Using Genetic Algorithms to Optimize Hashcat Masks

Rafael Dias Campos
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

Abstract

Password cracking tools like Hashcat play a critical role in Information Security Red Team operations by enabling the recovery of passwords from hashed data. Among the attack modes available in Hashcat, mask attacks offer a customizable approach to generating candidate passwords. However, creating efficient masks that maximize recovery rates while minimizing computational cost remains challenging. This paper explores the application of genetic algorithms to optimize Hashcat masks for offline hash cracking. By evaluating commonly used mask sets, such as Corporate Masks and Extreme Breach Masks, we identify limitations in their efficiency. We then apply a genetic algorithm to evolve optimized mask sets based on password recovery rates and computational cost. Experimental results demonstrate that the genetic algorithm outperforms traditional mask sets in some scenarios, leading to more effective password recovery strategies.

CCS Concepts

• **Security and privacy → Cryptanalysis and other attacks**; • **Computing methodologies → Genetic algorithms**.

Keywords

Hashcat, Cybersecurity, Offline hash cracking, Genetic Algorithm

ACM Reference Format:
Rafael Dias Campos. 2024. Using Genetic Algorithms to Optimize Hashcat Masks. In *Proceedings of Computação Natural*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

During Information Security Red Team operations, it's common to encounter password hashes when performing offensive tasks. For example, during post-exploitation activities in an Active Directory environment, it's possible to extract NTLM hashes from the domain controller. In order to recover the original password from these hashes, it's necessary to use password cracking tools. One of the most popular tools for this task is Hashcat [1], that supports a wide range of hash types and attack modes and make efficient use of the graphics card (GPU) to improve the speed of the attacks.

Hashcat works by receiving a list of target hashes and a list of candidate passwords. For each candidate password, Hashcat applies a hash function to it and compares the result with the target hashes. If the result matches any of the target hashes, the password is considered cracked. Otherwise, the process continues with the next candidate password. The different attack modes in Hashcat differ in the way the candidate passwords are generated, and can be summarized as follows:

- **Dictionary attack**: Hashcat uses a wordlist of candidate passwords, called a dictionary, to try to recover the passwords. These attacks can be further extended by using rules to modify the candidate passwords.
- **Mask attack**: Hashcat generates candidate passwords based on a mask or list of masks. These masks define the structure of the candidate passwords, with each character in the mask representing a character in the candidate password.
- **Hybrid attack**: Hashcat combines the dictionary and mask attacks, generating candidate passwords by applying masks either before or afther the words in the dictionary.

The most difficult part of the password cracking process is generating effective candidate passwords. On one hand, it's desired to generate as many candidate passwords as possible, since that increases the chances one of them will match the target hash. On the other, hand generating too many candidate passwords and calculating their respective hashes can be computationally expensive, slowing down the cracking process. For this reason, the main goal of password cracking is to generate the most effective candidate passwords, that is, the ones that are more likely to match the target hashes.

When working with the dictionary attack mode, it's common to use wordlists available on the internet. These wordlists are usually created from real passwords leaked in data breaches, and therefore should contain good candidate passwords. Alternatively, it's possible to create custom wordlists by combining existing wordlists or using tools like CUPP [4] to generate them. Lastly, it's possible to use rules to modify the candidate passwords, for example by adding numbers or special characters to them.

However, finding effective masks can be a more challenging process. There are two large commonly available masks on the internet: the Corporate Masks [3] and the Extreme Breach Masks [6]. These masks are based on patterns observed in real-world passwords, but they might not be well optimized for recovering the most passwords in the shortest time. This paper proposes evaluating these using genetic algorithms to optimize Hashcat masks, improving the efficiency of password cracking. The GitHub repository for this project can be found at https://github.com/RafaelDiasCampos/Genetic-Algorithm.

2 Hashcat Masks

Hashcat masks are strings that define the structure of the candidate passwords. Each character in the mask represents a character in the candidate password, and can either represent a fixed character or a character set. By default, Hashcat supports the following character sets:

Computação Natural, Belo Horizonte, MG, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- **?d**: Digits from 0 to 9.
- **?l**: Lowercase letters from a to z.
- **?u**: Uppercase letters from A to Z.
- **?s**: Special characters.
- **?a**: All printable ASCII characters.
- **?b**: All ASCII characters.

For example, the mask **?l?l?l?l?l?l?l** represents a candidate password with 6 lowercase letters. The mask **?d?d?d?d?d?d?d** represents a candidate password with 6 digits. The mask **?l?d?l?d?l?d** represents a candidate password with 3 pairs of lowercase letters and digits.

For this paper, we will consider masks that contain only the character sets **?d**, **?l**, **?u**, **?s**, and **?a**. We'll also seaparate the masks based on their number of characters. This way, each list of masks can be used to recover passwords of a specific length.

