Timing attack using Genetic Algorithm

Jayveersinh Raj
3rd year student,
BS. Computer Science and Engineering (Majors, Data Science),
Innopolis University, Innopolis, Russia
j.raj@innopolis.university

***Abstract*—This paper is a report of timing attack performed on a server that relies on string matching to check login information. The timing attack was performed using Genetic Algorithm.**

***Index Terms*—Timing attack, password cracking, Genetic Algorithm**

I. Introduction

This paper is a report on a timing attack carried out on a server that uses string matching to validate login information. The timing attack was carried out using the Genetic Algorithm. The search utilizes the idea of timing attack as computation time information may leak due to lookup in the dictionary for password matching, and comparing.

A. Timing attacks

A timing attack is a side-channel attack in cryptography in which the attacker attempts to compromise a cryptosystem by analyzing the time it takes to execute cryptographic algorithms. Every logical operation in a computer takes time to execute, and the time can vary depending on the input; an attacker can work backwards to the input with precise measurements of the time for each operation. Finding secrets using timing information may be much easier than using cryptanalysis of known plaintext/ciphertext pairs. To increase the rate of information leakage, timing information is sometimes combined with cryptanalysis.

B. Genetic Algorithms

The ability of generic algorithms (GAs) to add direction to what appears to be random search is their main feature. This could result in a powerful and efficient search technique[1]. These algorithms use simulation to solve complex mathematical problems certain extent, the way in which biological genetic processes operate, see Liepins and Hilliard[2,3], they use an evolutionary theory concept to ”breed” progressively better solutions to problems with a large solution space.

A genetic algorithm begins with a randomly selected population of function inputs represented by strings (chromosomes), which are in the form of guessed key strings. In more optimal way for this case, a better guessed population can be formed from a finite set of possibilities to converge faster by calculating the time of each correct position. However, it is important to note that these timings are affected by machine, and processes running on it, hence, a dedicated machine is required for it.

GA uses the current string population to generate a new population in which the string on average, the new population produces better fitting results than the current population. This is accomplished by evaluating the fitness of each attempt [1]. This fitness function in our case is the measured time for each character. Since the server just uses string matching, this information leak is inevitable. The more successful passwords would then be used to create new successive generations of fitter passwords. This process is repeated until an optimal key emerges, that produces substantial decryption. GA follows three processes cycle for transition from one population to the next one in a similar way to that found in living creatures. This cycle consists of selection, crossover, and mutation, as shown in Figure 1

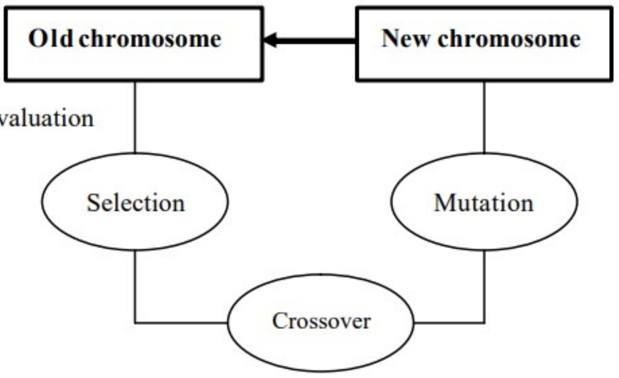


Figure 1. The basic generic algorithm cycle.

II. Related Work

In [1] Ali and Al-Salami presents a method for cryptanalysis of the RSA cryptosystem based on the use of a genetic algorithm. The search employs the timing attack concept, as computation time information may leak due to various

Identify applicable funding agency here. If none, delete this.

modular operations performed throughout the RSA encoding. This method suggests accelerating the process in order to reduce the number of plaintext-ciphertext samples required for a successful timing attack. The timing attack concept proposed in this work, along with its preliminary implementation, has yielded promising results on RSA cryptosystem samples. Further research into the application of the genetic algorithm technique to a practical RSA system has yielded promising results.

