Reversed TurboFan

```python
elif(sum(res)>1):
    tmp = copy.copy(res)
    for x in range(5):
        if(res[x]!=0):
            tmp[x]+=pwn(c[:-1],D1[x][:-1])# turbofan-1
            tmp[x]+=pwn(c[:-2],D1[x][:-1])# turbofan-1
            tmp[x]+=pwn(c[:-3],D1[x][:-1])# turbofan-1

            tmp[x]+=pwn(c[:-2],D1[x][:-2])# turbofan-2
            tmp[x]+=pwn(c[:-3],D1[x][:-2])# turbofan-2
            tmp[x]+=pwn(c[:-4],D1[x][:-2])# turbofan-2
            tmp[x]+=pwn(c[:-5],D1[x][:-2])# turbofan-2
            tmp[x]+=pwn(c[:-6],D1[x][:-2])# turbofan-2

            tmp[x]+=pwn(c[:-3],D1[x][:-3])# turbofan-3
            tmp[x]+=pwn(c[:-4],D1[x][:-3])# turbofan-3
            tmp[x]+=pwn(c[:-5],D1[x][:-3])# turbofan-3
            tmp[x]+=pwn(c[:-6],D1[x][:-3])# turbofan-3
            tmp[x]+=pwn(c[:-7],D1[x][:-3])# turbofan-3
            tmp[x]+=pwn(c[:-8],D1[x][:-3])# turbofan-3


            tmp[x]+=pwn(c[:-4],D1[x][:-4])# turbofan-4
            tmp[x]+=pwn(c[:-5],D1[x][:-4])# turbofan-4
            tmp[x]+=pwn(c[:-6],D1[x][:-4])# turbofan-4
            tmp[x]+=pwn(c[:-7],D1[x][:-4])# turbofan-4
            tmp[x]+=pwn(c[:-8],D1[x][:-4])# turbofan-4
            tmp[x]+=pwn(c[:-9],D1[x][:-4])# turbofan-4
            tmp[x]+=pwn(c[:-10],D1[x][:-4])# turbofan-
            tmp[x]+=pwn(c[:-11],D1[x][:-4])# turbofan-

            tmp[x]+=pwn(c[:-5],D1[x][:-5])# turbofan-5
            tmp[x]+=pwn(c[:-6],D1[x][:-5])# turbofan-5
            tmp[x]+=pwn(c[:-7],D1[x][:-5])# turbofan-5
            tmp[x]+=pwn(c[:-8],D1[x][:-5])# turbofan-5
            tmp[x]+=pwn(c[:-9],D1[x][:-5])# turbofan-5
            tmp[x]+=pwn(c[:-10],D1[x][:-5])# turbofan-
```