In this paper, we'll only consider masks that generate candidate passwords with a length between 8 and 15 characters. These lengths were chosen because passwords shorter than 8 characters can often be cracked by doing an exhaustive search of every password candidate. On the other hand, passwords longer than 15 characters are less common and take longer to crack, so they are not likely to be recovered in a reasonable time.

3 Measuring Effectiveness of Masks

Initially, it's necessary to decide how to evaluate the effectiveness of a list of masks. The most obvious metric for measuring a mask effectiveness is the number of candidate passwords generated by the mask that match the target hashes. However, this metric alone is not enough, since the goal is to generate the most candidate passwords in the shortest time possible. Therefore, it's necessary to consider the time taken to generate the candidate passwords, which is proportional to the number of candidate passwords generated by that mask.

Additionally, it's important to consider that, in most real world scenarios, there's a limited for performing the brute force attacks. However, at the same time, it's not an issue if a mask takes longer to execute than the available time, since it can be executed only partially. At the same time, there should be a penalty for masks that exceed the maximum cost, to prioritize masks that fit the desired cost. With that in mind, we decided to evaluate the masks based on the following metrics:

- **Number of recovered password (matches)**: The number of candidate passwords generated by the mask that match the target hashes.
- **Cost**: The number of candidate passwords generated by the mask.

And on the following parameters:

- **Maximum cost**: The maximum desired cost of the mask, that indicates the available time to run the attack.
- **Fitness penalty**: The percentage penalty applied to the fitness of the mask that exceeds the maximum cost.

Combining this, we arrived at the following fitness function for evaluating the masks:

$$\text{BaseFitness} = \text{Matches}/\text{Wordlist size} \tag{1}$$

$$\text{Fitness} = \begin{cases} \text{BaseFitness}, & \text{if Cost} \leq \text{MaxCost} \\ \text{BaseFitness} \times (1 - \text{FitnessPenalty}), & \text{otherwise} \end{cases} \tag{2}$$

3.1 Calculating the number of matches

One of the metrics used for calculating the fitness of a mask is the percentage of recovered passwords. This step consists of using a wordlist to represent the passwords and measuring how many of them are generated by the mask. In this paper, we used the Weakpass 4 wordlist [5], that can be freely obtained on the internet. This list contains over 2 billion passwords obtained in data breaches, so it should be a good source of real passwords.

After obtaining the wordlist, we separated it into smaller wordlists, based on the password length. With these smaller wordlists, it's now possible to load one of them into memory and iterate over the passwords and check if they are generated by the mask. However, the large number of passwords in the wordlist makes this process computationally expensive. Therefore, it's necessary to use a more efficient method to calculate the number of matches.

For this method, we first sorted each wordlist according to the following order:

- **Lowercase letters**
- **Uppercase letters**
- **Digits**
- **Special characters**

With the sorted wordlists, it was possible to create a tree-like structure. This was done by recursively performing a binary search on each wordlist, and storing the amount of passwords that match a certain mask. The structure can be observed in figure AAA

The created structure allows for quickly calculating how many passwords a mask would crack, by traversing it and returning the value of the last node. However, special care needs to be taken when dealing with a list of masks. Multiple masks can generate the same password, so it's necessary to avoid counting them repeatedly. To solve this issue, we used a set to store the last nodes reached by the masks, and later sum the values of these nodes to calculate the total number of matches.

With this method, it was possible to calculate the number of matches for a mask in a fraction of the time it would take to iterate over the wordlist.

3.2 Calculating the cost

The other metric used for calculating the fitness of a mask is the cost, that is, the number of candidate passwords generated by the mask. Before calculating the cost of a mask, it's useful to calculate the cost of a character set, according to the number of characters in that set. The table cost of each character set can be seen in table 1.

Since the masks generate a candidate password by combining the character sets, it's possible to calculate their cost by multiplying the cost of the character sets that compose it.

3.3 Deciding the parameters

After designing a method for evaluating the fitness of a list of masks, it's necessary to decide the parameters used in the fitness function. For this paper, we decided to use the following parameters:

Table 1: Cost of character sets

Character set	Cost	Comments
?d	10	Digits from 0 to 9
?l	26	Lowercase letters from a to z
?u	26	Uppercase letters from A to Z
?s	33	Special characters
?a	95	All printable ASCII characters

- **Maximum cost**: 5×10^{15}
- **Fitness penalty**: 0.05

The maximum cost was chosen based on the runtime of about 10 hours in a computer with an Nvidia RTX 3070 Ti, measured at 76146.5 MH/s with NTLM hashes. The fitness penanlty was chosen to only slightly penalize masks that exceed the maximum cost, since it's not a significant issue if a mask takes longer to execute than the available time.

4 Calculating the fitness of the current methods

After designing a method for evaluating the fitness of a list of masks, it's necessary to calculate the fitness of the current used methods. For this paper, we used the Corporate Masks [3] and the Extreme Breach Masks [6], as they are popular and freely available on the internet. Both of these lists contains masks for passwords of different lengths, and we calculated the fitness of each mask in each list.