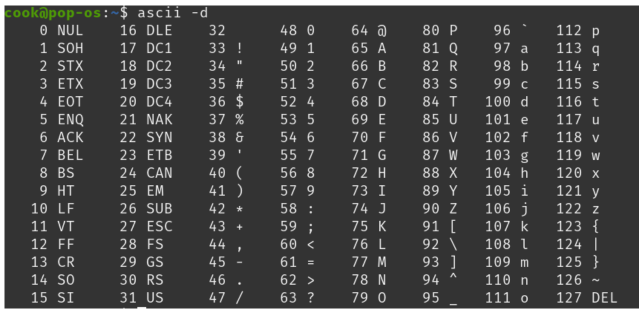
III. Methodology

A basic comparison based methodology was used to infer the complexity of such task as brute force, and a genetic approach to this. However, even in genetic approach different fitting functions, and different initial population creation techniques were implemented. Firstly, length was approximated for the password by using timeit library of python. After getting the length of the password, it was passed to the generating function to generate initial population, then that population was passed to the main loop function that repeatedly executes the genetic algorithm loop. For timing calculations, timeit library was used which repeatedly calculated execution time for 1000 times to find minimum best time taken. Some of the ones implemented for comparison purposes:

- Fitness functions:
  - Score based on timing when entire string is passed
  - Score based on combined time, passing single characters

- Initial population:
  - Random strings/characters in given range of ASCII values
  - Brute forced finite probable string ranging from given ASCII ranged values

- ASCII ranges:
  - 32-125
  - 65-125
  - 97-125



```
cook@pop-os:~$ ascii -d
    0 NUL    16 DLE    32        48 0     64 @     80 P     96 `    112 p
    1 SOH    17 DC1    33 !      49 1     65 A     81 Q     97 a    113 q
    2 STX    18 DC2    34 "      50 2     66 B     82 R     98 b    114 r
    3 ETX    19 DC3    35 #      51 3     67 C     83 S     99 c    115 s
    4 EOT    20 DC4    36 $      52 4     68 D     84 T    100 d    116 t
    5 ENQ    21 NAK    37 %      53 5     69 E     85 U    101 e    117 u
    6 ACK    22 SYN    38 &      54 6     70 F     86 V    102 f    118 v
    7 BEL    23 ETB    39 '      55 7     71 G     87 W    103 g    119 w
    8 BS     24 CAN    40 (      56 8     72 H     88 X    104 h    120 x
    9 HT     25 EM     41 )      57 9     73 I     89 Y    105 i    121 y
   10 LF     26 SUB    42 *      58 :     74 J     90 Z    106 j    122 z
   11 VT     27 ESC    43 +      59 ;     75 K     91 [    107 k    123 {
   12 FF     28 FS     44 ,      60 <     76 L     92 \    108 l    124 |
   13 CR     29 GS     45 -      61 =     77 M     93 ]    109 m    125 }
   14 SO     30 RS     46 .      62 >     78 N     94 ^    110 n    126 ~
   15 SI     31 US     47 /      63 ?     79 O     95 _    111 o    127 DEL
```

IV. GitHub link

The complete code with a readme file has been uploaded to the github with complete instructions on how to run it. Follow the link below to see it :
Clickable : GitHub.

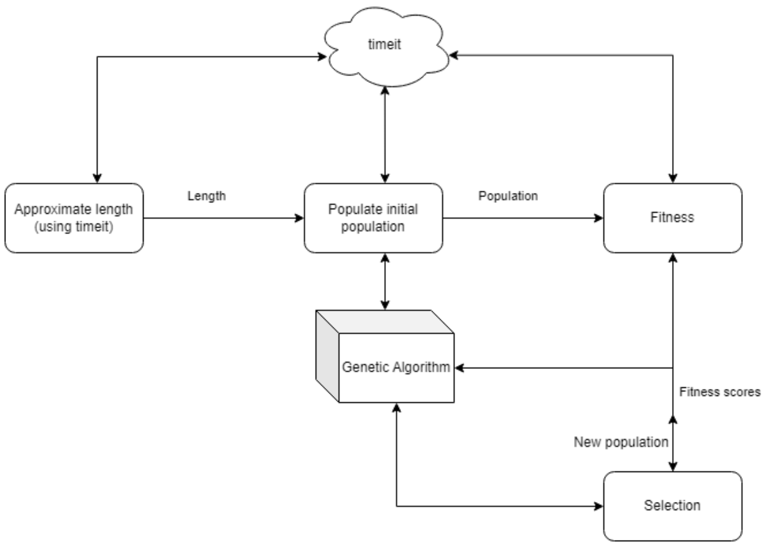
V. Experiments and Evaluation

A. Experiments

As described in the Methodology section, those were the different configurations for experiments. Along with them hyper-parameters were also tuned. Hyper-parameters were:

- Population size
- No of generations
- Mutation rate
- Ascii character range

The main loop until stopping criteria (number of generations) :



B. Evaluation

Evaluation metric is straight forward, first calculate the fitness score which is time based, and then we calculate accuracy which has this simple formula: $\frac{1.0}{1+fitnessscore}$

However, for the overall evaluation it is quite clear that we consider how many characters were correctly predicted on the right relative position.