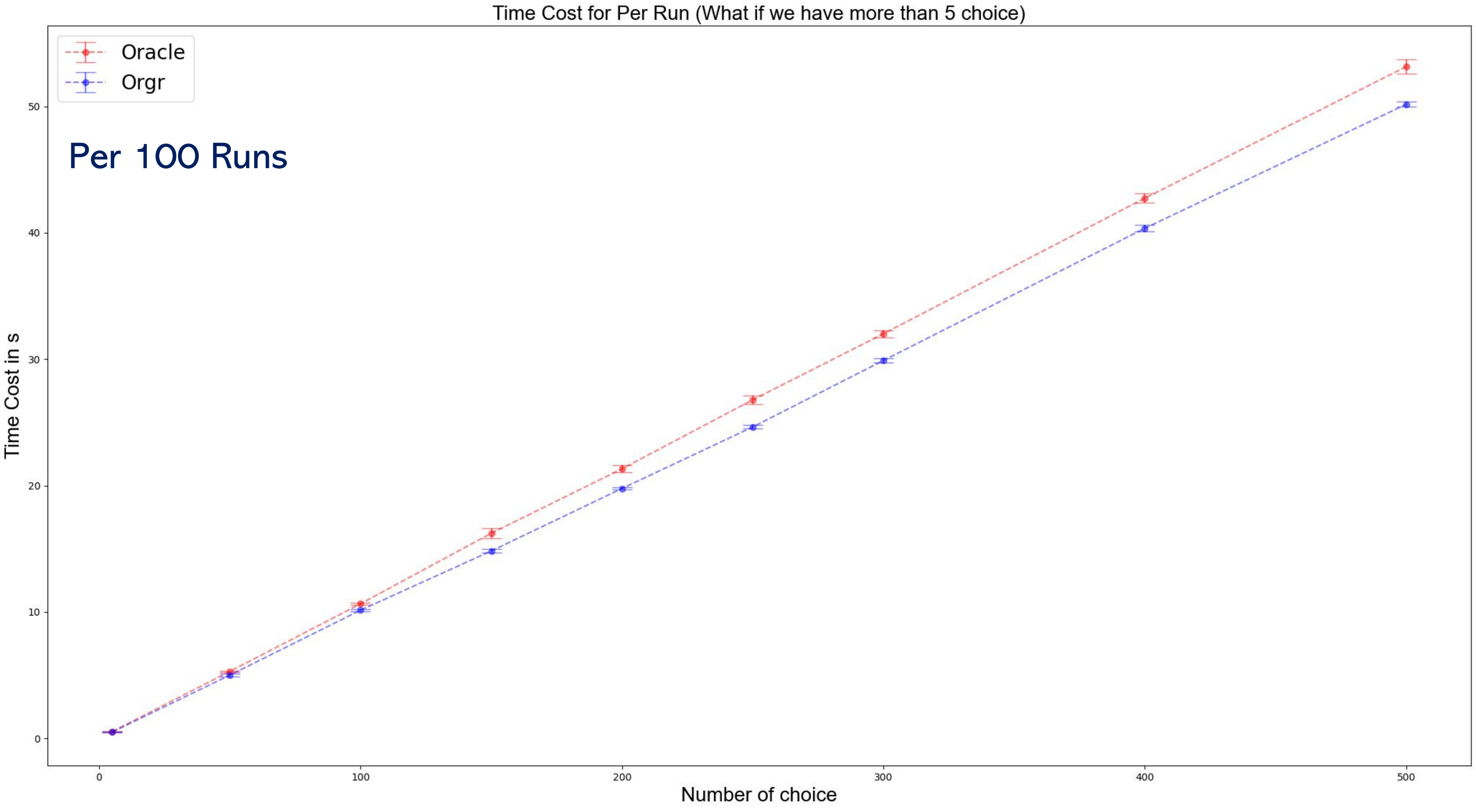
```python
            lol = c[::-1]
            hah = D1[x][::-1]
            tmp[x]+=pwn(lol,hah)# reversed-turbofan-1
            tmp[x]+=pwn(lol[:-1],hah)# reversed-turbofan-1
            tmp[x]+=pwn(lol[:-2],hah)# reversed-turbofan-1

            tmp[x]+=pwn(lol[:-1],hah[:-1])# reversed-turbo
            tmp[x]+=pwn(lol[:-2],hah[:-1])# reversed-turbo
            tmp[x]+=pwn(lol[:-3],hah[:-1])# reversed-turbo

            tmp[x]+=pwn(lol[:-2],hah[:-2])# reversed-turbo
            tmp[x]+=pwn(lol[:-3],hah[:-2])# reversed-turbo
            tmp[x]+=pwn(lol[:-4],hah[:-2])# reversed-turbo
            tmp[x]+=pwn(lol[:-5],hah[:-2])# reversed-turbo
            tmp[x]+=pwn(lol[:-6],hah[:-2])# reversed-turbo

            tmp[x]+=pwn(lol[:-3],hah[:-3])# reversed-turbo
            tmp[x]+=pwn(lol[:-4],hah[:-3])# reversed-turbo
            tmp[x]+=pwn(lol[:-5],hah[:-3])# reversed-turbo
            tmp[x]+=pwn(lol[:-6],hah[:-3])# reversed-turbo
            tmp[x]+=pwn(lol[:-7],hah[:-3])# reversed-turbo

            tmp[x]+=pwn(lol[:-4],hah[:-4])# reversed-turbo
            tmp[x]+=pwn(lol[:-5],hah[:-4])# reversed-turbo
            tmp[x]+=pwn(lol[:-6],hah[:-4])# reversed-turbo
            tmp[x]+=pwn(lol[:-7],hah[:-4])# reversed-turbo
            tmp[x]+=pwn(lol[:-8],hah[:-4])# reversed-turbo

            tmp[x]+=pwn(lol[:-5],hah[:-5])# reversed-turbo
            tmp[x]+=pwn(lol[:-6],hah[:-5])# reversed-turbo
            tmp[x]+=pwn(lol[:-7],hah[:-5])# reversed-turbo
            tmp[x]+=pwn(lol[:-8],hah[:-5])# reversed-turbo
            tmp[x]+=pwn(lol[:-9],hah[:-5])# reversed-turbo
            tmp[x]+=pwn(lol[:-10],hah[:-5])# reversed-turb
```

