OneRuleToFindThem: Efficient Automated Generation of Password Cracking Rules

XXX YYY ZZZ
AAA *BBB* *CCC*

Abstract

Password cracking tools such as Hashcat support the use of rules that transform a dictionary of words, such as common English words and previously-cracked passwords, into new candidate guesses for hashed passwords. Rules are necessary to achieve high cracking ratios, however, they are difficult and time-consuming to build by hand. We have developed an algorithm and implementation that automatically finds successful rules via the combinatorial generation of rules and empirical observation of how often each generated rule transforms a dictionary word to a target password.

Our algorithm includes numerous performance and logical optimizations to avoid the numerous pitfalls that would occur if a naïve brute-force technique were used. In this paper, we explain our algorithm in detail and experimentally compare the performance of its outputs to existing rule sets constructed via various approaches ranging from fully-manual to fully-automated like our own.

We show that our approach is completely automated and achieves comparable cracking performance to other top rule sets while also generating rules that do not exist in other rule sets. This makes cracking attempts using our rules mostly complementary to cracking attempts with other rule sets. Top performance is achieved by combining our generated rules with other rule sets.

1 Introduction

Although not the only form of authentication, the most common authentication for applications continues to be passwords. Properly implemented password authentication and very strong passwords, such as long passwords composed of random characters, is generally effective. However, it is well-known that a significant proportion of people elect to use passwords that combine a word with a few numbers or special characters as required by the software. These passwords often have predictable patterns and evolve in predictable ways over time as a user is forced to change their password, often resulting in simple modifications resulting in a new password with a high degree of similarity to the original [2]. While this approach might make passwords more amenable to memorization, it significantly weakens them in the face of a password cracking attack.

A common approach to cracking passwords is through the use of Hashcat [4], utilizing the technique of hashing candidate passwords in a wordlist and checking if the hash matches one found in list of password hashes. In order to avoid pre-generation of a massive exhaustive wordlist, Hashcat supports the use of 'rules,' which perform transformations on a list of words such as dictionary words and previously-cracked passwords. The resulting transformed passwords are then hashed and the same checks are made. Example rules include reverse (r), append character ($X), and replace (sXY, replace X with Y).

To produce the most guesses and therefore crack the most passwords, one must either have a large wordlist or a large number of rules (or both). However, because most passwords are not generated randomly, some rules will be (sometimes dramatically) more effective than others. Because each additional rule in the list increases the time the cracking process takes to complete, an attacker is incentivized to minimize the number of rules (and wordlist size) while maximizing the percent of hashes that are cracked. The attacker wishes to use only the most effective rules.

Researching how to improve this popular method of cracking is merited on the grounds that it can inform good password policy. Many organizations enforce 'password strength' through length and character diversity requirements, which may provide a false sense of security. If a user meets these requirements simply by appending '123!' to a weak password, that password hash (if it is leaked) might be easily cracked with a common rule. These extra characters do not provide significant protection to a user. If we can find effective password cracking rules, we can create effective password strength requirements by ensuring the rules fail on the user's chosen password. Research into improving cracking efficiency can also potentially help those who have forgotten their own pass-

word and wish to recover it.

Many effective rules already exist and some are distributed with the Hashcat software, such as the sizable 'dive' list with about 99k rules. Various approaches have been taken to produce these lists. Some creators manually curate rules, which is a time-consuming process, while others have tried various algorithmic and automated approaches. Often, existing rule sets are aggregated to various degrees to produce a 'super rule,' such as OneRuleToRuleThemAll [11] and also some of the highly-effective 'Pantagrule' lists of rules [10]. The purpose of this paper is to detail a novel fully-automated approach to rule generation, based on an iterative rule accumulation and scoring procedure. We compare the performance of our generated rule sets to existing publicly-available rules.

The rest of this paper is organized as follows. First, we review related work (Section 2). Next, we describe our algorithm in Section 3. We do this in three parts. First, we show a simple brute-force procedure, then we explain optimizations we made to increase its effectiveness, followed by optimizations we made to increase its performance in terms of time and memory. Then we explain our experimental methodology (Section 4) followed by our results (Section 5) and future work (Section 6).

2 Related Work

Several automated rule-generation techniques are based on the PACK toolkit [7], a collection of tools designed for analyzing password lists to detect masks, rules, character sets, and various other password characteristics that can produce results (rules and masks) designed to work with Hashcat. The 'nsa-rules' analysis by NSAKEY [8] and the rules it generates take advantage of this toolkit, as well as the more effective Pantagrule rules [10].

Pantagrule rule lists were generated using PACK's Levenshtein reverse path algorithm to produce rules which were then sorted by the frequency at which they were generated by PACK. This is similar to the approach NSAKEY took but Pantagrule used a larger set of base passwords to generate rules, which although initially public is now inaccessible. To further optimize the rules, Pantagrule ran the top generated rules against the Pwned Passwords NTLM list [6] using the RockYou wordlist. Ineffective rules were discarded. Several rule lists are created from rules generated by various subsets of the seed data (top passwords, random passwords, and a hybrid of the two).

The 'one.rule' Pantagrule rule list builds upon the OneRuleToRuleThemAll (ORTRTA) rule list [11], which was created by concatenating the top 25% of rules from various other rule lists. 'one.rule' appends top performing Pantagrule hybrid rules to ORTRTA and truncates the list to the size of the 'dive' rules in order to make comparisons against a commonly used rule list of the same size. ORTRTA exceeds the performance of dive on its own, both in total % of passwords cracked and in cracking efficiency on the Lifeboat data dump. Pantagrule's one.rule also compares favorably in total number of passwords cracked against dive at the same total number of rules and against ORTRTA as a superset of its rules, however it is less efficient than ORTRTRA as a consequence of containing significantly more total passwords. Pantagrule suggests that their 'one.rule' performs better than other known lists the size of dive and they recommend it as a first list to try when cracking hashes.

