Beyond the Dictionary Attack: Enhancing Password Cracking Efficiency through Machine Learning-Induced Mangling Rules

Abstract

In the realm of digital forensics, password recovery is a critical task, with dictionary attacks remaining one of the oldest yet most effective methods. These attacks systematically test strings from pre-defined wordlists. To increase the attack power, developers of cracking tools have introduced password-mangling rules that apply additional modifications like character swapping, substitution, or capitalization. Despite several attempts to automate rule creation that have been proposed over the years, creating a suitable ruleset is still a significant challenge. The current state-of-the-art research lacks a deeper comparison and evaluation of the individual methods and their implications. In this paper, we introduce RuleForge, an ML-based mangling-rule generator that integrates four clustering techniques, 19 mangling rule commands, and configurable rule-command priorities. Our contributions include advanced optimizations, such as an extended rule command set and improved cluster-representative selection. We conduct extensive experiments on real-world datasets, evaluating clustering methods in terms of time, memory use, and hit ratios. Our approach, applied to the MDBSCAN method, achieves up to an 11.67%pt. higher hit ratio than the best yet-known state-of-the-art solution.

Keywords:
Password, Rules, John the Ripper, Hashcat, Clustering

1. Introduction

Since the inception of password authentication in computing, the evolution of password-cracking techniques has been a significant area of focus. Password cracking is used not only by malicious hackers but also by the "good guys" such as law enforcement, cyber defense organizations, penetration testers, and security analysts to measure password strength (Proctor et al., 2002; Vu et al., 2007; Kelley et al., 2012), or simply by people who have forgotten their passwords. In digital forensics, recovering passwords is crucial for unlocking encrypted or password-protected data, making it an indispensable method for investigating cases of cybercrime, fraud, or data breaches.

Among the wide range of strategies invented and employed over the years, dictionary attacks have stood the test of time as one of the oldest yet still prevalent methods of breaching password-secured entry points. These attacks, leveraging a predefined list of potential passwords, exploit the human tendency to use memorable, hence often weak, passwords (Bishop and V. Klein, 1995).

The introduction of password-mangling rules (Peslyak, 2017; Steube, 2024) to dictionary attacks has significantly enhanced their effectivity, enabling attackers to systematically test modifications of candidate passwords far beyond simple wordlist matching. This approach preys on the common practice of creating passwords that are slight variations of dictionary words or predictable patterns (Bishop and V. Klein, 1995). These rules apply a series of modifications, such as character substitution, insertion, deletion, and capitalization, to each entry in a wordlist to expand the attack vector by orders of magnitude.

Regardless of the advancements in password-cracking techniques, the process of creating and optimizing mangling rules has for many years been largely manual, time-consuming, and somewhat esoteric. In recent years, researchers and developers have proposed several methods to automate the rule-creation process (Marechal, 2012; Kacherginsky, 2013; Steube, 2017a; Drdák, 2020; Li et al., 2022). Some of the newest approaches utilize machine learning, namely clustering (Drdák, 2020; Li et al., 2022), with the most recent approach being a method proposed by Li et al. (2022) called MDBSCAN. While the studies show a significant potential of such approaches, they lack a broader comparison of individual clustering methods and rule-creation strategies, leaving room for improvements and further research.

1.1. Contributions

Firstly, we conduct an in-depth examination and comparison of four clustering methods, evaluating their effectiveness for mangling-rule creation. Our analysis includes measuring clustering and password-generation time, memory consumption and hit ratios on real-world datasets. Secondly, we introduce several optimizations to the rule creation process, including an extended rule command set and advanced techniques for selecting cluster representatives. Thirdly, we apply these techniques to MDBSCAN, the best-performing method identified, and benchmark it against the current state-of-the-art approach from Li et al. (2022). Lastly, we compare our methods to other existing rule-creation and password-guessing tools, extending the comparison beyond clustering-based approaches. The results demonstrate that our method surpasses state-of-the-art solutions across multiple scenarios. Notably, we have achieved up to 11.67%pt. improvement in hit ratio over the best-performing yet-known solution. These contributions are showcased through the design and proof-of-concept implementation of RuleForge, a clustering-based mangling-rule generator that serves not only to validate our methodology but also allows other researchers and forensic practitioners to experiment with the automated creation of password-mangling rules.

1.2. Structure of the Paper

The paper is structured as follows. Section 2 overviews existing research in smart password guessing, the history and the current state of using password-mangling rules for dictionary attacks. In Section 3, we propose the design and a proof-of-concept implementation of our machine-learning-based rule generator. This section also describes our proposed enhancements to the rule-creation process. Section 4 describes the experimental evaluation of the rule generator and a comparison of ruleset-creation methods. Finally, Section 5 discusses the achieved results and pinpoints ways for possible future improvements.

2. Background and Related Work

Users frequently choose simple, memorable passwords (Bishop and V. Klein, 1995) that make them vulnerable to intelligent password-guessing techniques that mimic human behavior in password creation. Narayanan and Shmatikov (2005) proposed password guessing based on character distribution represented by Markovian models, later adopted by the famous Hashcat tool (Steube, 2020) as the default method for creating passwords in brute-force attacks. Düermuth et al. (2015) presented OMEN (the Ordered Markov ENumerator), an algorithm based on iterating over bins in order of decreasing likelihood, outperforming previously-known Markov-based password guessers. Weir et al. (2009) introduced password cracking with Probabilistic Context-Free Grammars (PCFG). The method was further improved by Houshmand et al. (2015), who added keyword and multiword patterns, Hranický et al. (2019, 2020), who proposed a faster and a distributed version, and Veras et al. (2014, 2021), who added semantic patterns, dividing password fragments into categories by semantic topics like names, sports, etc. Kanta et al. (2022, 2023) utilized contextual information for creating fine-tailored password dictionaries against specific targets. In recent years, deep-learning approaches for password guessing have been introduced. Ciaramella et al. (2006) studied Principal Component Analysis (PCA) preprocessing and different architectures of neural networks for password guessing. Melicher et al. (2016) deployed the "Fast, Lean, and Accurate" (FLA) technique for measuring password strength based on Recurrent Neural Networks (RNN). Hitaj et al. (2019) proposed creating passwords with Generative Adversarial Networks (GAN) and released the PassGAN generator. Xia et al. (2019) introduced password guessing based on PCFG, Long Short-Term Memory (LSTM) and a model called GENPass based on Convolutional Neural Networks (CNN).

Despite the invention of sophisticated techniques for guessing passwords in the past decades, the dictionary attack is still one of the most widely used methods, often used with additional mangling rules that multiply the number of password candidates and increase the chance of finding the correct password.

2.1. The Evolution of Password-Mangling Rules

The origins of password-mangling rules for dictionary attacks date back to 1991 when Alec Muffet released the legendary Crack program (Muffett, 1996). Crack offered a programmable dictionary generator and mangling rules that applied additional modifications to candidate passwords. The 1995 version 5.0 contained 21 pre-defined rulesets and a cookbook for creating new ones using 29 supported commands like character substitution or appendage. The syntax was similar to those used in state-of-the-art cracking tools like John the Ripper (Peslyak, 2015) and Hashcat (Steube, 2020).

In 1996, Alexander Peslyak, better known as the "Solar Designer," created the *John the Ripper* (JtR) tool as a replacement for the popular Cracker Jack (Jackal, 1993) UNIX password cracker. In addition to a complete redesign of the tool, Peslyak (2015) added support for mangling rules compatible with those used in the original Crack program. Over the years, various improvements to John's rule engine have been added (Peslyak, 2019), including word shifting and memorization.