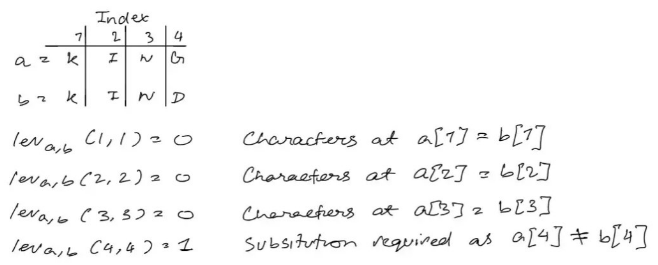
VI. Analysis and Observations

A. Analysis

The analysis on the different variants was using the simple metric of number of correct characters on correct relative position was simple to follow. However some complex analysis mechanisms for informational purpose like Levenshtein-distance were run.
General formula :

$$\operatorname{lev}(a,b)=\begin{cases}|a| & \text{if } |b|=0,\\ |b| & \text{if } |a|=0,\\ \operatorname{lev}\big(\operatorname{tail}(a),\operatorname{tail}(b)\big) & \text{if } a[0]=b[0],\\ 1+\min\begin{cases}\operatorname{lev}\big(\operatorname{tail}(a),b\big)\\ \operatorname{lev}\big(a,\operatorname{tail}(b)\big)\\ \operatorname{lev}\big(\operatorname{tail}(a),\operatorname{tail}(b)\big)\end{cases} & \text{otherwise,}\end{cases}$$

Provided by [4]:



$$\begin{array}{c|c|c|c|c} \multicolumn{5}{c}{\text{Index}}\\ & 1 & 2 & 3 & 4\\ \hline a= & K & I & N & G\\ b= & K & I & N & D \end{array}$$

$\mathrm{lev}_{a,b}(1,1)=0$ Characters at $a[1]=b[1]$

$\mathrm{lev}_{a,b}(2,2)=0$ Characters at $a[2]=b[2]$

$\mathrm{lev}_{a,b}(3,3)=0$ Characters at $a[3]=b[3]$

$\mathrm{lev}_{a,b}(4,4)=1$ Subsitution required as $a[4]\neq b[4]$

Example of Levenshtein distance on the strings King and Kind. Image provided by author.

B. Observation

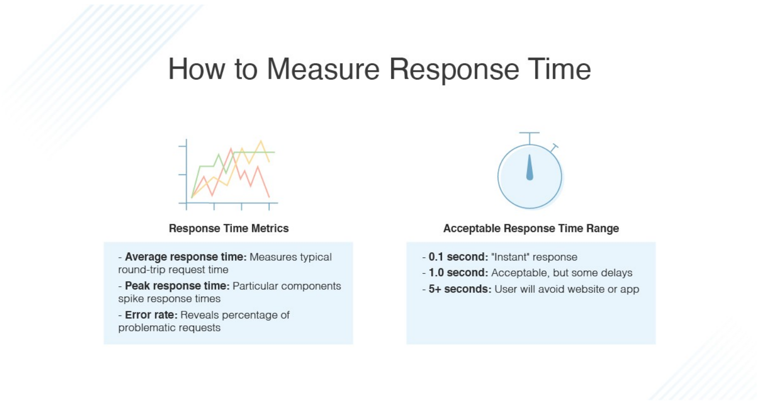
It is observed that only a small length of password, and with constrained ASCII characters can be implemented on a single machine due to the computational limitation, that too with not reproducible, and correct guess till longer generations. The reason is that timing calculations are very sensitive, and are subject to machine, and number of processes running. But since it is just an analogy to a real server, the server and response time should theoretically be machine independent (not considering internet bandwidth).