In addition to traditional rule-based approaches to guessing passwords, some techniques have been developed that attempt to avoid this entirely. PassGAN [5] is an approach that attempts to replace rule-based password guessing with a technique based on deep learning and generative adversarial networks (GANs). A neural network was trained to determine password characteristics and structures without making any assumptions about these. Like our approach, PassGAN makes use of part of the RockYou dataset and trains on it. They then test their results against both RockYou (with training data removed) and a leak of LinkedIn passwords. Their results show that the PassGAN approach is able to match 34.6% of passwords in the RockYou dataset and 34.2% in the LinkedIn dataset. While a typical rule-based attack has the disadvantage of being able to exhaust guesses once all rules have been applied to all initial passwords, PassGAN can generate guesses effectively forever. So while PassGAN can in theory eventually guess more passwords than any other approach, it needs to generate significantly more passwords to do this and can require up to 10x more guesses to reach the same number of matched passwords as competitors. However, PassGAN matches some passwords not matched by any password rule in the rule sets they compared against.

Another approach [9] attempts to leverage representation learning techniques to discover a representation of password distributions. This technique models password representation through a GAN instance and a Wasserstein Auto-Encoder (WAE) instance and two password guessing frameworks are proposed; CPG and DPG. The model produced by this approach improves on PassGAN against the RockYou test set, cracking 51.8% of passwords in the same number of guesses (5×10^{10}) it took PassGAN to crack 34%. Like PassGAN, the CPG and DPG frameworks guess some passwords that are not cracked by other approaches and DPG allows a guessing attack to focus on unique and otherwise ignored modalities of the target passwords.

3 Algorithm

Our rule generator requires two inputs: a set of rule primitives that will be combined to form complex rules, and a set of target passwords such as the RockYou list. We implement an efficient version of what is essentially a brute-force procedure. We first describe the brute-force procedure and then describe our optimizations.

3.1 Brute-force procedure

Given each initial target password (e.g., from Rockyou), we apply every primitive rule to the password to generate new passwords. For example, the primitive Hashcat rule 'r' (reverse) applied to the initial target password '123456' results in password '654321.' We use a primitive rule set consisting of elementary operations such as reverse ('r'), remove last character (']'), delete all 's' characters ('@s'), and so on, totaling nearly 400 primitive rules. The selected password is subjected to every primitive, resulting in about 400 new passwords. For each resulting password (such as '654321'), we check if it is one of our targets from our initial list of targets (e.g., Rockyou). If it is, we boost the score of the rule that was applied. In the end, we have a list of rules with scores indicating which rules were most successful. Initially we simply boosted the score by 1 for each 'hit' but we later adopted an approach that increases the score more for passwords that are harder to hit, discussed in more detail later.

After that initial step of applying rules to a single password, we proceed to choose another password and apply all primitive rules to it, boosting the scores of rules that transform the password to a known target password. Then naïve brute-force approach would choose a new password to try either randomly or sequentially from a list of possible candidates but ultimately we choose the next password according to an ordering of all candidates by 'individual password strength,' with weaker passwords chosen earlier; details are given below.

Each password that is generated from applying primitive rules becomes a potential candidate itself, unless it is already known from the initial target set. For example, if the rule 'r' is applied to 'foobar,' producing 'raboof,' and 'raboof' is not already known from the target set, it becomes a candidate for selection. We record the history of rules that have already been applied, in this case just 'r.' When 'raboof' is eventually selected as the next password to try, each primitive rule is appended to its rule history, producing complex rules 'r]' 'r @s' and so on. If ']' applied to 'raboof,' which produces 'raboo,' is a target, then we boost the score of the complex rule 'r].' We note that the initial password 'foobar' (pulled from RockYou) was transformed to 'raboo' using complex rule 'r]' and 'raboo' is a target (in this example, though in reality it is not a member of RockYou). Thus, our procedure has discovered a successful rule that should be utilized in password cracking.

In summary, the brute-force procedure begins with an initial list of target passwords and puts them into a candidate set, picks a single candidate password at a time and applies all primitive rules, and boosts the scores of any rules that ultimately produced a password found in the initial list of targets. Each password generated from applying rules goes into the candidate set if it is not already in there, and the sequence of primitive rules that generated it is associated with the password.

Algorithm 1 Brute-force procedure, without optimizations

```
PrimitiveRules ← fileContents("primitives.rule")
Rules ← PrimitiveRules
Targets ← fileContents("rockyou.txt")
for all p ∈ Targets do
    setRuleHistory(p, {})
end for
Candidates ← Targets
Processed ← {}
while |Candidates| ≥ 0 do
    p ← chooseOne(Candidates)
    Candidates ← Candidates \ {p}
    Processed ← {p} ∪ Processed
    for all r ∈ PrimitiveRules do
        p' ← applyRule(p, r)
        H ← {r} ∪ {append(h, r) | h ∈ ruleHistory(p)}
        setRuleHistory(p', H)
        if p' ∈ Targets then
            for all h ∈ H do
                if h ∈ Rules then
                    s ← getScore(h)
                    setScore(h, s + strength(p'))
                else
                    setScore(h, strength(p'))
                    Rules ← {h} ∪ Rules
                end if
            end for
        end if
        if p' ∉ Processed ∪ Candidates then
            Candidates ← {p'} ∪ Candidates
        end if
    end for
end while
```

Figure 1 shows an example of the combinatorial explosion of candidates that results from the brute-force algorithm.