Jens "atom" Steube later decided to fix the missing multi-threading support in JtR's dictionary attack mode. In 2009, he released the *Hashcat* tool (Steube, 2020), originally called "atomcrack." The initial version was a simple yet very fast dictionary cracker (Steube, 2017b). Hashcat had a native support for password-mangling rules and adopted the syntax and semantics from JtR.

The release of CUDA (NVIDIA Corporation, 2019) and OpenCL (Munshi, 2009) started a revolution in the password cracking area. Cracking program creators quickly reacted by adding GPU support to their tools (Bakker and van der Jagt, 2010; Steube, 2017c; Peslyak, 2019). Steube was no exception and, in 2010, released cudaHashcat and oclHashcat (Steube, 2017c), the latter being eventually transformed into a single unified tool named just "hashcat." OpenCL support was also added to JtR in 2012 (Peslyak, 2019). Unlike Cracker Jack and JtR, Hashcat applied the rules directly inside the GPU kernel, which dramatically reduced the number of necessary PCI-E transfers (Steube, 2017c). Hashcat also extended the repertoire of rules with new ones such as ASCII value incrementation, character block operations, or separator-based character toggling (Steube, 2024). To the best of our knowledge, Hashcat is the only password cracker with an in-kernel rule engine and a self-proclaimed "world's fastest password cracker" (Steube, 2020). This could be true as Hashcat now computes all hash algorithms on OpenCL devices using highly optimized kernels. Moreover, the team Hashcat won several years of DEFCON and DerbyCon "Crack Me If You Can" (CMYIC) contests (Zivadinovic et al., 2016). The latest 2022 v6.2.6 release of Hashcat supports 56 unique mangling-rule commands (Steube, 2024).

2.2. Approaches to Automated Rule Creation

While both Hashcat and JtR provide several default rulesets and their respective websites document the syntax and semantics of the supported mangling rules (Peslyak, 2017; Steube, 2024), creating new rulesets is not a trivial task. To this day, several approaches have been proposed to automate the rule creation (Marechal, 2012; Kacherginsky, 2013; Steube, 2017a; Drdák, 2020; Li et al., 2022).

The *hashcat-utils* repository contains a simple utility called *generate-rules.c*, written by Jens Steube. The tool generates a specified number of random rules from either a time-based or a user-specified seed (Steube, 2017a). While the generated ruleset can theoretically be used for password cracking, their form is purely random without any deeper meaning, as there is no sophisticated system for their creation. From a research perspective, the tool can be used as a baseline rule creator for comparison with more advanced techniques.

Marechal (2012) proposed an algorithm for mangling-rule creation. The idea was to start with a list of mangling rules, either handpicked or randomly generated. Those were applied to a wordlist, resulting in mangled passwords. Next, Marechal proposed finding the largest common substring of the words and computing the required append or prepend operations to produce it from the remaining passwords. These operations resulted in rules that were then ranked by the number of passwords created. Marechal also released a proof-of-concept implementation called *rulesfinder*, which is still maintained to this day (Marechal, 2022). While the approach is working, its major drawback is the need for an existing ruleset.

A different technique was proposed by Peter "iphelix" Kacherginsky in 2013 with a proof-on-concept implementation as a part of the Password Analysis and Cracking Kit (PACK) (Kacherginsky, 2013). The PACK/Rulegen generator uses a similarity-based approach but does not apply clustering in the true sense of the word. For each candidate password, it creates a group of similar passwords. For each group of passwords, Rulegen calculates the Levenshtein distance (Levenshtein, 1966) between the originating password and other passwords in the group. By analyzing the calculated distances, the optimal sequence of operations is found and described by a series of rules (Kacherginsky, 2013).

Drdák (2020) researched possibilities for adopting machine learning for automated mangling-rule creation. The idea was to apply clustering to an existing password dictionary (a "training dictionary") to create clusters of passwords with similar syntax. One password was chosen as a representative of each cluster. Mangling rules were then automatically created to describe necessary modifications for transforming the representative to the remaining passwords in the cluster. Drdák chose the Affinity Propagation (AP) (Frey and Dueck, 2007) method and created a proof-of-concept implementation that showed promising experimental results. Drdák later published the findings in his bachelor's thesis (Drdák, 2020). While the general idea has been later proven usable by other researchers (Li et al., 2022), Drdák's study had its limitations. Firstly, Drdák tested only a single clustering method. Secondly, a significant issue was an extremely long computing of the distance matrix for larger training dictionaries.

The same issue was independently identified and later addressed by Li et al. (2022). They proposed a novel method called MDBSCAN, a modified version of the classic DBSCAN algorithm (Ester et al., 1996), that was customized for clustering passwords. To accelerate the distance calculation, they used the SymSpell (Garbe, 2012) fuzzy search algorithm. The research on using MDBSCAN for the rule generation problem shows great success in experimental results, even compared to PCFG (Weir et al., 2009) and PassGAN (Hitaj et al., 2019).

While MDBSCAN (Li et al., 2022) is, to the best of our knowledge, the most efficient clustering-based technique for automated rule creation, the authors focused mainly on DBSCAN and MDBSCAN and have not tested other clustering methods like Affinity Propagation (Frey and Dueck, 2007) or Hierarchical Agglomerative Clustering (HAC) (Jiawei et al., 2012). The rule-creation method is also not optimal, namely in terms of cluster representative selection, and, as we demonstrate in our paper, fails in certain scenarios. Also, a selection of only 14 rules was implemented. Moreover, we have not found any released proof-of-concept implementation of the proposed method.

2.3. Research goals

Although several approaches for automated mangling-rule creation have been proposed, significant gaps and unanswered questions remain. To fill these gaps and advance the state of the art in the field we have decided to:

1. Compare tested and yet-untested clustering methods: DBSCAN (Ester et al., 1996), MDBSCAN (Li et al., 2022), AP (Frey and Dueck, 2007), HAC (Jiawei et al., 2012).
2. Implement missing rule commands and experimentally verify their contributions.
3. Explore other possibilities for choosing a cluster representative and verify their benefits.
4. Compare these clustering-based approaches to other mangling-rule creation methods like PACK/Rulegen and other password-guessing tools like OMEN, etc.
5. Create an open-source proof-of-concept implementation to allow researchers and forensics practitioners to experiment with automated rule creation.