# Optimization (Randomness=40%)
# From Original Result(63.8%) to Final Result(97.5%)

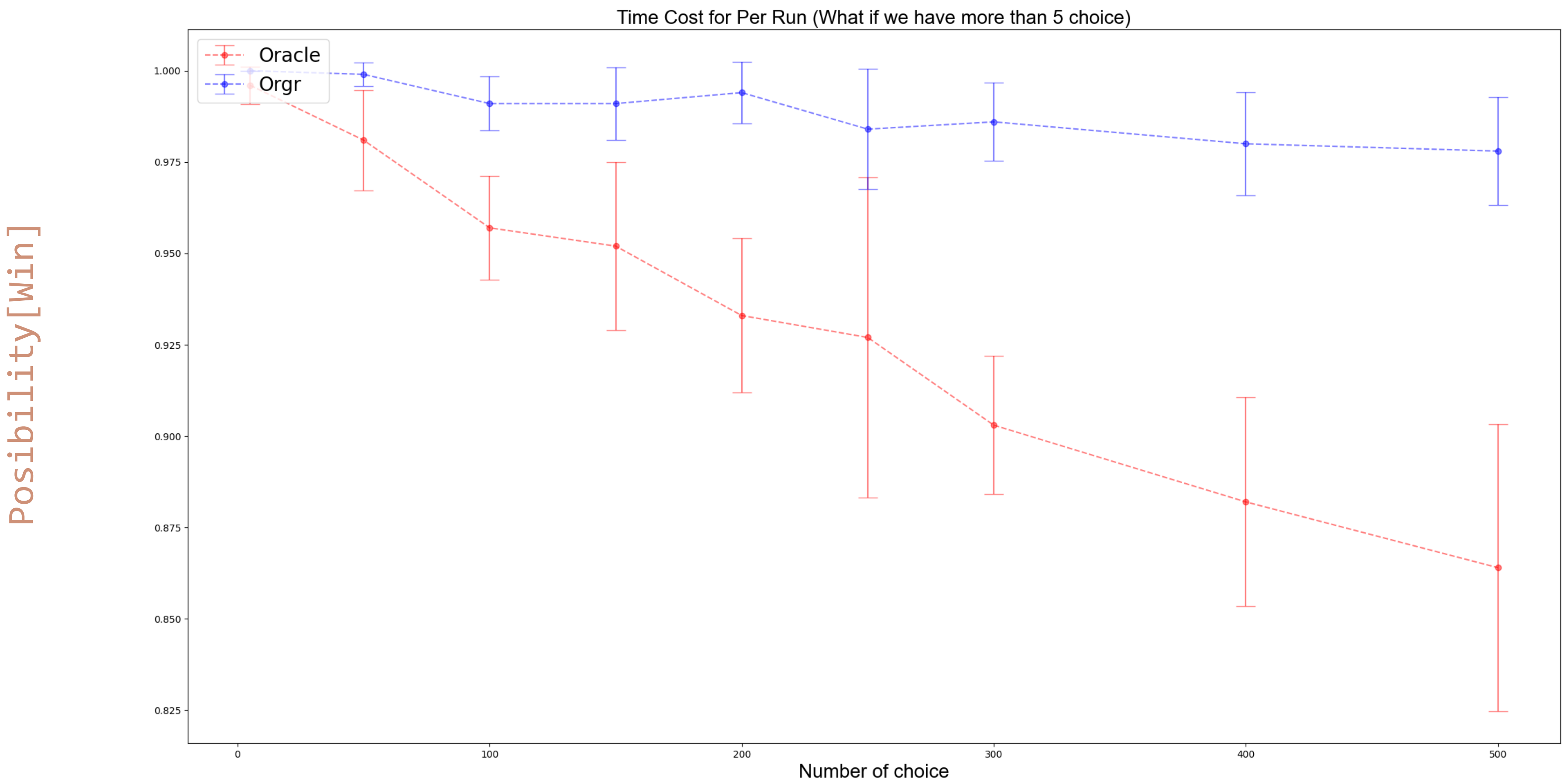
- Delete part of "Gap Check" Filter, Because it would wrongly eliminate some valid candidates ⇒ 12% success ⇒ 75.6%
- turbofan *2 without considering the randomness ⇒ + 15% success ⇒ %90.8
- Delete all the "Gap Check" in Filter, +1% success ⇒ 91.6%
- turbofan *2 with considering the randomness ⇒ +3.6% success ⇒ 95.2%
- turbofan *3 ⇒ +2% ⇒ 97.3%
- Multi-Level turbofan ⇒ +0.2% ⇒ 97.5% turbofan