3.2 Optimizations for effectiveness

The brute-force procedure suffers from significant drawbacks. Since it lacks any criteria for checking rule validity and structure or for preferring to examine some passwords before others, it is likely to generate worthless rules and take a long time to do so.

3.2.1 Hit a target only once

The rockYou wordlist, which is our input to the algorithm, includes some very basic words like 'password' and even the single letter 'a.' The brute-force procedure will discover rules such as ']]]]]] $a' that will transform any six-character password such as 'gh%@_$' into the password 'a,' and the procedure will boost the score of that rule. But that rule is

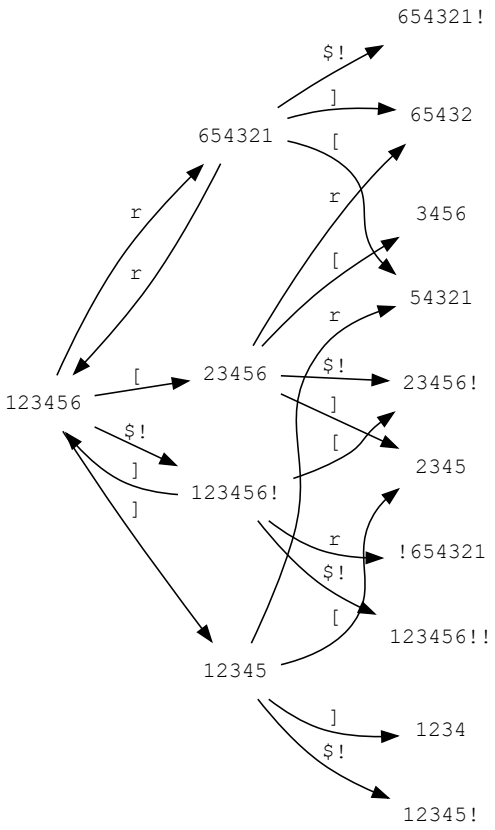

Figure 1: Small example of the combinatorial explosion of passwords generated by applying primitive rules. Note that some passwords may be reached by several distinct rule histories, e.g., starting with password '123456,' the password '54321' may be arrived at by applying complex rules 'r [' or '] r,' or even '$! r [[' (not shown in the graph).

hardly effective for cracking password hashes. However, the procedure will boost that rule for every six-character candidate because the rule will hit a target (namely, the target 'a'). When we allow this behavior, we see that the procedure yields abundant variations of this logic (erasing characters from either end, then adding a few to hit a small target), and they are not effective in experiments.

An easy way to prevent this behavior is to modify the 'if' block starting on line 17 in Algorithm 1 to what is shown in Algorithm 2.

Algorithm 2 Hit a target only once

```
if p' ∈ Targets then
    for all h ∈ H do
        ...
    end for
    Targets ← Targets \ {p'}
end if
```

3.2.2 Ordering by password strength

Because our algorithm applies all primitive rules to each candidate password, we will produce hits faster if the candidate passwords we choose are those that are the most likely to be transformed into a target password. Intuitively, it makes sense that the application of primitive rules to already very strong passwords, such as long passwords consisting of random characters, would be less effective than the application of rules to weaker passwords. In order to select these weaker passwords earlier in our rule generation procedure we first generate individual password strengths for a sample distribution of passwords.

For individual password strength we utilize a metric invented by Joseph Bonneau called the 'partial guessing metric' [1], which was compared to other metrics and determined to be particularly effective. Important properties of this metric are that it provides equal strength to all passwords in a uniform distribution $\mathcal{U}_N$ where each of N events are equally likely and that it rates any event more weakly than events less common in the distribution χ.

This metric is developed with the assumption that the population-wide distribution χ of passwords is completely known and addresses the issue of estimating the strength of previously unseen passwords when a sample is used as an approximation of χ. For our approximation we use as a sample the passwords in the RockYou dataset and the frequency at which they appear. We produce a mapping of the passwords in our distribution to their strengths and provide for estimating the strength of unseen passwords, allowing us to assign each candidate encountered with a strength value.

With strength values known for all initial candidates and the ability to determine the strength of new candidates, we can create a priority queue where high priority candidates are those with a low strength. We select these weaker candidates first, resulting in more hits of target passwords. We previously mentioned that rules have their scores increased when we hit a target password. Instead of simply incrementing the score for all rules equally, we made the decision to use the password strength of the hit password as the number by which the rule score is incremented. This yielded slightly better results than incrementing rules equally on every hit. Intuitively this makes sense. Weaker passwords are likely to be reached by considerably more rule variations and a very long or complicated rule that hits a small number of weak passwords is likely of

Rule length	Original count	Simplified count	Ratio
1	313	313	1.0
2	97,000	90538	0.93
3	30,762,000	26,726,754	0.87
4	296,735,000	255,805,952	0.86

Table 1: Impact of rule simplification. Rule length indicates number of primitives in each complex rule; e.g., 'r] $1' has length 3. Original count specifies the number of rules generated with a certain length, without rule simplification. Simplified count shows number of rules that remain after rewriting some to a simpler form. Simpler forms will typically be repeats and will be removed from the count.

little utility; the passwords probably will be reached by other rules as well. A rule that gets hits on very strong passwords on the other hand is likely to significantly increase coverage, so it makes sense to give these rules a boost.