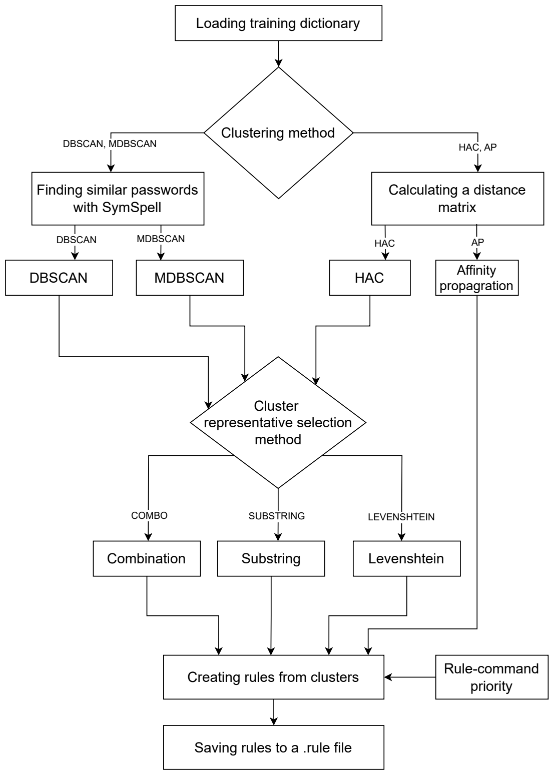

Loading training dictionary
Clustering method
DBSCAN, MDBSCAN
HAC, AP
Finding similar passwords with SymSpell
Calculating a distance matrix
DBSCAN
MDBSCAN
HAC
AP
DBSCAN
MDBSCAN
HAC
Affinity propagation
Cluster representative selection method
COMBO
SUBSTRING
LEVENSHTEIN
Combination
Substring
Levenshtein
Creating rules from clusters
Rule-command priority
Saving rules to a .rule file

Figure 1: Rule generation process

3. The Proposed Mangling-Rule Generator

To fulfill the research goals from Section 2 and also to provide a tool for both experimental and actual password-cracking purposes, we propose a design and a proof-of-concept implementation of RuleForge, an ML-based mangling-rule generator with four clustering methods: AP (Frey and Dueck, 2007), HAC (Jiawei et al., 2012), DBSCAN (Ester et al., 1996), and MDBSCAN (Li et al., 2022). Our tool is also equipped with an extended rule command set, enhanced methods for choosing cluster representatives, and configurable rule command priorities.

3.1. Design

The process of rule generation with RuleForge consists of several key steps, illustrated in Fig. 1. The workflow starts with processing the training password dictionary. For DBSCAN and MDBSCAN clustering methods, we find similar passwords according to the Damerau–Levenshtein (Damerau, 1964) distance and use the SymSpell (Garbe, 2012) fuzzy search algorithm to accelerate the process, like Li et al. (2022) proposed. For AP and HAC, we calculate a classic edit-distance matrix utilizing the Levenshtein distance metric (Levenshtein, 1966). Using the selected method, passwords clusters are created.

Next, we select strings to be considered representatives of their respective cluster. The primary reason for

Table 1: Rule commands implemented in RuleForge, applied on "Password"

Rule	Description	E.g.	Output
:	Do nothing	:	Password
l	Lowercase all letters	l	password
u	Uppercase all letters	c	PASSWORD
c	Uppercase all letters	c	PASSWORD
t	Toggle case	t	pASSWORD
TN	Toggle case at position N	T2	PaSsword
zN	Duplicate first character N times	z2	PPPassword
ZN	Duplicate last character N times	Z2	Passworddd
$X	Append character X to end	$1	Password1
^X	Prepend character X to front	^_	_Password
[	Delete first character	[	assword
]	Delete last character	]	Passwor
DN	Delete character at position N	D2	Pasword
iNX	Insert character X at position N	i4!	Pass!word
oNX	Overwrite ch. at pos. N with X	o2$	Pa$sword
}	Rotate the word right	}	dPasswor
{	Rotate the word left	{	asswordP
r	Reverse the entire word	r	drowssaP
sXY	Replace all Xes with Y	ss$	Pa$$word

choosing representatives is to create rules based on comparing passwords within a cluster to their given representative and model necessary transformations by the produced rules. With AP, the representative is determined by the clustering method itself. For the remaining methods, the representative is selected using one of the methods discussed in Section 3.3. DBSCAN and MDBSCAN do not necessarily categorize every element into a cluster; they put these unclusterable elements into an "outlier cluster." Creating rules from this cluster is, understandably, ineffective. Therefore, we added an option to exclude these outliers from rule creation.

Once clusters are created and their representatives selected, the process of generating passwords starts. RuleForge generates rules by leveraging a user-provided rule priority file, specifying the sequence in which rules are formulated. The process is thoroughly explained in Section 3.4. Finally, RuleForge creates an output ruleset consisting of rule commands sorted by frequency or, optionally, a ruleset with a user-specified top number of rules.

3.2. Clustering Methods

As discussed above, RuleForge uses clustering to find groups of similar passwords. Once identified, we can notice differences between passwords in a group. These differences typically reveal how users create their passwords and serve as anchors for rule identification. Applying different clustering methods may lead to varied ways of grouping passwords and creating diverse rules. By experimenting with these methods within the tool, it is possible to attain varying password-cracking success rates. The following paragraphs describe the supported clustering methods and their use in RuleForge.

AP. Affinity Propagation considers all objects as possible exemplars, exchanging real-valued messages between them until a high-quality set of exemplars (and corresponding clusters) emerges (Frey and Dueck, 2007). The two key parameters that we need to specify are *damping*, which is the extent to which the current value is maintained relative to incoming values, and *convergence_iter*, which represents the number of iterations with no change in the estimated clusters that stop the convergence. In the configuration of the proposed rule generator, the parameter *convergence_iter* is set at 15, while the *damping* parameter is adjusted to fall within the 0.6 to 0.8 range. These values were selected to optimize the clustering process.

HAC. The Hierarchical Agglomerative Clustering method places each object into a cluster of its own. The clusters are then merged into larger clusters according to the criterion set by a parameter *distance_threshold* (Jiawei et al., 2012). For the HAC approach, we set the *distance_threshold* parameter to a value of 3, which has been experimentally verified to be the most effective for our use case.

DBSCAN. In order to form clusters, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) first identifies core points—objects that have at least *MinPts* neighbors (i.e., objects whose Damerau–Levenshtein distance (Damerau, 1964) to the core point is less or equal to ϵ). Each core point at first forms a cluster with just itself as its only member. Iteratively, clusters are extended or merged by adding objects that are neighbors of core points in each cluster. The result is a set of clusters and a set of non-clustered noise objects. *MinPts* and ϵ are user-definable parameters of the method (Ester et al., 1996). In DBSCAN clustering, we set ϵ as 1 and *MinPts* to 3. In our experience, higher values of ϵ lead the algorithm to degenerate into outputting a single cluster containing the vast majority of passwords. Higher values of *MinPts* cause the majority of passwords to be categorized as noise.

MDBSCAN. When used for clustering passwords, the DBSCAN algorithm has a tendency to form one large cluster containing the majority of passwords. To improve the granularity of its output clusters, Li et al. (2022) proposed a modified version titled MDBSCAN (Modified Density-Based Spatial Clustering of Applications with Noise). The modification lies in using an additional truncation metric to determine whether an object belongs to a cluster. When performing clustering, a note is made of the initial element of the cluster. An object is only added to a cluster if its Jaro–Winkler distance[1] (Winkler, 1990) to the initial point is less or equal to ϵ_2[2] with all else being unchanged from the DBSCAN algorithm. The user-definable parameters of this method are ϵ_1, ϵ_2, and *MinPts*, where ϵ_1 and *MinPts* are equivalent to the parameters of the DBSCAN algorithm. For our use, we set ϵ_1 to 2, ϵ_2 to 0.25 and *MinPts* to 3. The truncation feature of MDSBCAN allows for using a higher value of ϵ_1 than with DBSCAN—the large cluster generated by DBSCAN gets "broken up" by MDBSCAN—which is helpful because a higher value leads to fewer values being categorized as noise. However, going above the value 2 again leads to enormously large clusters. Lower values of ϵ_2 lead to the creation of too many useless single-password clusters, whereas higher values lead to the creation of too large clusters. And with *MinPts*, the results are the same as with DBSCAN.