The Extreme Breach Masks contains different lists of masks, depending on their total runtime. We used the largest one that isn't compressed (14 1-year) because it includes all the masks in the previous lists. Furhtermore, we considered that the masks were sorted by their other of priority, since hashcat runs them in order.

For each password length decided in section 2, we started calculating the fitness of the first mask in the list, using the parameters defined in section 3.3. We then added the second mask, and calculated the fitness of the combined list. This process was repeated until the fitness dropped from one iteration to the next, indicating that the cost of the masks was larger than the desired maximum cost. At this time, we returned the maximum fitness obtained, the masks that generated it, and their cost.

The results of calculations for the Corporate Masks can be seen in table 2. One additional observation is that there are not masks with 15 characters in the Corporate Masks, so we didn't calculate the fitness for this length. The results of calculations for the Extreme Breach Masks can be seen in table 3.

Analyzing these results, it is interesting to note that the Extreme Breach Masks have a much higher fitness for passwords of 9, 10, 11, 12 and 13 characters. This indicates that the Extreme Breach Masks are more effective than the Corporate Masks for these lengths. For passwords of lengths 8 and 14, both masks have similar fitness, indicating that they are both recover similar amounts of passwords. However, for length 8, the Corporate Masks have a much lower cost, while for length 14, the Extreme Breach Masks have a lower cost.

Table 2: Fitness of the Corporate Masks

Length	Fitness	Cost
8	0.972079	1605171098613760
9	0.562294	4999603084300800
10	0.386915	4998028389760000
11	0.208118	4989306592000000
12	0.085927	4986552000000000
13	0.032640	4863360000000000
14	0.030719	4940000000000000

Table 3: Fitness of the Extreme Breach Masks

Length	Fitness	Cost
8	0.999705	4999855044508416
9	0.955987	4997928685184000
10	0.846787	4988401456221184
11	0.444393	2834403868825600
12	0.175589	4411987691520000
13	0.065720	4394348800000000
14	0.034780	1857600000000000
15	0.187582	3600000000000000

5 Genetic Algorithm

After calculating the fitness of the current methods, it's necessary to design a method for optimizing the masks. For this paper, we decided to use a genetic algorithm, since it's a popular method for optimizing problems with a large search space. The genetic algorithm works by creating a population of candidate solutions, and then evolving them over multiple generations. A simple general-purpose genetic algorithm was implemented using Python, that receives the following parameters:

- **Population size**: The number of candidate solutions in each generation.
- **Generations**: The number of generations to evolve the candidates.
- **Crossover rate**: The probability of a candidate solution being crossed over.
- **Mutation rate**: The probability of a candidate solution being mutated.
- **Elitism**: The number of candidate solutions that are kept from one generation to the next.
- **Tournament size**: The number of candidate solutions that are selected to be part of the tournament selection step.
- **Initial population**: The initial candidate solutions.
- **Fitness function**: The function used to evaluate the fitness of a candidate solution.
- **Crossover function**: The function used to cross over two candidate solutions.
- **Mutation function**: The function used to mutate a candidate solution.

In order to use this algorithm, it is necessary to determine how an individual will be represented, implement the crossover and

mutation genetic operators and the fitness function. Lastly, it is necessary to define the parameters of the genetic algorithm, such as the population size, the number of generations, the crossover rate, the mutation rate, the elitism, and the tournament size.

The following sections will detail how these steps were performed.

5.1 Representing the Individual

An individual can be represented as the list of masks it contains. For this we first created a MaskCharset enum that contains the character sets used in the masks. Each charset contains their representation by hashcat, and their cost, as defined in table 1.

Following this, we created a Mask class that contains a list of MaskCharset, representing the mask. This class contains a cost attribute, calculated by multiplying the cost of each MaskCharset in the list, and a method to generate the mask string, that is used by hashcat.

Finally, we created a MaskIndividual class that contains a list of Mask, representing the individual, and a paramter that defines the maximum number of masks that can be contained in the individual. This class also contains a cost attribute, calculated by summing the cost of each Mask in the list.

5.2 Genetic Operators

The genetic operators used in the genetic algorithm are the crossover and mutation operators. The crossover operator works by selecting two individuals from the population, and combining them to create a new individual. The mutation operator works by selecting an individual from the population, and modifying it to create a new individual.

For the crossover operator, we generate the offspring by combining some masks from each parent. We first decide on the amount of masks each children will have, by generating a random number betwewen 1 and the maximum number of masks. This number is generated following a normal distribution centered around the average number of masks in each parent. We then select the masks from each parent, and combine them to create the children.

For the mutation operator, there are three possible mutations, that are randomly selected:

- **Add mask**: Adds a random mask to the individual.
- **Remove mask**: Removes a random mask from the individual.
- **Mutate mask**: Mutates a random mask in the individual by changing one of its character sets.

