Reasoning Analytically About Password-Cracking Software

Enze Liu, Amanda Nakanishi, Maximilian Golla†, David Cash, Blase Ur
University of Chicago, † Ruhr University Bochum
Email: {alexliu0809, anakanishi, davidcash, blase}@uchicago.edu, maximilian.golla@rub.de

***Abstract*—A rich literature has presented efficient techniques for estimating password strength by modeling password-cracking algorithms. Unfortunately, these previous techniques only apply to probabilistic password models, which real attackers seldom use. In this paper, we introduce techniques to reason analytically and efficiently about transformation-based password cracking in software tools like John the Ripper and Hashcat. We define two new operations, rule inversion and guess counting, with which we analyze these tools without needing to enumerate guesses. We implement these techniques and find orders-of-magnitude reductions in the time it takes to estimate password strength. We also present four applications showing how our techniques enable increased scientific rigor in optimizing these attacks' configurations. In particular, we show how our techniques can leverage revealed password data to improve orderings of transformation rules and to identify rules and words potentially missing from an attack configuration. Our work thus introduces some of the first principled mechanisms for reasoning scientifically about the types of password-guessing attacks that occur in practice.**

I. Introduction

Humans use predictable patterns in passwords [1], [2]. Modern password cracking exploits these patterns using data-driven methods relying on large corpora of leaked passwords [3]–[5]. It has become typical to measure password strength by simulating such password cracking [6]–[9]. Many password-cracking algorithms are probabilistic, creating a model and then assigning a probability to each possible password. Assuming the attacker rationally guesses passwords in descending order of likelihood, the strength of a password is proportional to the number of passwords with higher probability by that model. Researchers have developed efficient algorithms for estimating this mapping for the major probabilistic algorithms: Markov models [10]–[12], neural networks [13], and probabilistic context-free grammars [7], [10], [14].

Unfortunately, real-world attackers rarely use probabilistic tools; they use software like John the Ripper (**JtR**) and Hashcat [15]. Their reasons are pragmatic. In offline attacks, the wall-clock time to make and check a guess includes the time to generate a candidate, as well as the time to hash the candidate and see if it appears in the target store. While probabilistic algorithms perform well on a guess-by-guess basis, they impose a high computational cost for generating a guess [8]. Thus, for all but the slowest hash functions, it is faster to crack a comparable number of passwords using these software tools. While the likelihood of a password by a probabilistic model may correlate with the order in which software tools guess passwords [8], this proxy is imperfect.

JtR and Hashcat's mangled-wordlist attacks are their most commonly used [15] and most intellectually interesting. These attacks leverage the insight that passwords tend to differ in small and predictable ways; while one person may append a digit to a word, another might append a symbol to that same word. In a mangled-wordlist attack, the attacker creates a **wordlist** of common passwords and natural-language content, as well as a **rule list** of mangling rules (e.g., replace 's' with '$' and append a digit) written in a transformation language specified by the tool. The full attack applies each mangling rule to each word in the order specified by the input lists.

Thus, the practical strength of a password `pw` is strongly affected by if and when JtR/Hashcat would guess `pw`, yet it is difficult to compute this information. To date, one would simply run JtR/Hashcat on a given rule list and wordlist, enumerating guesses and recording when `pw` is generated [8]. This has several limitations. Generating a huge number of guesses is computationally expensive, and when one stops the attack it is not known which unguessed passwords would ever be guessed. Moreover, it is unclear which rule list and wordlist an intelligent attacker should choose, and how they should be ordered. Re-running JtR/Hashcat on the myriad possible lists is intractable. Knowledge of these tools' complex behaviors is limited to glimpses obtained via expense computations.

Our first contribution: An analytical approach to modeling transformation-based password guessing. We develop a more efficient approach for studying mangled-wordlist attacks without actually running them. We design and implement tools that analytically compute properties of JtR and Hashcat, including whether they would generate a particular password and how many guesses each rule generates. These techniques enable estimation of a password's strength by accurately and efficiently computing how many passwords would be guessed before it in an attack using these tools in a particular configuration. We term this process a **guess-number calculator**. While prior work has developed guess-number calculators for Markov models [10], probabilistic context-free grammars [7], and neural networks [13], ours is the first computationally efficient approach for modeling widely used cracking software.

In particular, we develop modules for **rule inversion** and **guess counting**. Rule inversion efficiently computes a compact representation describing the preimage set of a rule for a password `pw` (the set of words the rule will mangle into `pw`). This allows one to easily see whether a rule would generate

a target password with a given wordlist. Guess counting computes the number of guesses generated by a rule without running it. Both modules run much faster than naive execution.

For example, consider the task of determining whether passwords `password156` and `monkey!` would be generated by a rule that appends two digits to every entry of a given wordlist. One could first invert that rule by attempting to remove two digits from the end of the password. If this inversion is successful (`password156` becomes `password1`), then the task is reduced to a constant-time lookup of whether `password1` is in the wordlist. If (and only if) it is, `password156` would be guessed. Because `monkey!` does not end in two digits, we determine it cannot be guessed by that rule. We can also determine how many guesses this rule issues: 100 times the number of entries in the wordlist.