3.2.3 Rule simplification

The brute-force procedure appends each primitive rule to each rule in a password's rule history on line 15. For example, if the password 'password123' was reached by iterative appending of primitive rules '$1,' then '$2,' then '$3,' the password will have rule history '$1 $2 $3' (among others, possibly). If 'password123' is later selected as a candidate, each primitive rule will be added to the end of that history and tested to see if it hits a target. For example, '$4' will be added and since 'password1234' is a target, each rule in the history (with '$4' appended) will be boosted. Thus, the rule '$1 $2 $3 $4' will be boosted.

We have identified numerous conditions in which complex rules (sequences of primitive rules) are equivalent to a simpler rule. For example, the rule '$1] $a' is equivalent to '$a.' We also normalize rules by reordering some sequences of primitives. For example, the rule '^2] ^1' is equivalent to '^2 ^1]' (they both insert '12' at the front and remove the last character). If we normalize all rules according to some common simplification and sequencing logic, we can be sure to boost the normalized version of a rule instead of boosting different variations and thus lowering the score of the rule.

We have about 50 rule simplifications that are specified as regular expressions. Table 1 shows the number of rules generated originally (without simplification) and the number after simplifying, for different lengths of rules. It is clear that exponential growth is still present as the rule length increases. However, we benefit by ensuring we are scoring the normalized rule rather than equivalent variations.

We modify the brute-force procedure with this optimization at line 15 by first simplifying the new complex rule before adding it to the rule history. This change is shown in Algorithm 3.

Algorithm 3 Rule simplification

$H \leftarrow \{r\} \cup \{\text{simplify}(\text{append}(h, r)) | h \in \text{ruleHistory}(p)\}$

3.2.4 No-op rule detection

We also detect rules that accomplish nothing. For example, the rule 'r r' (reverse, then reverse again) will be boosted repeatedly since it essentially does not transform a password at all. If the candidate password is already a target, then the password generated by this rule is also a target (because it is the same word), so 'r r' will be boosted. In effect, the procedure will yield abundant high-scoring rules that accomplish very little and will not be effective for cracking hashes. These no-op rules are detected and eliminated as shown in Algorithm 4

Algorithm 4 Eliminate no-op rules

$H \leftarrow H \setminus \{h | h \in H : \text{isNoOpRule}(h)\}$

3.2.5 Inventing primitive rules

In order to facilitate generation of complex rules, we promote a complex rule to the primitive rule set if the rule produces a target sufficiently often (we experimentally chose this threshold to be 10 targets). For example, if the primitive rule '$3' is added to a rule history containing '$1 $2' and the resulting complex rule '$1 $2 $3' produces a target at least 10 times, then '$1 $2 $3' is added as a primitive. As a primitive, it will be added as a single unit to other rules, e.g., it will be added to '$1 $2' as in this example, yielding '$1 $2 $1 $2 $3.' In our experimental results section, we will show how many new primitive rules are invented.

3.3 Optimizations for time and memory

Other optimizations ensure our algorithm is time-efficient and uses limited memory.

3.3.1 Use of radix trees

Because a password (potential new candidate) can be reached by many combinations of primitive rules appled to a candidate, it is important for our procedure to recognize which of these potential new candidates have already been processed in order to avoid significant duplicate processing. The naïve approach of using a set very quickly becomes untenable with rapid growth in memory consumption. To mitigate this, our procedure takes advantage of radix trees to store unprocessed and processed passwords. The substring 'password' in 'password123', 'password!1', and 'passwordxyz' will only be stored once. This optimization dramatically slows down memory consumption as our process proceeds.

We make use of the same optimization to store our large number of generated rules.

3.3.2 Capping the candidate set

The main growth of memory in the brute-force procedure is the result of generating new password candidates. These candidates are saved to the queue and processed according to the main loop starting on line 9 of Algorithm 1. In practice, we specify a maximum number of cycles (i.e., how many times to repeat that loop), and we also choose a batch of candidates at a time. We typically run for 1,000 cycles and choose 400 candidates at a time. We can compute the number of password candidates that will ever be examined as the product of these two numbers (400,000). Whenever a password is generated and it was not previously known, it is scored according to its strength and added to a priority queue. Scores do not change after candidates are added to the queue, so periodically (say, every 100 cycles), we eliminate any members of the queue that are below the 400,000th position.

This technique allows us to cap the size of the candidate set. Doing so causes the memory requirements to grow in terms of the size of the radix tree storing the list of generated rules instead of the size of the candidate set. While the brute-force procedure suffers from excessive growth of memory due to the combinatorial nature of password generation, our more efficient variant reduces the resident memory size, thus allowing a significantly longer run time, which results in not only more rules but a better ordering of rules based on their scores.

4 Experimental Methodology

We chose to use the full RockYou wordlist containing about 14 million plaintext passwords. This is our target set, and the original set of candidates. In order to utilize our password strength metric, we require a target list that is sorted by frequency of occurrence in real-world usage. RockYou is not sorted in this way, but we can use the Pwned Hashes list [6], which includes frequencies. Though Pwned Hashes contains hashes, not plaintext passwords, we can hash each RockYou password and look up its frequency in the Pwned Hashes list, and order RockYou by those frequencies.

We ran the algorithm for 1,000 cycles and 400 candidates per cycle, resulting in 400,000 passwords being analyzed. Recall that for each analyzed password, all primitive rules are applied, generating far more candidates than can ever be analyzed.

Our algorithm produces a list of rules. We remove logical duplicates using the 'duprule' program [3], which catches some duplicate rules that our rule simplifier misses. For example, it finds that 'r] ^n' (reverse, remove last, add 'n' to front) is the same as '$n r]' (add 'n' to end, reverse, remove last), so the latter rule is removed. With these deduplicated rules, we use Hashcat and the same RockYou wordlist to attempt to crack the most frequent 100 million Pwned Hashes [6]. We record the percent cracked.