5.3 Fitness Function

The fitness function used in the genetic algorithm is the same as the one used to calculate the fitness of the current methods. This function receives an individual and calculates its fitness based on the number of matches and the cost of the individual, according to equation 1. The function also receives the maximum cost and the fitness penalty as parameters, with the values defined in section 3.3.

5.4 Deciding the Parameters

After designing the genetic algorithm, it's necessary to decide the parameters used in the algorithm. For this paper, we used the Python library Optuna [2] to perform hyperparameter optimization.

Table 4: Search space for the parameters

Parameter	Search space
Population size	10 to 300
Generations	10 to 300
Crossover rate	0.5 to 1.0
Mutation rate	0.01 to 0.5
Elitism	1 to 5
Tournament size	3 to 5
Maximum masks	5 to 30

Table 5: Optimized parameters

Parameter	Value
Population size	265
Generations	300
Crossover rate	0.675
Mutation rate	0.235
Elitism	5
Tournament size	5
Maximum masks	30

Table 6: Fitness of the Genetic Algorithm

Length	Fitness	Cost
8	0.998075	4998945864072960
9	0.877505	4743958039584768
10	0.719535	4732924205039616
11	0.402315	4917264338585600
12	0.206068	4858597775360000
13	0.064911	4917369600000000
14	0.043578	4991200000000000
15	0.187582	4600000000000000

This library works by performing multiple trials, each with a different set of parameters, and selecting the one that maximizes the fitness of the individuals. We first chose the search space for each parameter when considering masks of length 10, and then ran the optimization process for 500 trials. The search space for each parameter can be seen in table 4. The values chosen based on the optimization process can be seen in table 5.

5.5 Executing the Genetic Algorithm

After deciding the parameters of the genetic algorithm, the next step is to execute it. For this paper, we used the optimized parameters defined in table 5, and ran the genetic algorithm 10 times for each password length, using fixed seeds. The results of the genetic algorithm can be seen in table 6.

The progression of the fitness of the genetic algorithm can be seen in figures 1, 2, 3, 4, 5, 6, 7, and 8. These figures show the fitness of the best individual in each generation, in each of the 10 executions of the algorithm for that password length.

