Forensic Science International: Digital Investigation 52 (2025) 301865

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS EU 2025 - Selected Papers from the 12th Annual Digital Forensics Research Conference Europe

Beyond the dictionary attack: Enhancing password cracking efficiency through machine learning-induced mangling rules

Radek Hranický [*], Lucia Šírová [**], Viktor Rucký [***]

Brno University of Technology, Faculty of Information Technology, Božetěchova 2/1, Brno, 612 00, Czech Republic

ARTICLE INFO

Keywords:
Password
Rules
John the Ripper
Hashcat
Clustering

ABSTRACT

In the realm of digital forensics, password recovery is a critical task, with dictionary attacks representing one of the oldest yet most effective methods. To increase the attack power, developers of cracking tools have introduced password-mangling rules that apply modifications to the dictionary entries such as 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 research lacks a deeper comparison and evaluation of the individual methods and their implications. We present RuleForge, a machine learning-based mangling-rule generator that leverages four clustering techniques and 19 commands with configurable priorities. Key innovations include an extended command set, advanced cluster representative selection, and various performance optimizations. We conduct extensive experiments on real-world datasets, evaluating clustering-based methods in terms of time, memory use, and hit ratios. Additionally, we compare RuleForge to existing rule-creation tools, password-cracking solutions, and popular existing rulesets. Our solution with an improved MDBSCAN clustering method achieves up to an 11.67%pt. Higher hit ratio than the original method and also outperformed the best yet-known state-of-the-art solutions for automated rule creation.

1. Introduction

Since the advent of password authentication in computing, password cracking has been a significant area of focus. This technique is used not only by malicious hackers but also by the “good guys” such as law enforcement, cyber defense organizations, penetration testers, security analysts to measure password strength (Proctor et al., 2002; Vu et al., 2007), or individuals recovering lost credentials. In digital forensics, recovering passwords is crucial for accessing encrypted evidence, making it an essential step in the investigative process.

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. 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. 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).

Despite advances in 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, 2020; Drdák, 2020; Li et al., 2022). Recent approaches leverage machine learning, particularly clustering (Drdák, 2020; Li et al., 2022), with MDBSCAN (Li et al., 2022) being the latest method proposed. While these approaches show significant potential, they lack a comprehensive comparison of clustering methods and rule-creation strategies, leaving room for further research and improvements.

[*] Corresponding author.
[**] Corresponding author.
[***] Corresponding author.
E-mail addresses: hranicky@fit.vut.cz (R. Hranický), xsirov01@stud.fit.vutbr.cz (L. Šírová), xrucky01@stud.fit.vutbr.cz (V. Rucký).

https://doi.org/10.1016/j.fsidi.2025.301865

Available online 24 March 2025

1.1. Contributions

Firstly, we conduct a comprehensive evaluation of ruleset creation with four clustering methods, assessing generation time, memory usage, and hit ratios on real-world datasets. Secondly, we introduce optimizations to the rule creation process, including an extended rule command set and advanced techniques for selecting cluster representatives, improving flexibility and efficiency. Third, we test these optimizations on MDBSCAN and benchmark our solution against the state-of-the-art approach of Li et al. (2022), achieving an improvement of up to 11.67 % points in the hit ratio. Next, we compare our approach with other rule-creation and password-guessing tools, achieving the highest average hit ratio among all studied methods. Lastly, we compare the hit rate with popular widely used rulesets, outperforming nearly all of them. We also analyze the rules created and the strength of the recovered passwords. Our contributions are demonstrated through RuleForge, a clustering-based mangling-rule generator we developed as both a proof-of-concept and a practical tool for password research and real-world password cracking. Its flexibility allows to create of context-specific rulesets to match the unique characteristics of each investigation.

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), 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 Muffett 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 appending. 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 "Solar Designer" Peslyak created the *John the Ripper* (JtR) tool as a replacement for the popular Cracker Jack 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, 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. Hashcat had a native support for password-mangling rules and adopted the syntax and semantics from JtR.

The release of NVIDIA CUDA and OpenCL started a revolution in the password cracking. Developers quickly reacted by adding GPU support to their tools (Steube, 2020; Peslyak, 2019). Steube was no exception and, in 2010, released cudaHashcat and oclHashcat, 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. Hashcat also introduced new 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" (CMIYC) contests.[1] 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, 2020; Drdák, 2020; Li et al., 2022).