# Oracle vs Orgr

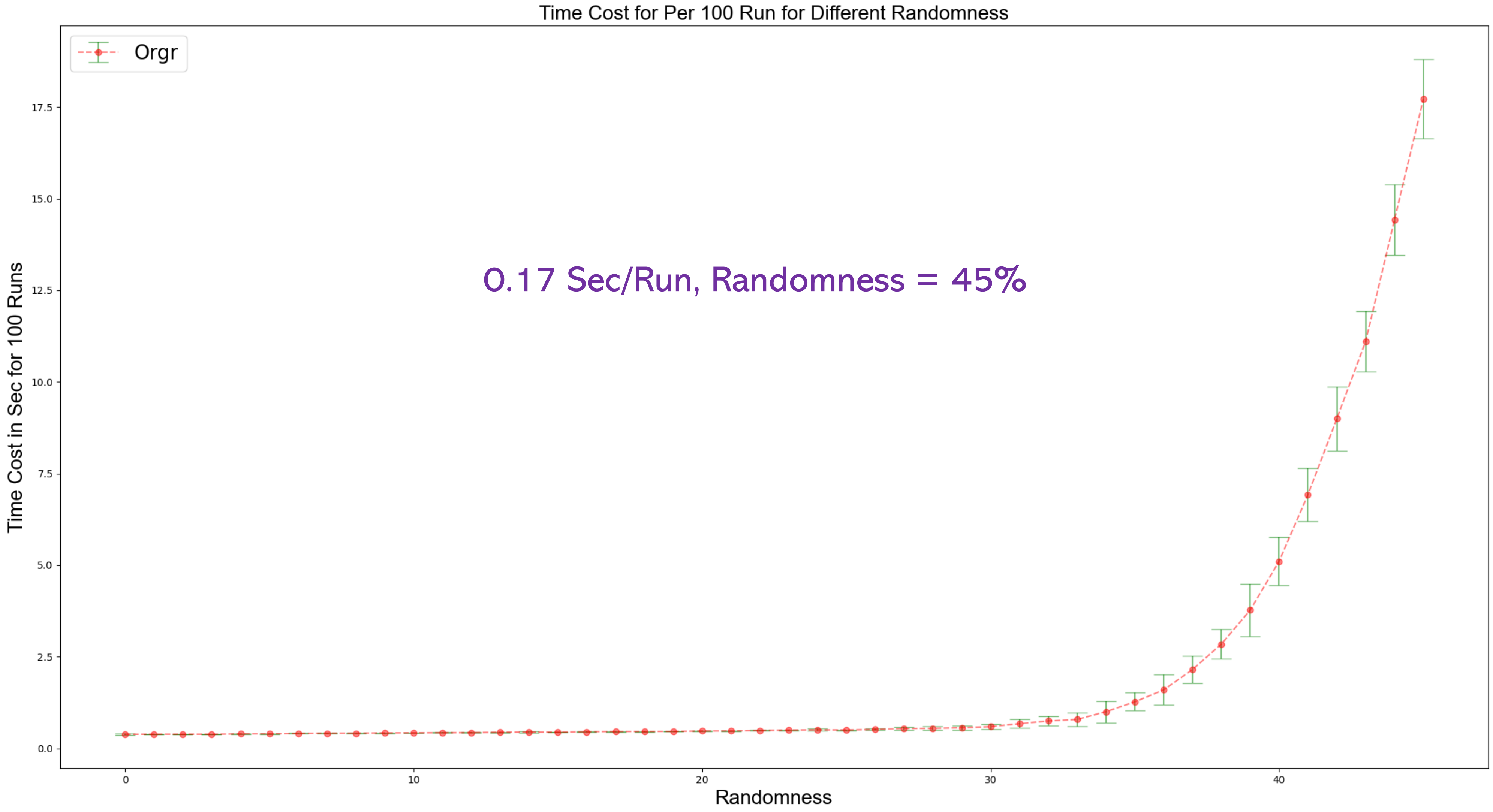
Tips:
The shape of Orgr is like a cute Tyrannosaurus