Algorithm 1 Rule identification

```
Global: Vector $\overrightarrow{r_p} = [r_1, r_2, \ldots, r_{19}]$ of rule commands in priority order, where $r_1$ and $r_{19}$ are commands with the highest and lowest priority respectively
Input: Password $P$ from a cluster $c_i$, representative $P_{rep}$ of a cluster $c_i$
Output: Sequence of rule commands $R$ generated by transforming $P$ to $P_{rep}$
while $P \neq P_{rep}$ do
    $r_f = None$    ▷ Initialize $r_f$ value to check whether
                    ▷ a suitable rule command was found.
    for each rule command $r \in \overrightarrow{r_p}$ do
        Calculate the number of transformations $n$ using $levenshtein\_distance(P, P_{rep})$.
        Create a password $P_m$ by modifying password $P$ with rule command $r$.
        Calculate the new number of transformations $n_m$ using $levenshtein\_distance(P_m, P_{rep})$.
        if $n_m < n$ then
            ▷ Suitable rule command $r_f$ found.
            $P = P_m$
            $r_f = r$
            break    ▷ Stop looking for other commands.
    if $r_f \neq None$ then
        $R.append(r_f)$
    else
        break    ▷ No other possible modification found.
return $R$    ▷ Return the final command sequence.
```

3.3. Choosing cluster representatives

Once the clusters are created, it is necessary to select a represenative for each cluster and search for possible transformations to the remaining passwords in the cluster.

Levenshtein Method. Existing works that use clustering for rule creation (Drdák, 2020; Li et al., 2022) always choose a representative as a concrete password from the cluster, concretely, the one with the lowest mean Levenshtein distance to others. Therefore, we call it this technique the "Levenshtein Method." Nevertheless, this approach is rather limiting. Assume the password clusters in Fig. 2. The blue candidates are representatives chosen by this method. In the leftmost cluster, `hello1` is selected as a representative. Assuming the rule commands from Table

[1]The Jaro–Winkler distance is a real number in the range 0 to 1.

[2]In the case of merging two clusters, one initial point is disregarded. The choice of object to disregard is not specified.

1, possible transformations to `hello2` and `hello3` are (1) deleting the last character and appending "2" or "3", (2) overwriting the 6th character with "2" or "3", or (3) replacing all occurrences of "1" with "2" or "3." Obviously, such modifications are only usable in very specific cases. What we want are rules that have general use.

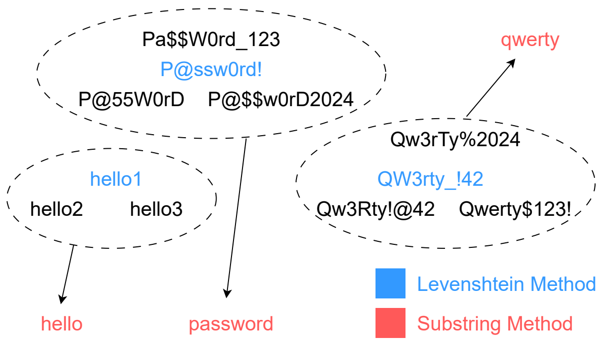


Figure 2: A visual comparison of cluster representative selection methods

Substring Method. Therefore, we propose an alternative called the "Substring Method" which works as follows. Firstly, we transform all characters of the substring to lowercase. This way we obtain more general words to which capitalization rules may easily be applied. Next, we undo all "leetspeak-based" transformations like $a \rightarrow$ @ or $s \rightarrow$ \$. As we have observed, removing leetspeak often allows the extraction of original words and sentence fragments that inspired the password creator. Finally, we calculate the longest common substring of all passwords in the cluster. The resulting string is the cluster representative. Note that the representative may or may not be an actual existing password from the cluster.

Combo Method. While the Substring Method allows the creation of more generally usable rules, we achieved the best experimental results (See Section 4.) by using their combination. This "Combo Method" works as follows:

1. For each cluster, choose a representative using the Levenshtein Method and generate all possible rules (See Section 3.4.).
2. For each cluster, choose a representative using the Substring Method. Generate all possible rules to extend the previously created ruleset.
3. The top n most frequent rules create the final ruleset.

3.4. Rule Creation

The rule-generation process utilizes the Levenshtein distance (Levenshtein, 1966) to determine the number of editing operations required to transform a password within a cluster to its representative. This method of measuring edit distance helps find specific rules that, when used on passwords, make the edit distance smaller. A rule that decreases the edit distance is deemed appropriate and incorporated into the rule set. Multiple rules (such as sXY and oNX) may achieve identical modifications in certain instances. Therefore, RuleForge introduces a rule priority system, specifying which rules it prioritizes during the rule-creation process. These rules can be specified in the input rule-priority file, where one can determine which rules RuleForge should generate and in which priority. The rule generator proposed by Li et al. (2022) supports 14 different types of rules. With RuleForge, we expanded the number of rules to 19. Rules that RuleForge supports are displayed in Table 1. Other Hashcat rules that have not yet been implemented are considered for future work. This approach of using rule priority enables the exploration of different rule-priority configurations, leading to different password-cracking hit rates. The rule generation process is illustrated in Algorithm 1.

3.5. Proof-of-Concept Implementation

To create a proof-of-concept implementation of RuleForge, we chose a combination of two languages: Python and C#. Python mainly for its popularity, common knowledge among researchers, extensive computation and data-analysis support, and wide compatibility with machine-learning libraries. And C# chiefly because of our dependence on the SymSpell library, which is written therein, but also due to its better multithreading performance, which is useful in effectively computing distance matrices. We used the Python Scikit Learn[3] library to perform HAC and AP clustering. For DBSCAN and MDBSCAN, we made our own implementation in C# and made use of the SymSpell library. MDBSCAN was implemented, to the best of our efforts, according to the paper from Li et al. (2022). RuleForge is accessible on (** Blinded **) under the MIT License.

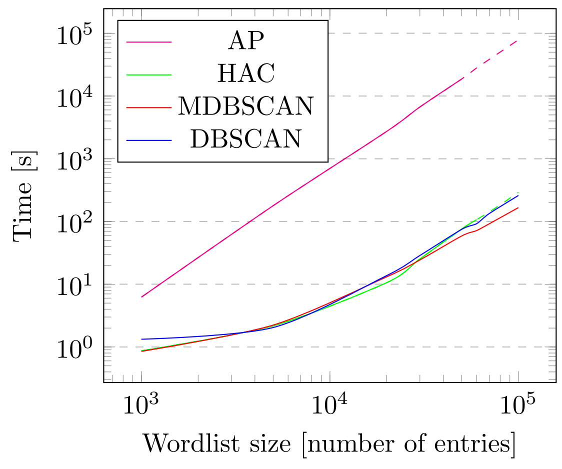
4. Experimental Results

4.1. Benchmarking of Clustering and Rule Creation

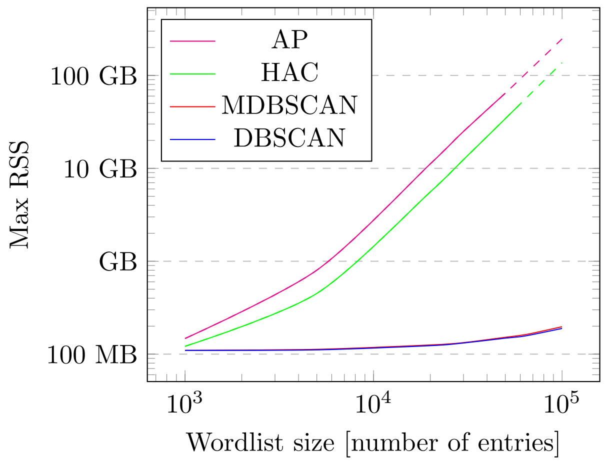
In this section, we analyze clustering and rule creation with the discussed methods and evaluate them on real-world datasets. Next, we compare the original (Li et al., 2022) and RuleForge's implementations of MDBSCAN, focusing also on different representative-selection methods. Finally, we compare the performance of RuleForge with other techniques and state-of-the-art tools. In the experiments, we use various password dictionaries. Table 2 describes each of them. All are also available on (** blinded **). Note, for some experiments, we use abbreviations (marked as "Ab.") instead of full names.