The *hashcat-utils* repository includes *generate-rules.c*, a simple utility by Jens Steube that generates random password-mangling rules based on a time-based or user-defined seed. 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 serves as a baseline for comparing more advanced techniques. The algorithm was later integrated directly into hashcat (Steube, 2020).

[1] https://contest.korelogic.com/.

Marechal (2012) proposed generating mangling rules by applying handpicked or randomly generated initial rules to a wordlist, producing mangled passwords. Their algorithm identifies the largest common substring among the results and derives append/prepend operations to recreate it from the remaining passwords. These operations represented rules that were then ranked according to the number of passwords created. Marechal's proof-of-concept tool, *rulesfinder*, remains actively maintained. Although the approach is working, its major drawback is the need for an existing set of rules.

Peter "iphelix" Kacherginsky (2013) introduced a novel technique and a proof-of-concept tool called Rulegen within the Password Analysis and Cracking Kit (PACK). It 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, 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).

Between 2019 and 2020, Drdák and Hranický (Drdák, 2020) explored using machine learning for automated rule creation by clustering a training dictionary based on password similarity. From each cluster, a password was chosen as a representative. Mangling rules were then created to describe necessary modifications for transforming the representative to the remaining passwords in the cluster. Using Affinity Propagation (AP) (Frey and Dueck, 2007), Drdák developed a proof-of-concept with promising results published in his bachelor's thesis (Drdák, 2020). While the general idea has been later proven useable by other researchers (Li et al., 2022), Drdák's study had its limitations. Firstly, Drdák tested only a single clustering method. The second 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) (Han 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 (Han 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.
6. Analyze the hit ratio of the generated rules and compare it with existing popular rulesets.

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 (Han 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 RuleForge rule generation process 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

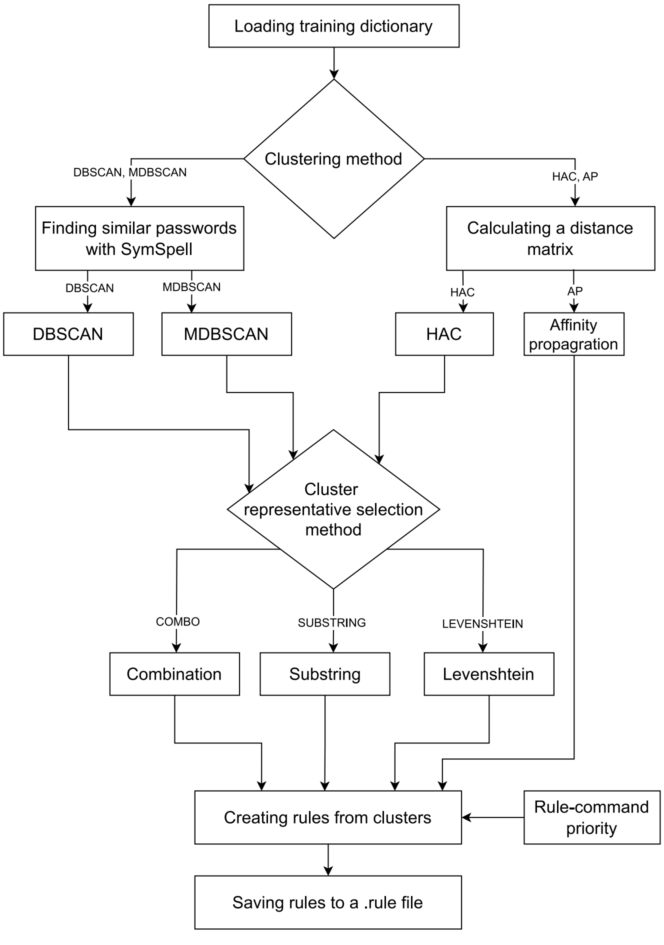

Loading training dictionary
Clustering method
DBSCAN, MDBSCAN
HAC, AP
Finding similar passwords with SymSpell
Calculating a distance matrix
DBSCAN
MDBSCAN
HAC
AP
DBSCAN
MDBSCAN
HAC
Affinity propagration
Cluster representative selection method
COMBO
SUBSTRING
LEVENSHTEIN
Combination
Substring
Levenshtein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

Fig. 1. Rule generation process.

Levenshtein distance metric (Levenshtein, 1966). Password clusters are then created using the selected method.