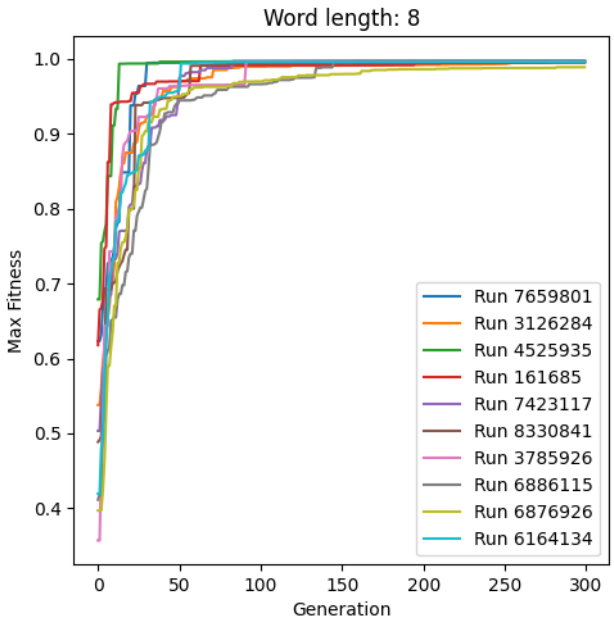

Figure 1: Fitness of the Genetic Algorithm for passwords of length 8

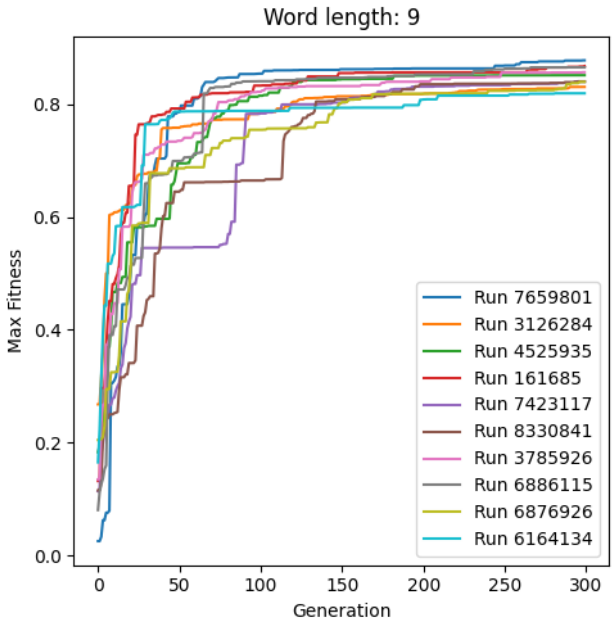

Figure 2: Fitness of the Genetic Algorithm for passwords of length 9

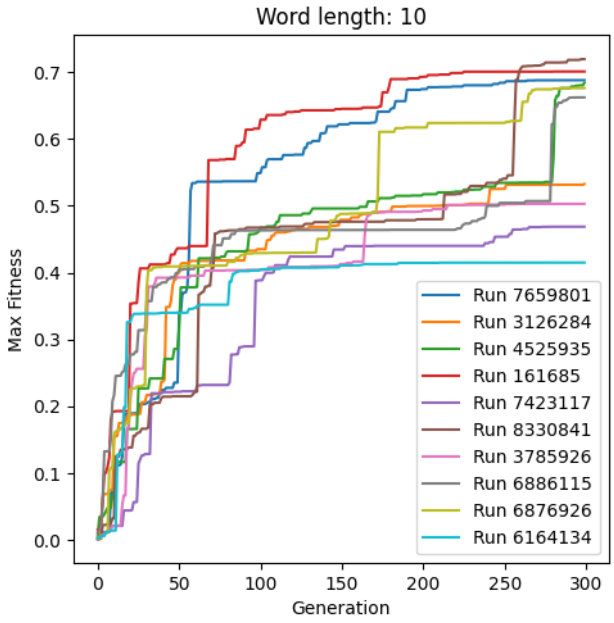

Figure 3: Fitness of the Genetic Algorithm for passwords of length 10

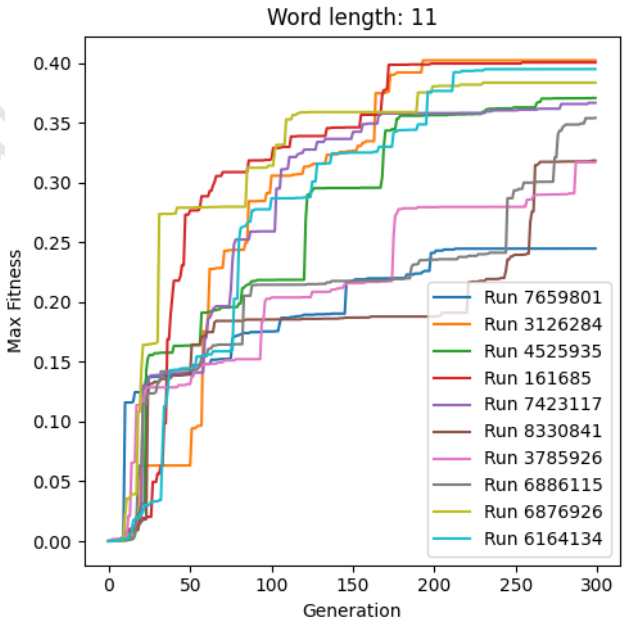

Figure 4: Fitness of the Genetic Algorithm for passwords of length 11

It is possible to observe that the maximum fitness for word lengths 8 and 9 appear to have stabilized before the end of the execution. On the other hand, the maximum fitness for word lengths 10, 11, 12, 13 and 14 were still unstable and could possibly get improved by running the algorithm for more generations. Lastly, the maximum fitness for word length 15 appears to indicate that multiple executions arrived at the same result, possibly because there are not many passwords with that length.