Time and space complexities are critical deciding factors, and thus, we first analyzed the computing time and memory requirements for clustering and rule creation with

[3]`https://scikit-learn.org/`



(a)



(b)

Figure 3: Time (a) and peak memory (b) requirements for generating rules from wordlists of different sizes

the four examined methods. For this purpose, we ran a series of benchmarks with dictionaries of different sizes on an AMD Ryzen 5 2600X workstation with 64 GB of RAM. The inputs were pseudo-random subsamples with 1,000 to 100,000 passwords from the RockYou dictionary. For each subsample, we used RuleForge in the "Combo" mode (See Section 3.) with AP, HAC, DBSCAN, and MDBSCAN. Fig. 3 shows the time and maximum resident set size (RSS) required to create clusters and generate mangling rules for inputs of different sizes. Dashed lines indicate extrapolated values for wordlist-size values that could not be measured due to a lack of memory in our workstation.

Based on the execution times, DBSCAN, MDBSCAN, and HAC demonstrate comparable and decent performance, with all three methods processing dictionaries containing 100,000 passwords in under 5 minutes. In contrast, AP exhibits significantly poorer performance, requiring about 21 hours to handle the same workload.

In terms of memory requirements, DBSCAN and MDBSCAN show linear complexity, whereas AP and HAC display quadratic complexity, which is due to the necessity of computing the full distance matrix, as mandated by the Scikit Learn library. In concrete values, at 100,000 passwords, DBSCAN and MDSBCAN require about 200 MB of memory, HAC requires 137 GB, and AP 247 GB. DBSCAN and MDBSCAN's efficient linear memory usage is achieved by leveraging the SymSpell library (Garbe, 2012) for finding similar passwords.

The doubling of memory usage from HAC to AP is caused by the fact that HAC can utilize a 1-byte integer distance matrix, whereas AP requires at least a 2-byte float distance matrix. Note that the memory requirements for AP and HAC are much higher than just the size of their distance matrices (for 100,000 passwords, this would be 10 GB and 20 GB for HAC and AP, respectively). This is caused–as we observed–by some inefficiencies in Scikit Learn's handling of distance-matrix clustering.

DBSCAN and MDBSCAN are thus very well suited for processing any-sized dictionaries. AP and HAC, on the other hand, are barely usable for larger dictionaries due to extensive memory requirements.

4.2. Cross-checking Clustering Methods and Rule Creation on Different Wordlists

Next, we compared the achievable hit ratios of rules generated from the clusters produced by each of the four clustering methods. For this purpose, we employed RuleForge in the "Combo" mode, except for AP which selects cluster representatives natively. We experimented with four training (t) dictionaries (tl, r65, ms, dw) for creating rulesets. Each ruleset was gradually applied to words from four attack dictionaries (pr, tm, en, dp) in a dictionary-cracking session with Hashcat in plaintext mode. The target "hashlists" were RockYou-75 and Xato-net-100k. For both, we calculated the number of hits. The dictionaries' descriptions are available in Table 2. To maintain fair conditions for all methods, we used the best (i.e. first) 1,000 rules generated by each method.

Table 3 shows the hit ratios on RockYou-75 and Table 4 on Xato-net-100k. In average, MDBSCAN produced rules with the best hit ratios. The second best-achiving method was AP which, in a few cases (r65+en on both targets; dw+pr and dw+dp on Xato-net-100k) even outperformed MDBSCAN. We believe this success of AP is caused by its virtually optimal cluster representative selection, but this is reclaimed by high computational and memory costs, as shown in the previous experiment.

4.3. Comparison of MDBSCAN-based Implementations

In this experiment, we focused on the best-performing MDBSCAN method and compared its implementations. As a baseline, we used the original version from Li et al. (2022), which we compared with RuleForge in the Levenshtein (RF-leven), Substring (RF-substr), and Combo (RF-combo) modes. Training, attack, and target dictionaries remained the same as in the previous experiment. Likewise, we used the first 1,000 generated rules.

Table 2: Password dictionaries for experimental evaluation

Ab.	**Name**	**Passwords**	**Description**
tl	tuscl-m	37,006	Tuscl leak (ASCII, $\leq$ 10 ch.)
r65	rockyou-65-m	29,596	RockYou subsample (ASCII, $\leq$ 10 ch.)
ms	myspace-m	30,000	MySpace leak (ASCII, $\leq$ 10 ch.)
dw	darkweb2017-top10k-m	9,999	Darkweb subsample (ASCII, $\leq$ 10 ch.)
tm	10-million-list-top-10000	9,999	9,999 Passwords from the 10-million list
pr	probable-v2-top12000	12,645	A subsample from the probable dictionary
en	english-6	15,542	English words up to 6 characters
dp	default-passwords	1,315	Commonly used passwords
-	RockYou-960	960,000	A 960k subsample of the RockYou leak
-	rockyou-75-m	59,090	ASCII subsample of the RockYou leak
-	Xato-net-100k	99,987	Top passwords from the Xato 10m dataset
-	phpbb-m	184,344	ASCII passwords from phpBB leak

Table 3: Attacks on rockyou-75-m

Rules		Hit ratio			
t	Method	pr	tm	en	dp
tl	mdbscan	56.54%	51.56%	22.60%	2.60%
	dbscan	47.46%	40.13%	16.48%	1.89%
	hac	48.61%	42.40%	17.82%	1.95%
	ap	53.49%	47.32%	20.43%	2.29%
r65	mdbscan	57.43%	53.23%	23.22%	2.66%
	dbscan	47.28%	39.95%	16.52%	1.88%
	hac	44.24%	37.49%	16.88%	1.89%
	ap	57.14%	51.46%	23.31%	2.61%
ms	mdbscan	55.85%	50.15%	21.30%	2.43%
	dbscan	48.48%	41.34%	16.90%	1.87%
	hac	51.35%	46.40%	19.07%	2.10%
	ap	49.72%	42.61%	17.37%	1.90%
dw	mdbscan	55.99%	52.05%	23.02%	2.72%
	dbscan	47.62%	40.25%	17.13%	2.61%
	hac	45.11%	38.53%	17.84%	1.84%
	ap	55.09%	49.57%	22.34%	2.68%

Table 4: Attacks on Xato-net-100k

Rules		Hit ratio			
t	Method	pr	tm	en	dp
tl	mdbscan	40.91%	48.26%	21.17%	3.44%
	dbscan	36.18%	42.88%	17.77%	2.80%
	hac	34.89%	43.15%	17.62%	3.01%
	ap	39.41%	46.25%	20.06%	3.13%
r65	mdbscan	40.98%	49.32%	21.27%	3.67%
	dbscan	36.06%	42.44%	17.68%	2.74%
	hac	29.57%	35.93%	16.79%	2.35%
	ap	40.36%	47.11%	21.34%	3.40%
ms	mdbscan	39.62%	46.46%	19.75 %	3.17%
	dbscan	36.38%	43.13%	17.87%	2.77%
	hac	34.95%	42.89%	17.40%	2.78%
	ap	37.26%	43.69%	18.11%	2.28%
dw	mdbscan	41.39%	49.77%	21.50%	3.84%
	dbscan	36.77%	42.93%	18.29%	3.40%
	hac	30.00%	34.78%	17.31%	2.41%
	ap	41.51%	48.66%	21.45%	3.86%