Next, we select strings to be considered representatives of their respective cluster. The reason is to create rules based on comparing passwords within a cluster with 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 techniques from 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-command 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 rules 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.

3.2.1. *AP*

Affinity Propagation treats all objects as potential exemplars, exchanging messages to identify high-quality exemplars and clusters (Frey and Dueck, 2007). Key parameters are *damping*, which is the extent to which the current value is maintained relative to incoming values, and *convergence_iter*, representing how many iterations without change stop the clustering. In our experiments, *convergence_iter* is 15, while *damping* is 0.7, as these settings produced the best results.

3.2.2. *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 *distance_threshold* (Han et al., 2012). Our setup uses *distance_threshold* of 3, which has been experimentally verified to be the most effective for our use case.

3.2.3. *DBSCAN*

Density-Based Spatial Clustering of Applications with Noise identifies core points—objects that have at least *MinPts* neighbors within ϵ distance. Each core point initially forms a cluster with itself and then expanding by including neighboring objects. The result is a set of clusters and a set of non-clustered noise objects. We set ϵ as 1 and *MinPts* to 3. In our experience, higher values of ϵ lead the algorithm to output a single cluster containing the vast majority of passwords. Higher values of *MinPts* categorize the majority of passwords as noise.

3.2.4. *MDBSCAN*

Modified MDBSCAN (Li et al., 2022) addresses DBSCAN’s tendency to form one large cluster when clustering passwords by introducing a truncation metric. MDBSCAN’s parameters are ϵ_1, ϵ_2, and *MinPts*, where ϵ_1 and *MinPts* are equivalent to DBSCAN’s. An object is only added to a cluster if its Jaro–Winkler distance (Winkler, 1990) to the initial point of the cluster is less than or equal to ϵ_2. The truncation allows a higher ϵ_1 by breaking up the large cluster, reducing noise. We set ϵ_1 to 2, ϵ_2 to 0.25, and *MinPts* to 3. Setting ϵ_1 above 2 leads to enormously large clusters. High ϵ_2 leads to the creation of too many useless single-password clusters, whereas higher values produce too large clusters. *MinPts* configuration has the same impact 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 representative for each cluster and search for possible transformations to the remaining passwords in the cluster.

3.3.1. *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, 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 useable in very specific cases. What we want are rules that have general use.

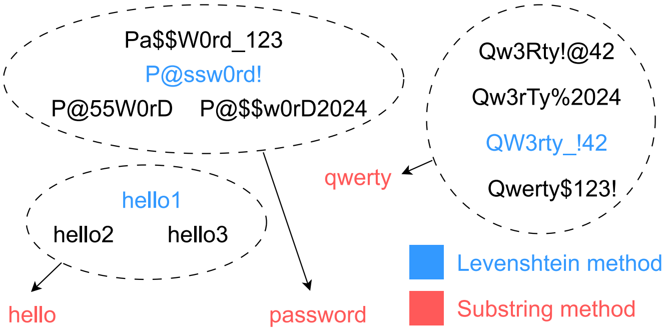


Fig. 2. A visual comparison of cluster representative selection methods.

Table 1
Rule commands implemented in RuleForge, applied on “Password” (bold are newly added).

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

3.3.2. *Substring Method*

To overcome the obstacles of the previous approach, we invented 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* → @ or *s* → $. 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 created by this method may not always be an actual existing password from the cluster.

3.3.3. *Combo Method*

While the Substring Method allows the creation of more generally useable rules, the sole method is not extremely powerful. Therefore, we propose a third option that combines the previous two methods, leading to the best experimental results from all (See Section 4.). 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. Measuring edit distance helps find specific rule commands that, when used on passwords, make the edit distance smaller. A command that decreases the edit distance is deemed appropriate and incorporated into the generated rule. Multiple commands (such as `sXY` and `oNX`) may achieve identical modifications in certain instances. Therefore, RuleForge introduces a rule-command priority system, specifying which commands it prioritizes. The configuration can be specified in a priority file, where one can determine which rule commands RuleForge should utilize and in which priority. The generator proposed by Li et al. (2022) supports 14 different rule commands. With RuleForge, we expanded this number to 19. The commands supported by RuleForge are displayed in Table 1. Other Hashcat rule commands that have not yet been implemented are considered for future work. This approach of using rule-command priority allows the exploration of different 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 for its popularity, common knowledge among researchers, extensive data-analysis support. 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[2] 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 GitHub[3] under the MIT License.