We compared performance of our rules against several other lists of rules, including some that incorporate our own rules:

- An empty rule list, to see what percentage RockYou itself can crack.
- Rules generated from the PACK algorithm [7] on all of RockYou input, then trimmed to various sizes.
- Different sizes of our generated rules, ordered by rule score: top 64 rules, top 10,000 rules, top 100,000.
- The 'best64' and 'dive' rules that come with the Hashcat distribution.
- OneRuleToRuleThemAll (ORTRTA) [11], a combination of other rules: d3adhob0.rule, hob064.rule, KoreLogicRulesPrependRockYou50000, _NSAKEY.v2.dive.rule, and generated2.rule by oclHashcat v1.20.
- OneRuleToRuleThemAll with our generated rules appended, then trimmed to the size of the 'dive' ruleset (99k rules) for comparison purposes, which we refer to as *OneRuleToFindThem* (ORTFT).
- Pantagrule's [10] top-performing rule list, pantagrule.private.v5.popular. Pantagrule rules are generated by running PACK on a private massive set of plaintext passwords, or alternatively, a similarly large set from hashes.org, which is no longer available. Since we do not have access to these sets, we trained PACK on RockYou and call this list of rules PACK, mentioned above.
- Pantagrule's rules pantagrule.private.v5.popular plus our generated rules, with duplicates removed from the combined set.

Note that we try various sizes of our generated rules and PACK generated rules, because we know that these rule are ordered by some kind of score. Other rules from the community are not guaranteed to be ordered.

We also check the number of duplicate rules (according to the 'duprule' program [3]) that we share with other rules like OneRuleToRullThemAll, dive, and Pantagrule-popular. If we find that our generated rules are mostly duplicates of other rules, then our technique is not adding much diversity and therefore not much value. If, however, we find we do not share many rules in common, then our procedure is finding new rules that may be used to complement existing rules created by others.

Generally speaking, one can crack more password hashes by trying more guesses. More rules with the same word dictionary produce more guesses, so more rules generally result in more cracked passwords. However, there are diminishing returns. One rule set might be able to earn a certain percent cracked hashes while another rule set might be able to achieve

the same percent but with fewer rules. The latter rule set will run faster since there are fewer guesses.

We borrow the rules-per-percent metric (RPP) from Pantagrule [10], which is a measure of the average number of rules required to crack a single percent of the hashes for a given set of hashes. We subtract the percent of cracked hashes one obtains by using no rules, i.e., using the RockYou word dictionary on its own (which cracks 6.33% of the top 100 million Pwned Hashes). RPP is defined as,

$$RPP = \left\lfloor \left(\frac{\|Rules\|}{100 * \frac{\|Cracked\|}{\|Hashes\|} - 6.33} \right) \right\rceil$$

where $\lfloor \rceil$ rounds to the nearest integer.

5 Results

A short list of the highest-scoring rules generated by our algorithm is shown in Table 2. These rules match our intuition about how people typically modify an old password or dictionary word to make a new one. We also did a run of Hashcat in 'debug' mode with our top 50k rules, which allows us to record which specific rules in our rule set were responsible for cracking the most passwords. Included in the top ten most successful rules were nine of our highest scoring rules (all except 'remove last character').

As shown in Table 3, while it is not the case that selecting only a 'top-n' set of rules from our generated rules produces a clear win against some common similarly-sized rule sets like 'dive' (99k) or various Pantagrule rule sets, our results clearly indicate that we are generating some strong results with many rules that are not included in these existing lists. Interestingly and somewhat surprisingly, given the relative ease of hand-curating small rule lists, we do exceed best64 in performance with our top 64 generated rules.

In rarecoil's analysis of their Pantagrule rule sets they compare 'dive.rule' to a combination of OneRule and generated Pantagrule rules ('one.rule', equal in size to dive) in order to showcase the utility of their rules. At the time of its creation Pantagrule's 'one.rule' performed better than known lists equal in size to dive and as far as we know this is still true today.

Our results *OneRuleToFindThem* compared to dive and Pantagrule's highly effective 'one.rule' of the same size demonstrates that we are more effective than dive, cracking around 5% more passwords, and almost as effective as 'one.rule,' cracking only 2% fewer passwords. In nearly reaching the efficacy of the most effective known list of this size, we show that our approach is effective. Despite being marginally less effective as a single rule set, our approach uses many rules that Pantagrule does not; this makes our approach to generating rules complementary to Pantagrule's and using both rule files during a cracking attempt should yield good results. Pantagrule notes that their approach has only a couple-thousand

Rule	Score	Explanation
$1	508,091	Add '1' to end
T0	369,973	Toggle case of first character
$2	355,021	Add '2' to end
t	313,526	Toggle case of all characters
$7	290,926	Add '7' to end
$3	284,959	Add '3' to end
]	281,415	Remove last character
$1 $2	273,308	Add '12' to end
$5	253,183	Add '5' to end
$4	246,386	Add '4' to end
$s	239,729	Add 's' to end
$6	232,530	Add '6' to end
$1 $2 $3	229,973	Add '123' to end

Table 2: Top rules generated by our procedure. Scores represent relative success at matching target passwords (an approximation of cracking success).

rule overlap with OneRuleToRuleThemAll; this is true of our *OneRuleToFindThem* as well.