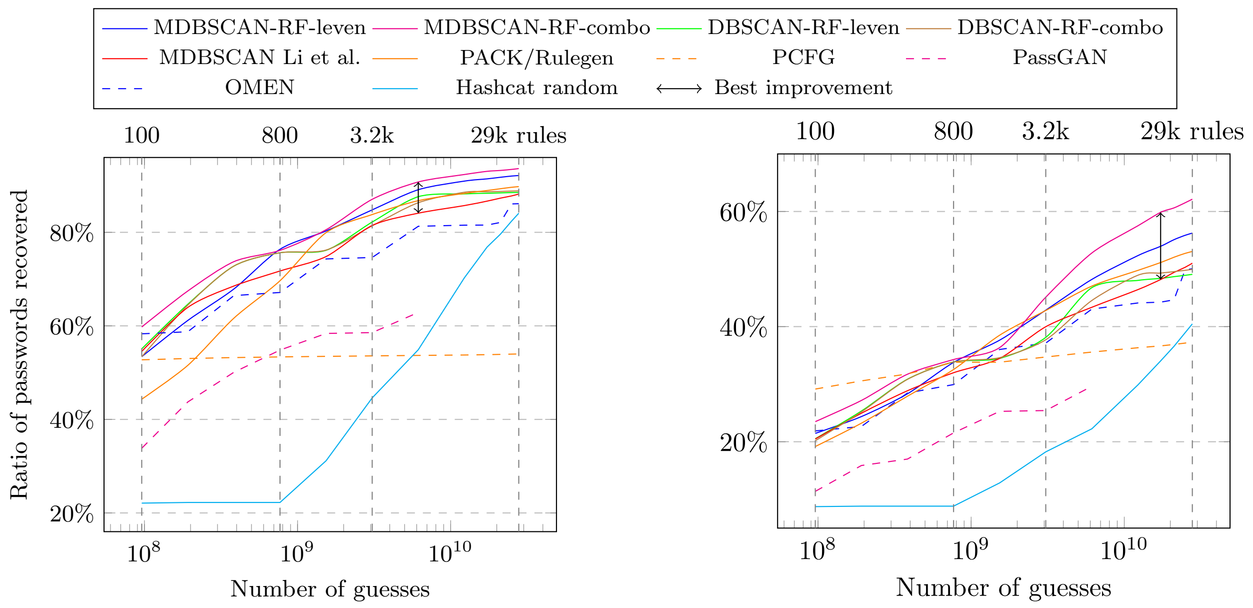
Table 5 describes the hit ratio of attacks on RockYou-75 and Table 6 on Xato-net-100k. RF-combo produced the highest average hit ratio and, in all cases, outperformed the original version from Li et al. (2022), emphasizing our contributions. Interestingly, in the r65+en attack on RockYou-75, the Substring Method resulted in a higher hit ratio than Combo. Note that this is the same combination as where AP produced better results than MDBSCAN. Similarly, in the tl+tm attack on Xato-Net-100k, the Levenshtein Method also performed better than Combo. Such anomalies are caused by the nature of passwords in the chosen dictionaries and show that there is no optimal method for all cases.

4.4. Comparison of Rule-Creation Methods

In the last experiment, we compared hit rates of dictionary attacks with rulesets generated by different methods. As both the training dataset and the attack wordlist, we used a pseudo-random subsample of 960,000 passwords from the RockYou dataset, named "RockYou-960." We then conducted a series of cracking sessions with the first n rules from the ruleset, where $n = 100, \ldots, 29000$, and measured the hit rate on Xato-net-100k and phpbb-m dictionaries from Table 2. We tested the original MDBSCAN, as proposed by Li et al. (2022), and RuleForge's implementation of MDBSCAN and DBSCAN in both Levenshtein and Combo modes.

AP and HAC were not used in this experiment due to exceptional memory requirements. Considering a single byte for password distance, the matrix would require a memory of ∼922 GB or ∼461 GB if a triangular matrix was used, which was beyond the capabilities of our experimental machine.

To compare our attacks in a broader scope, we also deployed several tools from related work (See Section 2.). Concretely, we tested iphelix's PACK/Rulegen (Kacherginsky, 2013). Next, we used PCFG as originally proposed by Weir et al. (2009), i.e., without enhancements like Markov, etc. We also deployed the Ordered Markov ENumerator (OMEN), proposed by Düermuth et al. (2015), and PassGAN by Hitaj et al. (2019). As the last three methods do not produce mangling rules, equivalent num-



(a) Target: Xato-net-100k **(b)** Target: phpbb-m

Figure 4: Password recovery performance comparison (Training: RockYou960, Attack: RockYou960)

Table 5: Attacking rockyou-75-m: MDBSCAN RF vs. Li

Rules		Hit ratio			
t	Method	pr	tm	en	dp
tl	Li et al.	52.44%	46.04%	18.55%	2.19%
	RF-leven	55.12%	51.45%	21.10%	2.53%
	RF-substr	53.42%	48.22%	22.34%	2.36%
	RF-combo	56.54%	51.56%	22.60%	2.60%
r65	Li et al.	55.14%	50.49%	19.41%	2.30%
	RF-leven	55.83%	51.70%	21.44%	2.50%
	RF-substr	53.65%	47.69%	23.76%	2.51%
	RF-combo	57.43%	53.23%	23.22%	2.66%
ms	Li et al.	51.19%	43.96%	17.26%	2.10%
	RF-leven	51.06%	44.41%	18.04%	2.06%
	RF-substr	52.76%	48.08%	20.12%	2.26%
	RF-combo	55.85%	50.15%	21.30%	2.43%
dw	Li et al.	52.49%	45.87%	18.42%	2.27%
	RF-leven	54.01%	49.84%	20.91%	2.58%
	RF-substr	50.99%	44.69%	20.48%	2.24%
	RF-combo	55.99%	52.05%	23.02%	2.72%

Table 6: Attacking Xato-Net-100k: MDBSCAN RF vs. Li

Rules		Hit ratio			
t	Method	pr	tm	en	dp
tl	Li et al.	39.16%	43.33%	18.80%	2.91%
	RF-leven	40.84%	49.11%	20.27%	3.53%
	RF-substr	37.11%	44.11%	21.16%	3.06%
	RF-combo	40.91%	48.26%	21.17%	3.44%
r65	Li et al.	39.11%	44.57%	18.29%	2.92%
	RF-leven	40.40%	48.76%	19.93%	3.80%
	RF-substr	33.64%	40.27%	20.70%	2.85%
	RF-combo	40.98%	49.32%	21.27%	3.67%
ms	Li et al.	37.00%	39.80%	17.51%	2.61%
	RF-leven	37.93%	44.47%	18.41%	2.96%
	RF-substr	35.03%	41.82%	17.74%	2.56%
	RF-combo	39.62%	46.46%	19.75%	3.17%
dw	Li et al.	39.10%	42.55%	18.60%	3.00%
	RF-leven	40.66%	48.71%	20.30%	3.74%
	RF-substr	34.28%	39.53%	18.48%	2.66%
	RF-combo	41.39%	49.77%	21.50%	3.84%

bers of password guesses were generated instead, using RockYou-960 as the training dictionary for creating models. Finally, we used a random ruleset generated by Hashcat to serve as a baseline for other methods.