4. Experimental results

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. Next, we compare the hit ratio of RuleForge with other techniques and state-of-the-art tools. Finaly, we compare hit rates with popular rulesets. In the experiments, we use various password dictionaries. Table A.5 describes each of them. All are also available on our GitHub repository.[3] Note, for some experiments, we use abbreviations (from the “Ab.” column) instead of full names.

4.1. *Benchmarking of clustering and rule creation*

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 the four examined methods. 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 1000 to 100,000 passwords from the RockYou dictionary. 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.

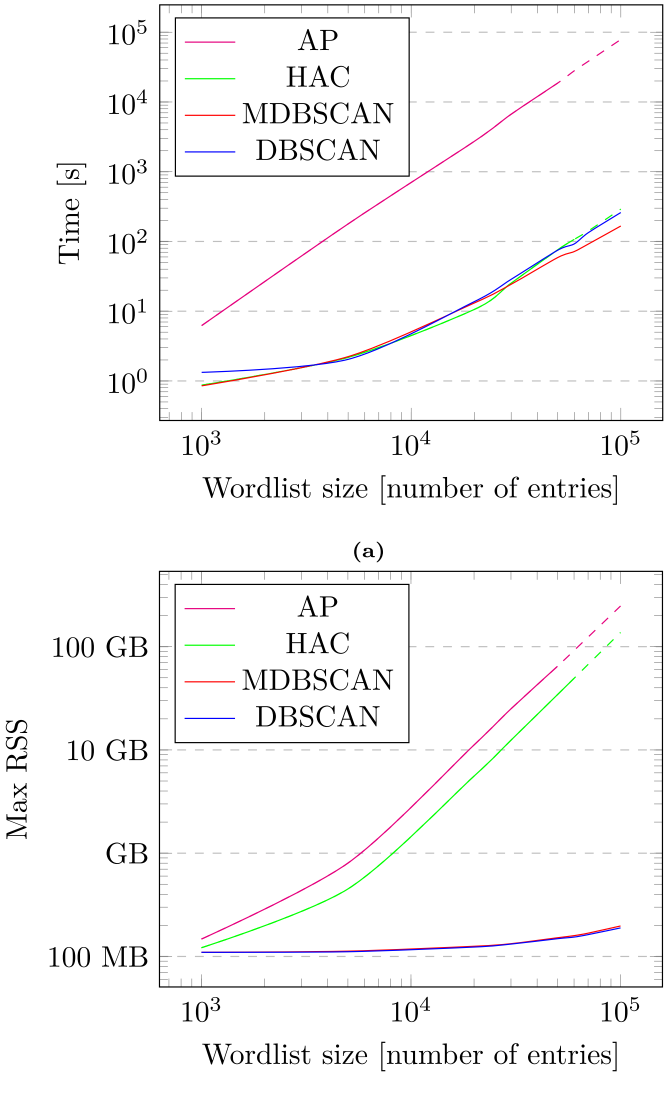
DBSCAN, MDBSCAN, and HAC demonstrate comparable and decent performance, all processing dictionaries with 100,000 passwords in under 5 min. In contrast, AP exhibits significantly poorer performance, requiring about 21 h to handle the same workload.

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

[2] https://scikit-learn.org/.

[3] https://github.com/nesfit/RuleForge/.



(a)

(b)

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

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 useable 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 6.2.6 in plaintext mode. The target "hashlist" was RockYou-75 (See Table A.5.), for which, we calculated the number of hits. To maintain fair conditions for all methods, we used the best (i.e. first) 1000 rules generated by each method.

Table 2 shows the hit ratios. On average, MDBSCAN produced rules with the best hit ratios. The second best-achiving method was AP which, for r65+en, it 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. The dictionaries were the same as in the previous experiment. Likewise, we used Hashcat 6.2.6 and the first 1000 generated rules.

Table 3 describes the hit ratio of attacks on RockYou-75 and Table 4 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 demonstrate that there is no optimal method for all cases.

4.4. Comparison of rule-creation methods

In this 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." Using Hashcat 6.2.6, we 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 A.5. 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 as they would require a minimum of 461 GB memory, 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

Table 2
Attacks on rockyou-75-m.