While this comparison highlights the effectiveness of our *rules* in comparison to the Pantagrule rule set, it do not necessarily indicate the effectiveness of our *procedure* compared to the procedure used to generate the Pantagrule rule sets. This is because while we used RockYou as our wordlist, Pantagrule was developed with the use of a public but now inaccessible wordlist (containing about 57 times as many words as RockYou). However, as Pantagrule describes their procedure we can repeat their rule generation using the same wordlist we used, producing effectively our own version of Pantagrule's rule files. These are labeled in Table 3 as the 'PACK top-n' rules. If one compares our top-100k rules to the PACK top-100k rules, we are within 1%, indicating that our procedure is more effective than the comparison of our rules only to Pantagrule's final rules might suggest. Furthermore, we once again excel in the class of very small rule sets, defeating 'PACK top-64' with our top 64 rules.

Overall, while our generated rule files on their own are not unambiguously best-in-class for large rule sets, we believe that in nearly matching a best-in-class approach that we have demonstrated both the efficacy of our own approach to generating rules and produced a set of rules that would be beneficial to use in tandem with rules produced from other top rule lists.

In Figure 2, we see some select rules plotted according to the number of cracked hashes vs. attempted guesses. As Hashcat runs, each word in the wordlist (RockYou) is given to each rule, generating some number of guesses (equal to the number of rules). Then the next word in the wordlist is tried in the same manner. We run Hashcat on a version of RockYou that is ordered according to frequency as given by the Pwned Hashes dataset [6]. We see in the plot that small rule sets like our top-64 rules have a nice trend in the early number of

attempts. This is a result of running through RockYou more quickly with just 64 variations of each RockYou word being tried. However, of course, with only 64 rules, only 25% of the hashes are cracked. Switching to the large Pantagrule-popular + Ours rule list, we see it takes more time to crack an equivalent number of hashes as the 64-rule list, but ultimately finds far more. Curiously, OneRuleToRuleThemAll (ORTRTA) also finds a good amount but has poor performance early on. We have not studied the makeup or organization of ORTRTA enough to understand this behavior.

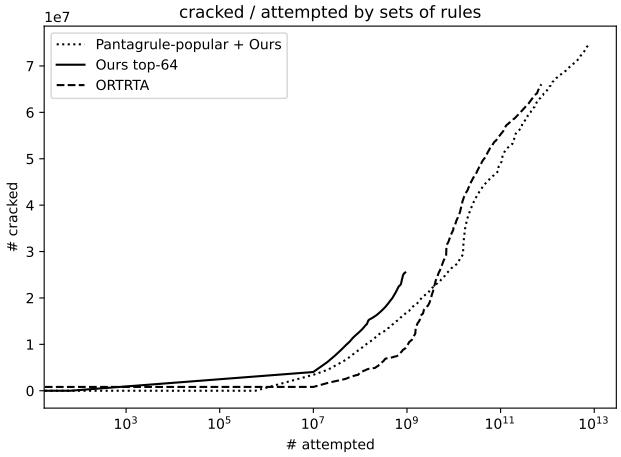

Figure 2: Number of cracked hashes per attempted guess, for select rule sets. Note the x-axis is logarithmic.

As a means of determining if a set of rules is efficient at cracking, we use the rules-per-percent metric (RPP). This metric penalizes larger rule sets that crack the same number of hashes as smaller ones. Table 3 shows the size, cracked %, and RPP for various rules, and Figure 3 plots the relationship between number cracked and RPP. In the figure, the best place to be is near the top left: more cracked, lower RPP.

Examining Table 3, we can summarize these findings by picking the best rule for achieving a certain cracked % while minimizing time (so minimizing RPP). Table 4 shows these results.

Figure 4 shows that the number of complex rules grows per cycle, but gradually levels off. Recall that a complex rule is created when it has never been seen before and is able to transform a candidate password into a target. Over time, fewer rules are generated that are both novel and successful. Also recall that particularly successful rules are promoted to primitives. The frequency of this occurrence also levels off, as shown in the figure.

The 'duprule' program [3] eliminated 34,365 duplicate rules from our generated set of 529,536 rules (6.5%). Table 5 shows how many deduplicated rules generated by our procedure are also found in various other rule lists. The low

Rules	Count	Cracked	RPP
No rules (RockYou itself)	0	6.33%	N/A
PACK top-64	64	24.57%	4
best64	64	24.99%	5
Ours top-64	64	25.49%	3
Ours top-10k	10,000	53.76%	211
PACK top-50k	50,000	61.13%	912
ORTRTA	51,998	66.03%	871
dive	99,092	64.71%	1697
ORTFT (ORTRTA + Ours)	99,092	69.92%	1558
Pantagrule-one-royce	99,092	71.93%	1511
Ours top-100k	100,000	62.79%	1771
PACK top-100k	100,000	63.92%	1736
Pantagrule-popular	478,736	73.98%	7077
Pantagrule-popular + Ours	574,487	74.84%	8385

Table 3: Rules-per-password cracked metric (RPP) for various rule lists, ordered by size of the list and then by cracked %. The formula for RPP is defined in the text.

% cracked	Best rules	RPP
$> 5\%$	No rules (RockYou itself)	N/A
$> 20\%$	Ours top-64	3
$> 50\%$	Ours top-10k	211
$> 60\%$	PACK top-50k	912
$> 65\%$	ORTFT (ORTRTA + Ours)	1588
$> 70\%$	Pantagrule-one-royce	1511
Max	Pantagrule popular + Ours	8385

Table 4: Summary of cracked % and RPP values for top rules.

'percent dup' values indicate that our generated rules do not have much overlap with existing large rule sets. Thus, our techniques compliment each other, and likely the best cracking performance may be obtained by combining rule sets.