Experimental results are displayed in Fig. 4. Note that the results from our RuleForge generator use shorthands in the legend: `(M)DBSCAN-RF-{leven,combo}`. The horizontal axis displays the amount of rules on the top and the corresponding number of guesses on the bottom. The guess count is always the size of the dictionary multiplied by the current number of rules. The vertical axis displays the hit ratio. Solid lines represent methods that generate and utilize password-mangling rules, while dashed lines signify other password-guessing methods.

The best average hit ratio was achieved by RuleForge's MDBSCAN in the Combo mode, which also showed the best absolute hit ratio in most measurements. Anomalies were observed at around 800 rules on Xato-net-100k, where it was briefly exceeded by the classic Levenshtein method of RuleForge, and around 1600 rules on phpbb-net, where PACK performed better than MDBSCAN. Interestingly, for lower amounts of guesses on phpbb-m, all methods were surpassed by PCFG, which then degraded to one of the worst methods in our scenario. RuleForge's DBSCAN in both modes also performed well and, in many measurements, exceeded the original MDBSCAN from Li et al. (2022).

To best quantify the benefits of our improvements to the original work, the most important factor is the difference between MDBSCAN-RF-combo (the solid purple line) and MDBSCAN Li et al. (the solid red line). The biggest measured difference between our implementation and the original MDBSCAN was a 6.68%pt. improvement at 6,400 rules on Xato and a 11.67%pt. improvement at 18,000 rules on phpbb-m. Those are marked by black line segments with arrows.

Contrary to the experiments of Li et al. (2022), PACK surpassed the original version of MDBSCAN at higher guess counts and even outperformed MDBSCAN-RF over a specific short range on phpbb-m. OMEN performed slightly worse than the previously-mentioned rule-based methods but still followed closely. The nature of password guessing with Markovian chains creates a curve with a stairs-like shape. PCFG-based guessing generates passwords in a probability order, starting from the most probable candidate password. Interestingly, PCFG showed much better performance on phpbb-m, where it had the highest hit ratios on smaller amounts of guesses, than on Xato-net-100k, where the increments in hit ratio were minimal. PassGAN performed rather poorly, notably at lower amounts of guesses. Execution times were also high, which prevented us from conducting measurements for high numbers of guesses, as it would take weeks to generate the passwords. Its success rate grew with increasing numbers of guesses but never exceeded OMEN or the clustering-based methods. As anticipated, the randomly generated ruleset showed the lowest hit ratios.

5. Conclusion

Our research demonstrates the significant potential of the clustering-based generation of password-mangling rules in enhancing dictionary attacks, often yielding success ratios that surpass those of other state-of-the-art password-guessing methods. Appropriately tailored rules modify dictionary passwords in a way that mimics human behavior, increasing the chance of successful hits.

Nevertheless, concrete methods differ in speed, memory requirements, and the usability of the rulesets they produce. Affinity Propagation (AP) inherently selects cluster representatives and generates high-quality clusters that result in rules with relatively high success rates. However, the method suffers from poor time and space complexity, rendering it impractical for larger training dictionaries. Hierarchical Agglomerative Clustering (HAC) offers significantly better time complexity, but its memory requirements remain high due to the need to compute a complete distance matrix. With DBSCAN and MDBSCAN, the memory requirements can be minimized if the SymSpell fuzzy search algorithm (Garbe, 2012) is used instead. From all the examined methods, MDBSCAN, proposed by Li et al. (2022), led to the most effective rules, primarily thanks to its capability to "break down" the cluster of outlier passwords into smaller ones, allowing the creation of more effective rules.

The quality of the generated rules depends not only on the clusters produced but also on the selection of their representatives. As demonstrated by our experiments, the traditional Levenshtein Method used by Li et al. (2022) is not always optimal. Combining it with the substring-based approach we propose generally yields superior results. Our Combo Method achieved significantly better outcomes in most cases.

The strategy for the generation of rule commands is also crucial. By incorporating commands for case toggling, word rotations, reversals, and character overwrites, we achieved higher hit ratios than Li et al. (2022), not only in MDBSCAN Combo mode but also in the standard Levenshtein mode, and, unexpectedly, even with classic DBSCAN in most measurements. In terms of MDBSCAN, we achieved up to an 11.67%pt. improvement in hit ratio over the original method.

Naturally, the success rate of each attack highly depends on the training dictionary that serves to create mangling rules and on the dictionary that is used in the attack wordlist. We assume the best results can be achieved by training on dictionaries that have a similar nature to the attack's target.

Last but not least, we released RuleForge, an open-source clustering-based rule generator that serves not only as a proof-of-concept implementation of our enhancements but also as a tool for further password recovery research and a mangling-rule generator for actual cracking tasks. The release contains sources, documentation. and all password datasets that were used in this paper.

Looking ahead, we would like to evaluate the behavior of alternative distance metrics and other methods like Spectral Clustering (Jia et al., 2014) and their potential benefits to password-mangling rule creation. We also believe AP and HAC could be optimized for this specific use case. Moreover, it could also be helpful to conduct a deeper evaluation on a larger set of password dictionaries. Another potential lies in experimenting with hybrid methods that would combine multiple state-of-the-art approaches. With the spread of AI in recent years, we also believe deep learning and LLM-based approaches could show usability in mangling rule creation and evaluation.

References

Bakker, M., van der Jagt, R., 2010. GPU-based password cracking. Research project. University of Amsterdam, System and Network Engineering, Amsterdam. URL: `https://rp.os3.nl/2009-2010/p34/report.pdf`.

Bishop, M., V. Klein, D., 1995. Improving system security via proactive password checking. Computers & Security 14, 233–249. doi:`10.1016/0167-4048(95)00003-Q`.

Ciaramella, A., D'Arco, P., De Santis, A., Galdi, C., Tagliaferri, R., 2006. Neural network techniques for proactive password checking. IEEE Transactions on Dependable and Secure Computing 3, 327–339. doi:`10.1109/TDSC.2006.53`.

Damerau, F.J., 1964. A technique for computer detection and correction of spelling errors. Commun. ACM 7, 171–176.

Drdák, D., 2020. Automated Creation of Password Mangling Rules. Bachelor's thesis. Faculty of Information Technology. Brno University of Technology. Czechia.

Düermuth, M., Angelstorf, F., Castelluccia, C., Perito, D., Chaabane, A., 2015. OMEN: Faster password guessing using an ordered markov enumerator, in: Piessens, F., Caballero, J., Bielova, N. (Eds.), Engineering Secure Software and Systems, Springer International Publishing, Milan, Italy. pp. 119–132.

Ester, M., Kriegel, H.P., Sander, J., Xu, X., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), AAAI Press. p. 226–231.

Frey, B.J., Dueck, D., 2007. Clustering by passing messages between data points. Science 315, 972–976. doi:10.1126/science.1136800.

Garbe, W., 2012. SymSpell. URL: https://github.com/wolfgarbe/SymSpell. [Online; Accessed: 2024-04-16].

Hitaj, B., Gasti, P., Ateniese, G., Perez-Cruz, F., 2019. PassGAN: A deep learning approach for password guessing, in: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (Eds.), Applied Cryptography and Network Security, Springer International Publishing. pp. 217–237. doi:10.1007/978-3-030-21568-2_11.