Rules		Hit ratio			
t[a]	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 %

[a] Training dictionary.

Table 3
MDBSCAN RF vs. Li, rockyou-75-m.

Rules		Hit ratio			
t[a]	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 %

[a] Training dictionary.

Table 4
MDBSCAN RF vs. Li, Xato-Net-100k.

Rules		Hit ratio			
t[a]	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 %

[a] Training dictionary.

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 numbers 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 indicates the number of rules (top) and corresponding guesses (bottom), calculated as the dictionary size multiplied by the rule count. The vertical axis displays the hit ratio.

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 guess counts 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).

Our improvements are best illustrated by the difference between MDBSCAN-RF-combo (the solid black line) and MDBSCAN Li et al. (the solid red line). The biggest change was a 6.68%pt. improvement at 6400 rules on Xato and a 11.67%pt. improvement at 18,000 rules on phpbb-m. Those are marked by black-lined arrowed segments.

We also examined the strength of the passwords recovered by the MDBSCAN-RF-Combo ruleset of 29k rules using zxcvbn,[4] which calculates the strength as an estimated number of guesses required to crack the password.

From Xato-net-100k, we recovered 93.63 % of all passwords. For those with strength under e^{14}, we recovered the majority: 95.08 %. From passwords with strength higher or equal e^{14} and lower than e^{18}, we cracked 65.70 %. About 1 % had strength over or equal to e^{18}, where the success rate was 21.83 %.

From PhpBB, we recovered 62.14 % of all passwords. For strengths under e^{14}, we cracked 85.16 % of passwords. For strengths higher or equal e^{14} and lower than e^{18}, the success was 45.63 %. About 19.89 % had strength over or equal to e^{18}, where the success was 10.68 %.

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 created 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.

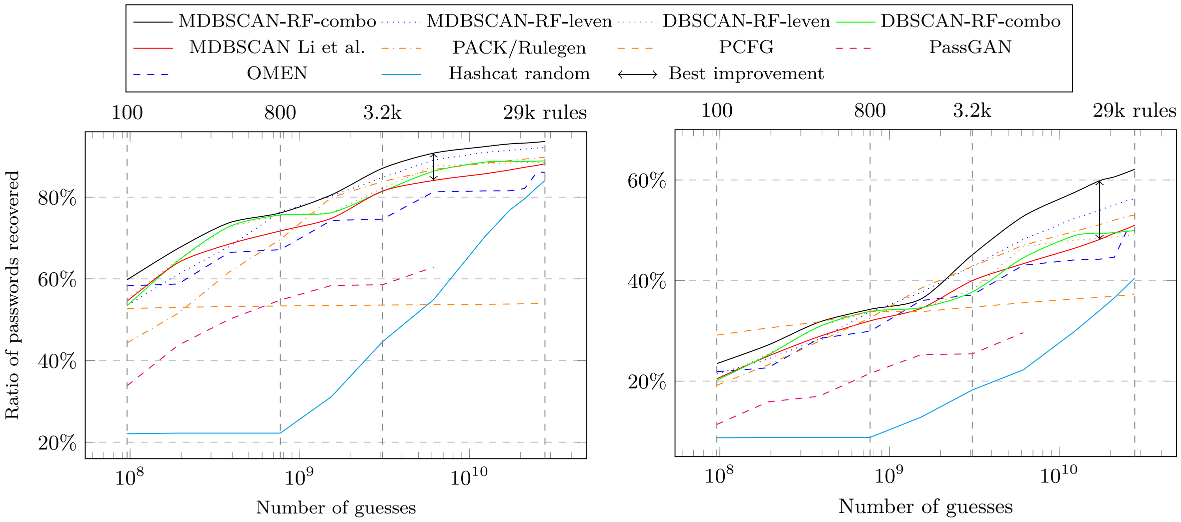
4.5. Comparison with popular rulesets

Lastly, we compared MDBSCAN and DBSCAN against widely used popular password cracking rulesets: d3ad0ne and Insidepro-PasswordsPro from the Hashcat repository, OneRuleToRuleThemStill[5] from the stealthspolit repository, and Unicorn30kGenerated[6] from Unic0rn28. The settings are the same as in the previous experiment. Hashcat random rules again serve as a baseline.