In the early cycles of the algorithm, common passwords are selected from the candidate set, and primitive rules are applied to them to generate new passwords. We check if each generated password matches a target password, and if so we call it a 'hit.' Once a target password is hit, it is no longer considered a target. Since many passwords in the RockYou list are simple variations of other passwords in the list, we hit a lot of targets early but fewer over time. This trend is shown Figure 5. The decline in hit percent appears to be exponential.

Figure 6 shows the time required per cycle. As the number of cycles increases, more rules have been generated and stored in the radix tree, thus requiring more work to find and add rules.

Figure 7 shows the growth of resident memory over time. The growth is logarithmic and thus avoids the excessive memory use required by a simple brute-force procedure. Note, however, that the memory usage grows past 60 GB, which is significant for consumer-grade computers. Memory usage can

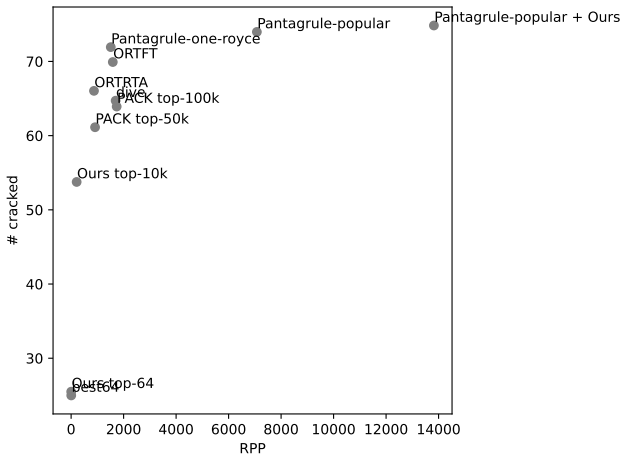

Figure 3: Number of cracked hashes per RPP value, for various rules.

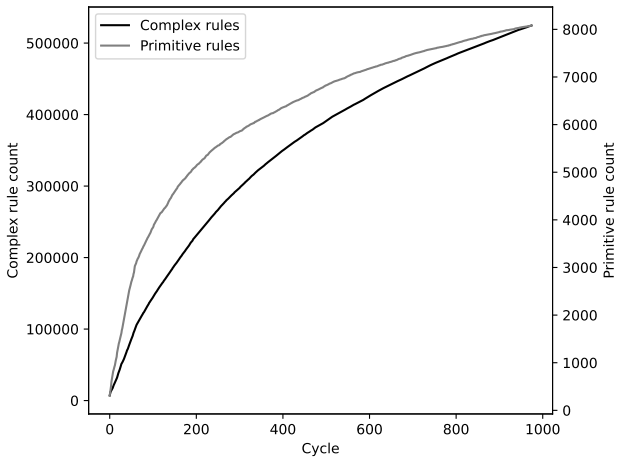

Figure 4: Growth of complex and primitve rules over time (cycles). As targets are hit, more complex rules are added.

Rules	Count	Duplicates	Pct. dup.
best64	64	47	73.4%
ORTRTA	51,998	5,318	10.2%
dive	99,092	3,712	3.7%
PACK-100k	100,000	4,919	4.9%
Pantagrule popular	478,736	11,227	2.3%

Table 5: Counts of rules that are found in both our generated rules and each existing rule set. 'ORTRTA' represents the rule set 'OneRuleToRuleThemAll' [11]. 'Pantagrule popular' refers to Pantagrule's 'pantagrule.private.v5.popular.rule' [10]. The 'Count' column indicates the count of rules in the rule set, the 'Duplicates' column indicates the count of rules in the rule set that match one of our generated rules, and the 'Pct. dup.' column is defined as the 'Duplicates' column divided by the 'Count' column.

be reduced by running the algorithm for fewer cycles (specified by a 'max cycles' parameter) and/or fewer password candidates chosen per cycle (also specified by a parameter).

When we consider Figure 4 (rule growth per cycle), Figure 5 (hit percent per cycle), Figure 6 (seconds per cycle), and Figure 7 (memory per cycle) all together, we see that our algorithm expends a growing amount of resources to generate a decreasing number of rules. The algorithm produces diminishing returns. This is to be expected: the 'easy' and most successful rules are found early, while uncommon rules that hit password targets that are infrequent (passwords that rarely appear in the Pwned Hash set) are found rarely and only after extensive searching.

The same phenomenon can be observed in Table 3, which shows the 'rules-per-password cracked' (RPP) metric for various rule lists. This metric estimates the number of rules required to crack a single password. With very small lists of just the most effective rules, such as our top-64 or best64 from Hashcat, one can crack a significant portion of hashes with minimal effort. These are the 'easy' hashes. The long-tail of rare passwords are much harder to crack and require more work the greater their rarity.

6 Future Work

We have not exhausted all possibilities for strong algorithms or rule-generation heuristics in the area of 'rule-first' rule generation. For example, one could imagine taking our approach but boosting rule scores only if a rule produces a stronger password from a weaker one, possibly eliminating some rules (like many including 'remove last character') that we currently score higher than their experimental results indicate is appropriate.

In order to get a better picture of RPP trends, one of the more important metrics in analyzing the efficacy of rule sets, more research and experimentation with different data sets and more sizes would be beneficial.

Interestingly, our research has also revealed that many existing lists of rules include many rules that are functionally duplicates of each other, making the cracking process unnecessarily longer. 'duprule' or a similar program should probably be run against most existing rule lists before using them in a cracking attempt, and more research into making sure rule lists contain as few functional duplicates as possible is certainly warranted. More research is merited in the area of preventing the generation of duplicate Hashcat rules with existing automated approaches.