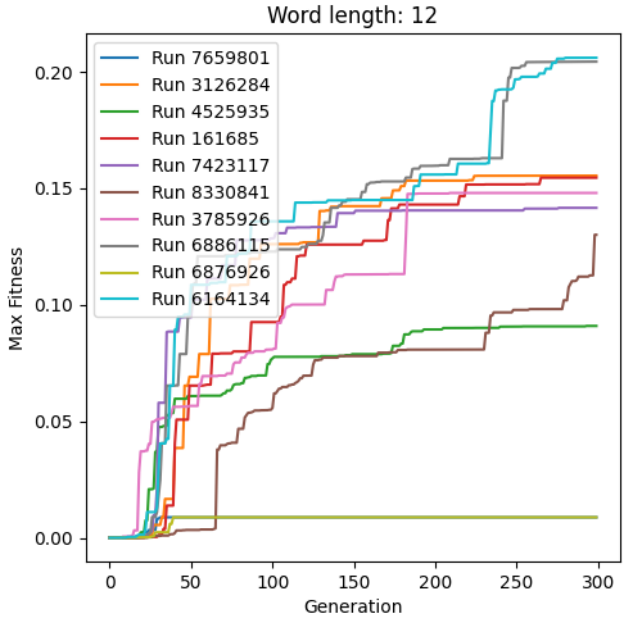

Figure 5: Fitness of the Genetic Algorithm for passwords of length 12

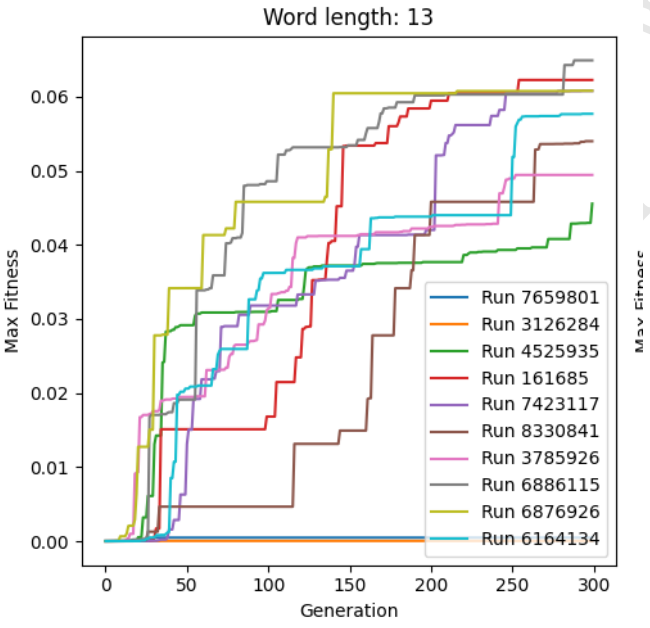

Figure 6: Fitness of the Genetic Algorithm for passwords of length 13

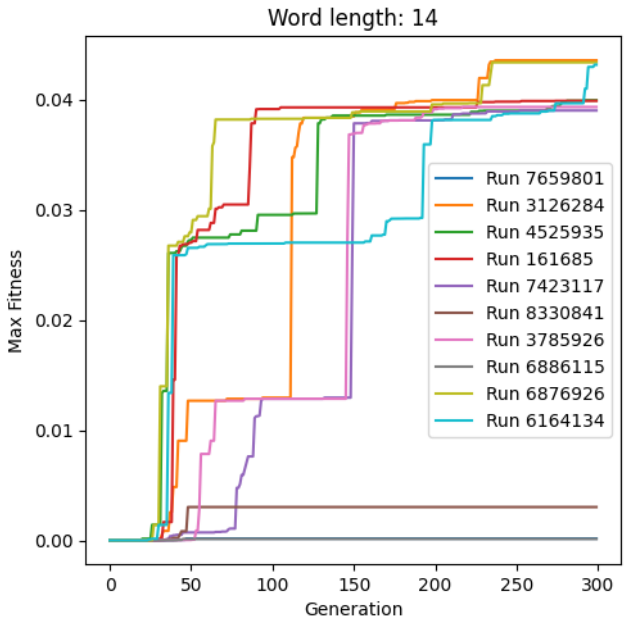

Figure 7: Fitness of the Genetic Algorithm for passwords of length 14

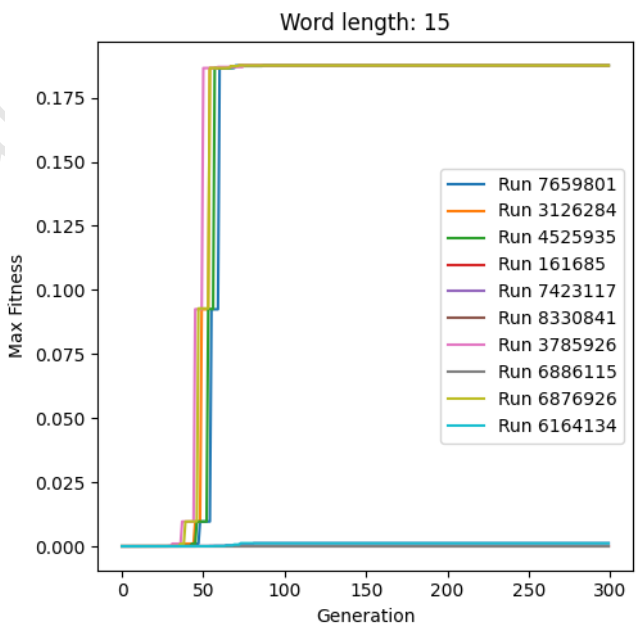

Figure 8: Fitness of the Genetic Algorithm for passwords of length 15

Comparing the results obtained by the genetic algorithm with the results obtained by the current methods, it is possible to observe that the genetic algorithm was able to achieve a fitness greater than the Corporate Masks for all lengths. The algorithm obtained a fitness similar to the Extreme Breach Masks for lengths 8, 13, and 15, and a fitness greater than the Extreme Breach Masks for lengths 12 and 15. However, the algorithm obtained a fitness lower than the Extreme Breach Masks for lengths 9, 10, 11, and 14. A comparison

Table 7: Comparison of the fitness of the Genetic Algorithm with the current methods

Length	Genetic Algorithm	Corporate	Extreme Breach
8	0.998075	0.972079	0.999705
9	0.877505	0.562294	0.955987
10	0.719535	0.386915	0.846787
11	0.402315	0.208118	0.444393
12	0.206068	0.085927	0.175589
13	0.064911	0.032640	0.065720
14	0.043578	0.030719	0.034780
15	0.187582	-	0.187582

of the final results can be seen in table 7. The masks generated by the genetic algorithm can be seen in the appendix.