The resulting hit ratios in Fig. 5 show that RuleForge's MDBSCAN in the combo mode outperformed all except OneRuleToRuleThemStill. This high-quality ruleset produced better results mainly between 800 and 3.2k rules. For lower counts, RuleForge was slightly better. From 6.4k rules, the two were comparable, with the ruleset having less than 1%pt. more success. It is important to emphasize that our objective was not to develop a universally optimal ruleset. Instead, we aimed to create a tool for automated rule generation that can be tailored to meet the specific requirements of individual investigations.

The first 1k rules of MDBSCAN-combo consisted exclusively of single-rule commands like appending, prepending, overwriting and inserting a single character, but also word truncation, and other commands. Case toggles were present on 224 lines of the ruleset. Generally,

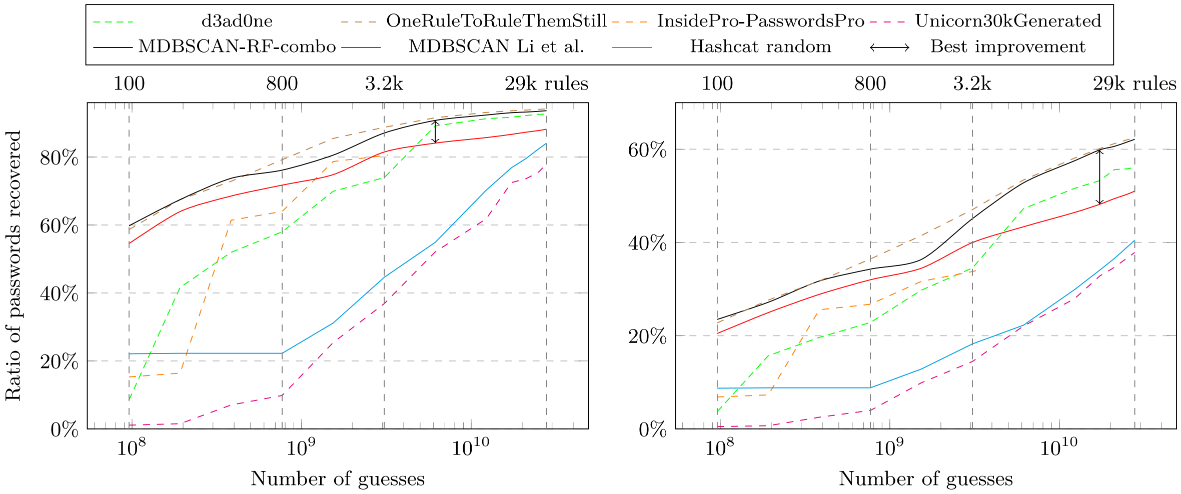
[4] https://github.com/dropbox/zxcvbn.
[5] https://github.com/stealthsploit/OneRuleToRuleThemStill.
[6] https://github.com/Unic0rn28/hashcat-rules.



(a) Target: Xato-net-100k

(b) Target: phpbb-m

Fig. 4. Hit rate comparison with other methods (Training: RockYou960, Attack: RockYou960).



(a) Target: Xato-net-100k

(b) Target: phpbb-m

Fig. 5. Hit rate comparison with popular rulesets (Training: RockYou960, Attack: RockYou960).

compared with the other rulesets, ours had significantly more insertion and truncation commands. The generated rulesets are also available in the RuleForge repository.

5. Conclusion

Our research demonstrates the significant potential of the clustering-based generation of password-mangling rules in enhancing dictionary attacks, often outperforming state-of-the-art guessing methods. AP produces high-quality clusters, but has prohibitive time and space complexity, which limits scalability. HAC improves time efficiency, but still requires substantial memory. DBSCAN and MDBSCAN reduce memory demands when paired with SymSpell. MDBSCAN, noted for splitting large password clusters into smaller, more effective ones, achieved the best results among the tested methods.

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 rule generation strategy 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. With MDBSCAN, we achieved up to an 11.67%pt. improvement in hit ratio over the original method.

The attack's success depends on the training and attack wordlists. Using dictionaries similar to the nature of the target is likely to yield the best results. In a digital forensic lab, the proposed method can be utilized to generate context-specific rulesets tailored to the unique characteristics of each investigation. Forensic analysts frequently build profiles of suspects based on known passwords, nationalities, interests, and typical password modification patterns. Automated rule creation simplifies crafting case-specific rulesets by training on relevant wordlists and previously-used passwords. This dramatically reduces the overhead of manual ruleset development, enhancing investigative efficiency. This enables forensic practitioners to concentrate on higher-level analytical tasks without sacrificing performance.