VII. Conclusion

In conclusion, using the computation time it is possible to crack a password, similar to what [1] did. It is very important to note that such task is computational heavy, and hence distributed dedicated machines are advised if the search space is high. In my project however, I provide the possibility to change these ASCII character ranges. Fruitful results would not be achieved given the complexity, combinations, and search space if huge range is used on single machine. In this case we are using smaller search space to prove the concept but with higher computational power (mainly distributed) higher search space can be covered. Moreover, patience is required to let the machine complete generations, also, it might not be the entirely same password guessed, but it would be somewhat similar since genetic algorithm promises good solutions, not the best. However, it does not mean that the exact password is not achievable.

In real life server the fitting function would depend on the response time that could be calculated in the following way:

Provided by [5]:



How to Measure Response Time

Response Time Metrics

- **Average response time:** Measures typical round-trip request time
- **Peak response time:** Particular components spike response times
- **Error rate:** Reveals percentage of problematic requests

Acceptable Response Time Range

- **0.1 second:** "Instant" response
- **1.0 second:** Acceptable, but some delays
- **5+ seconds:** User will avoid website or app

Moreover, it is important to note that this only works if the server implements string matching. There are many ways by which timing attacks as such can be prevented, which would be discussed later in the subsection. But one should remember that even if the server has this vulnerably, hacking is a crime by law, and also immoral. If one suspects such server, he/she should not exploit it to do harm or steal information. The goal of this project is not to promote such thing, but is to create an awareness of a vulnerability and an exploit that should be considered while development.

A. Preventing timing attack

So, now that we've successfully hacked ourselves, how can we protect ourselves? Normally, it can be difficult to protect a system from timing attacks because both the compiler and the runtime may use end-fast instructions. It could also be caused by arithmetic operations in the processor and other factors. A first move could be to heavy the check password function to make it finish in constant-instruction count, however it should be noted that this is a simple, but costly solution in terms of time. [6] suggests the following counter measures:

- **Constant-time techniques:**
  A common approach to cryptographic code protection is to ensure that its behavior is never data dependent: that the sequence of cache accesses or branches, for example, is independent of either the key or the plaintext.
- **Injecting noise :**
  In theory, it should be possible to prevent timing channel exploitation without eliminating contention by ensuring that the attacker's measurements contain so much noise that they are essentially useless.
- **Virtual time:**
  The virtual time approach seeks to eliminate all access to real time by supplying only virtual clocks whose progress is completely deterministic and independent of the actions of vulnerable components.
- **Black-box mitigation :**
  Instead of attempting to synchronize all observable clocks with the execution of individual processes, this approach aims to achieve system-wide determinism by controlling the timing of externally visible events. As a result, it applies only to remotely exploitable attacks that rely on variations in system response time. This avoids much of the performance cost of virtual time, as well as the difficult issue of locating all possible clocks within a system.
- **Partitioning time :**
  Attacks that rely on concurrent or consecutive access to shared hardware can be mitigated by providing timesliced exclusive access (in the first case) or carefully managing the transition between timeslices (in the second case).

For more refer [6].

References

[1] H. Ali and M. Al-Salami "Timing Attack Prospect for RSA Cryptanalysts Using Genetic Algorithm Technique", The International Arab Journal of Information Technology, vol. 1, no. 1, Jan 2004

[2] Kolodziejczyk J., "The Application of Genetic Algorithm in Cryptoanalysis of Knapsack Cipher," http:// www.cryptography. com/ dpa.../, 1998.

[3] Liepins G. E. and Hillard M. R., "Genetic Algorithms: Foundations and Applications," Annals of Operations Research, vol. 21, pp. 31- 58, 1989.

[4] Vatsal, "Text Similarity w/ Levenshtein Distance in Python", towardsdatascience.com. https://towardsdatascience.com/text-similarity-w-levenshtein-distance-in-python-2f7478986e75 (accessed Dec. 5, 2022)

[5] Staff Contributor , "How to Check, Measure, and Improve Server and Application Response Time With Monitoring Tools", dnsstuff.com. https://www.dnsstuff.com/response-time-monitoring (accessed Dec. 5, 2022)

[6] Ge, Q. et al. (2016) A survey of microarchitectural timing attacks and countermeasures on contemporary hardware - Journal of Cryptographic Engineering, SpringerLink. Springer Berlin Heidelberg. Available at: https://link.springer.com/article/10.1007/s13389-016-0141-6 (Accessed: December 6, 2022).