6 Conclusion

This paper proposed evaluating the Corporate Masks and the Extreme Breach Masks using genetic algorithms to optimize Hashcat masks. Initially, it is interesting to observe that the Corporate Masks obtained a significantly worse fitness than the Extreme Breach Masks for most password lengths, indicating that it is not very efficient for recovering passwords. With the genetic algorithm, it was possible to generate masks that were later compared with the results obtained by the current methods. After this step, it was observed that the genetic algorithm was able to achieve a fitness greater than the Corporate Masks for all lengths. Furthermore, the algorithm obtained a fitness similar to the Extreme Breach Masks for lengths 8, 13, and 15, and a fitness greater than the Extreme Breach Masks for lengths 12 and 15. However, it obtained a fitness lower than the Extreme Breach Masks for lengths 9, 10, 11, and 14.

We can conclude that the results obtained by the genetic algorithm can be used to improve the efficiency of password cracking, by generating more effective masks for some specific lengths. However, for some lengths, the Extreme Breach Masks are still more effective and should be used instead. It is possible that the genetic algorithm could achieve better results by running for more generations, or by using a different choice of parameters or genetic operators. Lastly, different methods could be used to search for the optimal masks, such as simulated annealing or particle swarm optimization.

References

[1] Jens 'atom' Steube, Gabriele 'matrix', and Gristina. 2015. *Hashcat.* https://hashcat.net/

[2] Multiple contributors. 2023. *Optuna.*

[3] golem445. 2020. *Corporate Masks.*

[4] Muris Kurgas. 2018. *CUPP - Common User Passwords Profiler.*

[5] nyxgeek. 2025. *Weakpass 4.* Retrieved February 2, 2025 from https://weakpass.com/wordlists/weakpass_4.txt

[6] Sean T Smith. 2022. *Extreme Breach Masks.*

A Masks generated by the Genetic Algorithm

The masks generated by the genetic algorithm can be seen in the following tables.

A.1 Masks for passwords of length 8

?a?a?l?l?a?l?a?l
?l?a?a?a?a?a?a?a
?u?l?l?l?d?s?s?u
?a?d?a?d?d?a?a?a
?s?a?s?s?s?d?s?s
?s?a?l?l?l?l?l?a
?u?a?a?a?a?a?a?a
?u?d?d?s?d?l?u?a
?u?a?d?a?a?l?d?d
?a?a?s?u?d?u?u?a
?s?a?l?l?l?l?l?s
?d?a?a?a?a?a?a?a
?s?l?a?d?d?l?a?a
?s?a?l?d?l?l?l?s
?a?l?a?d?a?d?a?a
?a?a?a?a?l?d?d?a
?u?l?l?l?s?s?s?u
?a?l?l?a?a?l?a?l
?s?a?s?l?l?l?a?a
?a?a?l?l?a?l?a?d
?a?a?d?d?d?a?a?a
?a?l?l?l?a?a?a?a
?s?a?d?d?l?a?a?l
?s?d?d?d?l?s?a?a
?u?l?s?s?d?d?l?d
?a?l?a?s?u?d?u?u
?s?u?d?a?d?d?u?u
?s?a?a?l?d?a?d?d
?a?d?u?s?l?a?a?s
?u?l?u?a?d?u?l?l

A.2 Masks for passwords of length 9

?d?d?d?a?a?a?a?l?l
?d?d?d?a?a?a?l?l?a
?a?d?l?l?l?s?d?s?l
?l?u?u?s?l?u?a?s?l
?a?l?a?a?a?a?d?d?d
?d?l?l?a?a?u?d?a?d
?a?l?l?l?d?l?l?s?l?d
?a?l?l?l?l?a?l?l?a?a
?a?l?l?l?l?a?l?d?a?a
?a?l?s?d?a?d?s?d?a
?l?a?l?l?l?d?d?l?d?a
?d?l?l?a?a?l?l?a?l
?a?a?l?l?d?l?l?s?l?d
?a?u?u?u?u?u?a?u?a
?l?l?a?d?a?a?s?d?d
?l?l?l?d?a?l?a?a?l
?a?a?l?l?a?l?s?a?d
?d?d?d?a?a?a?d?l?a
?d?d?d?a?a?a?l?d?a
?l?l?l?a?a?d?l?a?l

?a?a?l?l?a?l?l?a?d
?a?a?a?a?a?d?d?d?a
?d?l?l?l?a?u?l?a?l
?d?a?a?l?a?l?l?a?l
?a?u?u?u?u?u?a?d?a
?d?a?d?d?l?s?u?d?a
?u?l?s?a?l?d?l?l?l
?s?a?s?d?s?s?s?s?s