Last but not least, we released RuleForge, an open-source clustering-based rule generator, as both a proof-of-concept and a practical tool for password recovery research. The release includes source code, documentation, all referenced password datasets, and rulesets generated for experiments 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 that AP and HAC could be optimized in terms of memory requirements. The strength analysis of the cracked passwords showed that a ruleset trained on RockYou-960 worked well with easy to advanced passwords, but had difficulties with very strong ones. In the future, we, therefore, plan to train on stronger passwords to create more sophisticated rules and examine their success. With the spread of AI, we also intend to explore transformer-based models for rule creation and evaluation. Additionally, we are currently developing a much faster, compiled, optimized and GPU-accelerated version of RuleForge.

Acknowledgements

The research presented in this paper has been supported by the “Smart Information Technology for a Resilient Society” project, no. FIT-S-23-8209, granted by Brno University of Technology.

Appendix A. Used Dictionaries

Table A.5 provides an overview of all password dictionaries used in our experiments, including their abbreviations, names, password counts, and descriptions.

Table A.5
Password dictionaries for experimental evaluation

Ab.	Name	Passwords	Description
tl	tuscl-m	37,006	Tuscl leak (ASCII, ≤ 10 ch.)
r65	rockyou-65-m	29,596	RockYou subsample (ASCII, ≤ 10 ch.)
ms	myspace-m	30,000	MySpace leak (ASCII, ≤ 10 ch.)
dw	darkweb2017-top10k-m	9999	Darkweb subsample (ASCII, ≤ 10 ch.)
tm	10-million-list-top-10000	9999	9999 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	1315	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

References

Bishop, M., Klein, D.V., 1995. Improving system security via proactive password checking. Comput. Secur. 14, 233–249.

Ciaramella, A., D’Arco, P., De Santis, A., Galdi, C., Tagliaferri, R., 2006. Neural network techniques for proactive password checking. IEEE Trans. Dependable Secure Comput. 3, 327–339.

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. FIT, 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: Engineering Secure Software and Systems. Springer, 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 2nd KDD-96. AAAI Press, pp. 226–231.

Frey, B.J., Dueck, D., 2007. Clustering by passing messages between data points. Science 315, 972–976.

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

Han, J., Micheline, K., Jian, P., 2012. Data mining: concepts and techniques. Morgan Kaufmann Series in Data Management Systems, 3 ed. Morgan Kaufmann, Waltham.

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, pp. 217–237.

Houshmand, S., Aggarwal, S., Flood, R., 2015. Next gen PCFG password cracking. IEEE Trans. Inf. Forensics Secur. 10, 1776–1791.

Hranický, R., Lištiak, F., Mikuš, D., Ryšavý, O., 2019. On practical aspects of PCFG password cracking. In: Data and Applications Security and Privacy XXXIII. Springer, pp. 43–60.

Hranický, R., Zobal, L., Ryšavý, O., Kolář, D., Mikuš, D., 2020. Distributed PCFG password cracking. In: Computer Security – ESORICS 2020. Springer, pp. 701–719.

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

Kacherginsky, P., 2013. Password analysis and cracking Kit. URL: https://github.com/iphelix/pack/. Online; Acc.: 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.

Kanta, A., Coisel, I., Scanlon, M., 2023. Harder, better, faster, stronger: optimising the performance of context-based password cracking dictionaries. FSI: Digit. Invest. 44, 301507.

Levenshtein, V.I., 1966. Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. 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 Trans. Dependable Secure Comput. 20, 3588–3600.

Marechal, S., 2012. Automatic mangling rules generation. Passwordsˆ12 Conference. University of Oslo, Norway.

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.

Narayanan, A., Shmatikov, V., 2005. Fast dictionary attacks on passwords using time-space tradeoff. In: Proceedings of the 12th ACM CCS, New York, NY, USA, pp. 364–372.

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; Acc: 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; Acessed: 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. Behav. Res. Methods Instrum. Comput. 34, 163–169.

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 NDSS Symposium, pp. 386–401.

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. Int. J. Hum. Comput. Stud. 65, 744–757.

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, CA, USA, pp. 391–405.

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 Trans. Multimed. 22, 1323–1332.