Houshmand, S., Aggarwal, S., Flood, R., 2015. Next gen PCFG password cracking. IEEE Transactions on Information Forensics and Security 10, 1776–1791. doi:10.1109/TIFS.2015.2428671.

Hranický, R., Lištiak, F., Mikuš, D., Ryšavý, O., 2019. On practical aspects of PCFG password cracking, in: Foley, S.N. (Ed.), Data and Applications Security and Privacy XXXIII, Springer International Publishing. pp. 43–60.

Hranický, R., Zobal, L., Ryšavý, O., Kolář, D., Mikuš, D., 2020. Distributed PCFG password cracking, in: Chen, L., Li, N., Liang, K., Schneider, S. (Eds.), Computer Security – ESORICS 2020, Springer International Publishing. pp. 701–719.

Jackal, 1993. Cracker Jack, THE Unix Password Cracker [Read Me]. Doc's for Cracker Jack v 1.4. URL: http://justinakapaste.com/cracker-jack-the-unix-password-cracker-read-me/. [Online; Accessed: 2024-04-06].

Jia, H., Ding, S., Xu, X., Nie, R., 2014. The latest research progress on spectral clustering. Neural Computing and Applications 24, 1477–1486. doi:10.1007/s00521-013-1439-2.

Jiawei, H., Micheline, K., Jian, P., 2012. Data mining: concepts and techniques. 3 ed., Morgan Kaufmann series in data management systems. Waltham: Morgan Kaufmann.

Kacherginsky, P., 2013. Password Analysis and Cracking Kit (PACK). URL: https://github.com/iphelix/pack/. [Online; Accessed: 2024-04-06].

Kanta, A., Coisel, I., Scanlon, M., 2022. A novel dictionary generation methodology for contextual-based password cracking. IEEE Access 10, 59178–59188. doi:10.1109/ACCESS.2022.3179701.

Kanta, A., Coisel, I., Scanlon, M., 2023. Harder, better, faster, stronger: Optimising the performance of context-based password cracking dictionaries. Forensic Science International: Digital Investigation 44, 301507. doi:10.1016/j.fsidi.2023.301507.

Kelley, P.G., Komanduri, S., Mazurek, M.L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L.F., Lopez, J., 2012. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms, in: 2012 IEEE Symposium on Security and Privacy, IEEE. pp. 523–537. doi:10.1109/SP.2012.38.

Levenshtein, V.I., 1966. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710.

Li, S., Wang, Z., Zhang, R., Wu, C., Luo, H., 2022. Mangling rules generation with density-based clustering for password guessing. IEEE Transactions on Dependable and Secure Computing 20, 3588–3600. doi:10.1109/TDSC.2022.3217002.

Marechal, S., 2012. Automatic mangling rules generation. Passwordsˆ12 conference, Department of Informatics, University of Oslo, Norway .

Marechal, S., 2022. Rulesfinder. URL: https://github.com/synacktiv/rulesfinder. Synacktiv, [Online; Accessed: 2024-04-06].

Melicher, W., Ur, B., Segreti, S.M., Komanduri, S., Bauer, L., Christin, N., Cranor, L.F., 2016. Fast, lean, and accurate: Modeling password guessability using neural networks, in: 25th USENIX Security Symposium (USENIX Security 16), pp. 175–191.

Muffett, A., 1996. Crack version v5.0 user manual. URL: https://www.techsolvency.com/pub/src/crack-5.0a/c50a/manual.html.

Munshi, A., 2009. The OpenCL specification, in: Proceedings of the 21st IEEE Hot Chips Symposium (HCS), Standford, CA, USA. pp. 1–314. doi:10.1109/HOTCHIPS.2009.7478342.

Narayanan, A., Shmatikov, V., 2005. Fast dictionary attacks on passwords using time-space tradeoff, in: Proceedings of the 12th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA. pp. 364–372. doi:10.1145/1102120.1102168.

NVIDIA Corporation, 2019. CUDA Toolkit documentation v10.1.243.

Peslyak, A., 2015. When was John created? (john-users Openwall mailing list). URL: https://www.openwall.com/lists/john-users/2015/09/10/4. [Online; Accessed: 2024-04-06].

Peslyak, A., 2017. John the Ripper rules. URL: https://www.openwall.com/john/doc/RULES.shtml. [Online; Accessed: 2024-04-06].

Peslyak, A., 2019. John the Ripper password cracker changelog, version 1.9, Openwall. URL: https://www.openwall.com/john/doc/CHANGES.shtml. [Online; Accessed: 2024-04-06].

Proctor, R.W., Lien, M.C., Vu, K.P.L., Schultz, E.E., Salvendy, G., 2002. Improving computer security for authentication of users: Influence of proactive password restrictions. Behavior Research Methods, Instruments, & Computers 34, 163–169.

Steube, J., 2017a. Generate Rules (generate-rules.c), the hashcat-utils GitHub repository. URL: https://github.com/hashcat/hashcat-utils/blob/master/src/generate-rules.c. [Online; Accessed: 2024-04-06].

Steube, J., 2017b. hashcat-legacy. URL: https://hashcat.net/wiki/doku.php?id=hashcat-legacy. [Online; Accessed: 2020-11-16].

Steube, J., 2017c. oclHashcat. URL: https://hashcat.net/wiki/doku.php?id=oclhashcat_old. [Online; Accessed: 2024-04-06].

Steube, J., 2020. Hashcat description. URL: https://hashcat.net/wiki/doku.php?id=hashcat. [Online; Accessed: 2021-03-02].

Steube, J., 2024. Hashcat: Rule-Based Attack. URL: https://hashcat.net/wiki/doku.php?id=rule_based_attack. [Online; Accessed: 2024-04-06].

Veras, R., Collins, C., Thorpe, J., 2014. On semantic patterns of passwords and their security impact, in: Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium, pp. 386–401. doi:10.14722/ndss.2014.23103.

Veras, R., Collins, C., Thorpe, J., 2021. A large-scale analysis of the semantic password model and linguistic patterns in passwords. ACM Transactions on Privacy and Security (TOPS) 24, 1–21.

Vu, K.P.L., Proctor, R.W., Bhargav-Spantzel, A., Tai, B.L.B., Cook, J., Schultz, E.E., 2007. Improving password security and memorability to protect personal and organizational information. International Journal of Human–Computer Studies 65, 744–757. doi:10.1016/j.ijhcs.2007.03.007.

Weir, M., Aggarwal, S., d. Medeiros, B., Glodek, B., 2009. Password cracking using probabilistic context-free grammars, in: Proceedings of the 30th IEEE Symposium on Security and Privacy, Oakland, California, USA. pp. 391–405. doi:10.1109/SP.2009.8.

Winkler, W.E., 1990. String comparator metrics and enhanced decision rules in the Fellegi–Sunter model of record linkage, in: Proceedings of the Survey Research Methods Section, American Statistical Association, pp. 354–359.

Xia, Z., Yi, P., Liu, Y., Jiang, B., Wang, W., Zhu, T., 2019. GENPass: A multi-source deep learning model for password guessing. IEEE Transactions on Multimedia 22, 1323–1332.

Zivadinovic, M., Milenkovic, I., Simic, D., 2016. Cash, hash or trash — hash function impact on system security, in: Jaško, O., Marinković, S. (Eds.), Proceedings of the 15th SymOrg International Symposium, ICT and Management section, pp. 788–791.