A.3 Masks for passwords of length 10

?a?d?a?d?a?a?d?d?d?d
?a?l?a?l?l?l?l?l?d?d
?a?l?a?d?a?a?d?d?d?d
?a?l?a?l?l?l?l?d?d?d
?a?l?a?l?a?l?d?d?d?d
?a?l?a?l?a?a?d?d?d?s
?d?d?d?d?l?l?a?l?a?a
?a?d?a?l?l?l?l?l?d?d
?a?l?l?l?l?l?l?l?l?d
?a?l?l?l?a?l?s?d?d?d
?l?l?l?l?a?a?d?d?d?d
?l?l?l?l?a?a?d?d?d?l
?a?l?l?l?a?a?d?d?d?d
?d?d?d?d?d?d?a?s?a?a
?a?d?d?d?d?d?a?d?a?a
?a?l?l?l?a?l?l?d?d?d
?d?d?d?d?d?d?a?s?a?a
?a?l?l?l?l?l?l?l?l?l
?d?d?d?d?d?l?a?l?a?a
?a?l?a?l?l?l?l?d?d?d
?a?l?l?s?a?a?d?d?d?d
?l?l?a?d?a?a?d?d?d?d
?a?l?l?l?a?l?l?l?l?d
?a?d?a?d?a?a?d?d?d?d
?d?d?d?d?d?d?a?l?a?a
?a?l?a?l?l?l?l?l?d?d
?d?d?d?d?d?d?a?l?a?a
?a?l?a?l?l?l?l?l?d?d
?u?l?l?l?l?l?a?d?d?a

A.4 Masks for passwords of length 11

?u?l?l?l?l?a?l?d?d?d?d
?l?d?d?s?d?a?d?d?d?d?a
?l?l?l?l?l?l?l?a?d?d?d
?d?d?d?d?a?a?d?l?d?d?a
?u?d?l?l?l?a?l?d?d?d?d
?l?d?d?d?d?l?d?d?l?d?a
?l?d?d?d?d?a?d?u?d?d?a
?a?l?d?d?d?a?d?d?d?d?a
?a?d?d?d?d?a?d?d?d?d?a
?l?l?l?l?l?l?l?l?a?l?d?d
?l?l?l?d?a?a?d?d?d?d?d
?l?d?d?d?s?l?d?d?d?d?u
?d?d?d?d?d?a?d?d?a?l?a
?l?l?l?l?l?a?a?d?d?d?d
?u?l?l?l?l?l?a?d?d?d?d
?d?d?d?d?a?a?d?d?d?d?a

A.5 Masks for passwords of length 12

?l?a?d?d?a?d?d?d?d?d?d?d
?l?l?l?a?d?d?d?d?d?d?d?d
?d?d?d?d?a?d?a?d?d?d?d?d
?l?l?l?l?l?l?l?d?d?d?d?d
?l?l?l?l?l?l?l?l?d?d?d?d
?d?d?d?d?a?d?d?d?d?d?d?d
?l?l?l?l?l?a?d?d?d?d?d?d
?d?d?d?d?a?d?d?d?d?d?d?d
?d?d?d?d?a?d?d?d?d?d?d?a
?l?l?d?d?a?d?d?d?d?d?d?d
?u?l?l?l?l?l?l?d?d?d?d?d
?l?l?d?a?d?d?d?d?d?d?d?d
?d?u?d?d?a?d?d?d?d?d?d?d
?d?d?d?d?a?d?d?d?d?d?d?a
?l?l?l?l?l?l?l?l?d?d?d?d
?d?d?d?d?a?d?d?d?d?d?d?d
?d?d?d?d?a?s?d?d?d?d?d?d
?d?d?d?d?a?d?d?d?d?d?d?d

A.6 Masks for passwords of length 13

?d?d?d?d?d?d?d?d?d?d?l?l?l
?a?a?d?d?d?d?d?d?d?d?d?d
?a?d?d?d?d?d?d?s?d?d?d?d
?u?l?l?d?d?d?d?d?d?d?d?d
?a?d?d?d?d?d?d?d?d?d?d?l
?l?l?a?d?d?d?d?d?d?d?d?d
?a?d?d?d?d?d?l?d?d?d?d?d
?d?d?d?d?d?d?d?d?d?d?l?l
?l?l?l?l?l?l?d?d?d?d?d?d
?l?l?l?l?l?d?d?d?d?d?d?d
?u?l?l?l?l?d?d?d?d?d?d?d

A.7 Masks for passwords of length 14

?l?l?l?d?d?d?d?d?d?d?d?d?d
?l?d?d?d?d?d?d?d?d?d?d?d?d
?d?d?d?d?d?d?d?d?d?d?d?d?d
?l?l?d?d?d?d?d?d?d?d?d?d?d
?d?d?d?d?d?d?d?d?d?d?d?d?d
?d?d?d?d?d?d?d?d?d?d?l?l?l
?l?l?l?l?l?d?d?d?d?d?d?d?d
?l?l?l?l?d?d?d?d?d?d?d?d?d
?u?l?l?l?d?d?d?d?d?d?d?d?d

A.8 Masks for passwords of length 15

?d?d?d?d?d?d?d?d?d?d?d?d?d?d?d
?d?d?d?d?d?d?d?d?d?d?d?d?d?d?d
?l?d?d?d?d?d?d?d?d?d?d?d?d?d?d