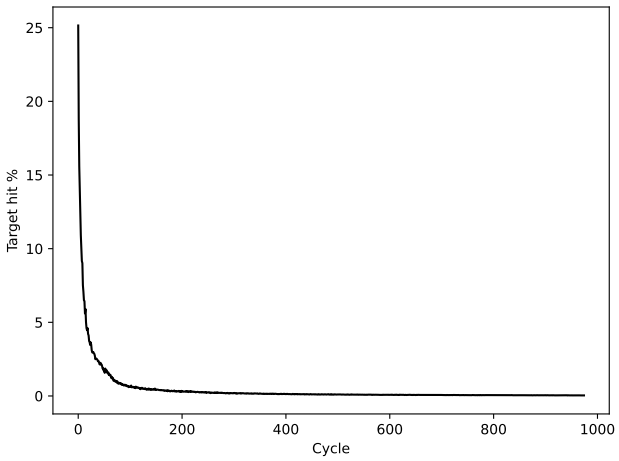

Figure 5: Percent of generated passwords that are target passwords (which we call a 'hit'), per cycle.

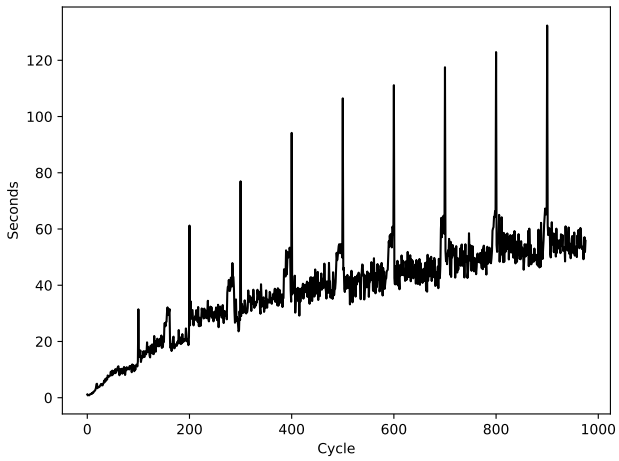

Figure 6: Time per cycle. The spikes are due to the time required every 100 cycles to reduce the queue of candidates (a priority queue) to limit memory growth.

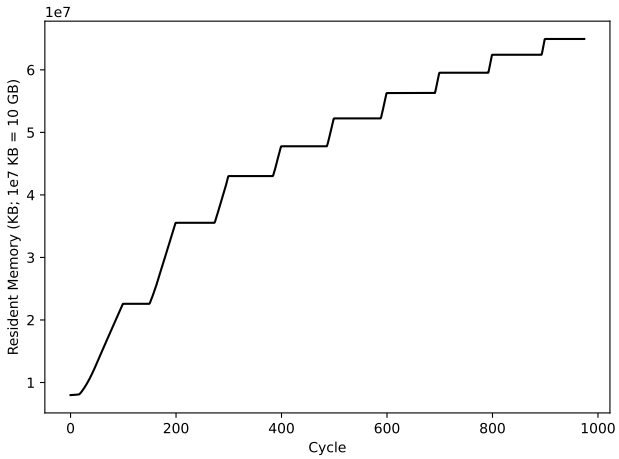

Figure 7: Resident memory per cycle. The growth is logarithmic due to the capped candidate set size and the logarithmic growth of the radix tree storing generated rules. The memory plot is not smooth because each 100 cycles memory is reduced by trimming the candidate set size. However, the process keeps this free memory space reserved for some time rather than releasing it back to the operating system.

Acknowledgments

We wish to thank XYZ for his contributions to this project.

Availability

Our code and results are available on GitHub at github.com/REDACTED/REDACTED. We used various datasets to generate our results:

- RockYou plaintext passwords: github.com/zacheller/rockyou
- Pwned Passwords version 8, ordered by prevalence: haveibeenpwned.com/Passwords
- Pantagrule rules: github.com/rarecoil/pantagrule
- Common English words: github.com/alex-pro-dev/english-words-by-frequency

Hashcat was used to measure the performance of rules: github.com/hashcat/hashcat. We also used 'duprule,' a duplicate rule detector: github.com/mhasbini/duprule.

References

[1] Joseph Bonneau. Statistical metrics for individual password strength. In *Security Protocols XX: 20th International Workshop, Cambridge, UK, April 12-13, 2012, Revised Selected Papers 20*, pages 76–86. Springer, 2012.

[2] Ameya Hanamsagar, Simon S Woo, Chris Kanich, and Jelena Mirkovic. Leveraging semantic transformation to investigate password habits and their causes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[3] M. Hasbini. duprule: Remove duplicate Hashcat rules. https://github.com/mhasbini/duprule. Accessed: 2023-02-05.

[4] Hashcat. Hashcat. https://hashcat.net/hashcat/. Accessed: 2023-02-07.

[5] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. PassGAN: A deep learning approach for password guessing. In *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, pages 217–237. Springer, 2019.

[6] Troy Hunt. Pwned Passwords. https://haveibeenpwned.com/Passwords. Accessed: 2023-02-05.

[7] Peter Kacherginsky. PACK. https://github.com/iphelix/pack. Accessed: 2023-02-07.

[8] NSAKEY. nsa-rules. https://github.com/NSAKEY/nsa-rules. Accessed: 2023-02-07.

[9] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, Massimo Bernaschi, and Mauro Conti. Improving password guessing via representation learning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1382–1399. IEEE, 2021.

[10] rarecoil. Pantagrule: Gargantuan hashcat rulesets generated from compromised passwords. https://github.com/rarecoil/pantagrule. Accessed: 2023-02-05.

[11] Not So Secure. One Rule to Rule Them All. https://notsosecure.com/one-rule-to-rule-them-all. Accessed: 2023-02-05.