# Oracle vs Orgr

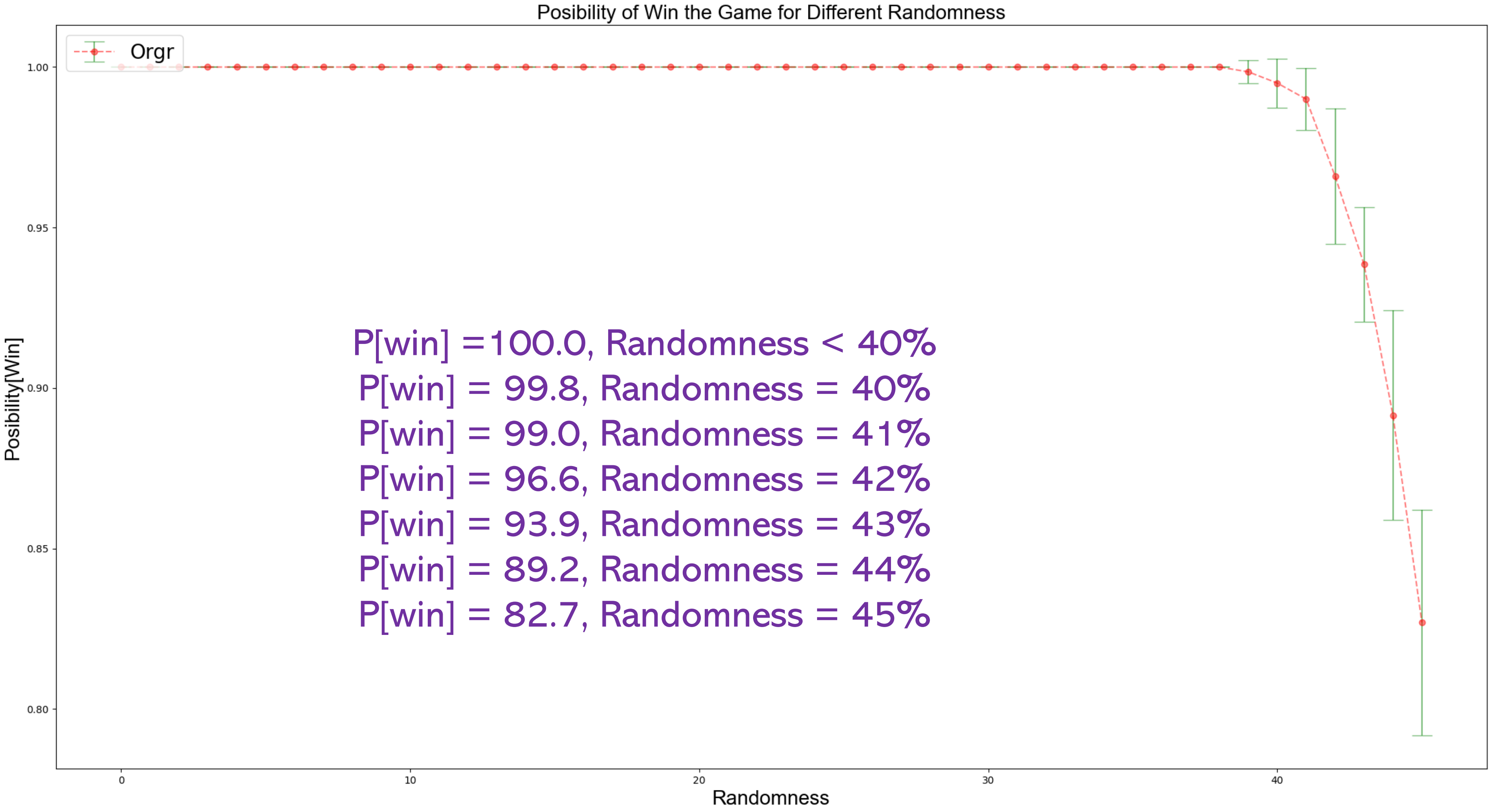


# Orgr



Time Cost for Per 100 Run for Different Randomness
Orgr
0.17 Sec/Run, Randomness = 45%
Time Cost in Sec for 100 Runs
Randomness
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
10
20
30
40

# Orgr




P[win] =100.0, Randomness < 40%
P[win] = 99.8, Randomness = 40%
P[win] = 99.0, Randomness = 41%
P[win] = 96.6, Randomness = 42%
P[win] = 93.9, Randomness = 43%
P[win] = 89.2, Randomness = 44%
P[win] = 82.7, Randomness = 45%

# How did I solve it?

- Repeated use of the key
  - Catch the nature of the problem
- Read & Try
  - We don't need to get the password/we have plaintext already
  - I tried 4 ways to solve the problem, what you can see is the best two
- Sit down & Think
  - Cybersecurity & Cryptography is a kind of magic
  - Vulnerability Discovery & Exploit is a kind of art

xm2146@nyu.edu