While these computations were straightforward for this simple rule, JtR and Hashcat support dozens of more complex transformations (e.g., substitutions, purging classes of characters, conditionally rejecting candidate guesses). Furthermore, a rule can compose many individual transformations. We formally analyze both tools' full rule languages, building efficient algorithms for handling most complex rules. Our tool can invert and guess count the vast majority of transformations supported by JtR/Hashcat, as well as arbitrary compositions of transformations. However, for some rules (e.g., character purging), we must resort to brute-force execution of JtR/Hashcat.

Given a password `pw`, our guess calculator determines how many guesses JtR/Hashcat would issue before guessing `pw`. Doing so efficiently enables quick estimation of a password's strength. After some moderate precomputation, our calculator responds in only a few seconds even for attacks making 300 trillion guesses, enabling the first real-time estimation of password strength against common mangled-wordlist attacks.

Our second contribution: Configuration tools/experiments. Probabilistic password-cracking algorithms use training data (typically huge sets of passwords) to model the probability of previously unseen passwords. While each probabilistic algorithm is somewhat sensitive to tweakable configuration parameters (e.g., the amount of smoothing [11] or the number of features [13]), tools' mangled-wordlist attacks are highly sensitive to their configuration [8]. This configuration encompasses what words and rules are included in the wordlist and rule list, as well as how those words and rules are ordered. What words and rules are included impacts whether passwords would be guessed at all, and the ordering impacts how quickly they will be guessed. In practice, attackers order rules based on intuition or taste, and experienced attackers closely guard personal lists they have developed and refined over years of cracking passwords [15]. The academic community gets small glimpses into these lists, such as through example lists [16] released as part of password-cracking contests [17] or distributed with the JtR and Hashcat software. Experienced attackers' lists, however, outperform those released publicly [8].

We extend our analytical approach to design, implement, and evaluate four optimization techniques that collectively enable data-driven configuration of transformation-based attacks. These optimizations make configuration more principled and less ad-hoc, and they also better align academic models with passwords' vulnerability to real-world attacks by experts using non-public lists. These four applications optimize both the order and the completeness of both rule lists and wordlists.

First, we present a principled way to order the rules in a rule list. Applying this optimizer to six real-world rule lists, we estimate the best possible guessing effectiveness of an attacker who might have sorted the rules differently. We find that the optimal order is fairly consistent across sets of passwords created under similar composition policies, but notably less consistent otherwise. Interestingly, we find that the order of one well-known rule list (SpiderLabs [18]) can be substantially improved, while another rule list (Megatron [19]) is already in nearly optimal order. We then apply an analogous process to optimizing the order of words in a wordlist, finding that doing so generally overfits to data and worsens configurations.

Finally, we use our tools to analyze the completeness of these lists in a principled way. We generate 15,085 JtR rules and integrate them into existing rule lists. Doing so results in passwords being guessed more quickly, and a larger fraction of passwords being guessed overall. We then analyze the completeness of our wordlists. We apply rule inversion to identify words that *would* have generated target passwords in evaluation sets had they been in wordlists, adding those that appear most often. This method suggests semantically meaningful words and short strings that collectively enable guessing passwords that otherwise would not be guessed.

In sum, we introduce some of the first techniques for reasoning analytically about the transformation-based password-guessing attacks that actually occur in the wild. Our guess-number calculator enables real-time, server-side password checking, while our optimizations better align academic models with experts' closely guarded configurations. To encourage further scientific modeling, we are open-sourcing our code.[1]

II. Related Work

We first present prior work using password-cracking algorithms as metrics of password strength. We then discuss probabilistic models of passwords and ways of analyzing them. Finally, we discuss the limited prior work analyzing transformation-based password-cracking software. Our work focuses on trawling password-guessing attacks. While password guessing on live servers can be rate-limited, large-scale (offline) guessing remains an important threat for three reasons: (1) humans often reuse credentials across accounts [20]; (2) rate-limiting is not always implemented correctly [21]; and (3) guessing against encrypted files (where cryptographic keys are derived from passwords) cannot be rate-limited without proportionally increasing the cost of legitimate accesses.

A. Metrics of Password Strength

Entropy was historically used as a metric of password strength. However, entropy is calculated for distributions, not

[1] https://github.com/UChicagoSUPERgroup/analytic-password-cracking

individual passwords, and ad-hoc attempts to estimate entropy for single passwords using heuristics are often inaccurate [7], [9], [22]. In response, Bonneau introduced partial guessing metrics that model attackers who guess optimally based on statistical properties of passwords in a target set [1]. While such methods have advantages, they require huge amounts of data and do not model actual attacks [8].

Many researchers thus model password strength by running or simulating password cracking [2], [7], [9], [11], [23]–[29]. This metric, **parameterized password guessability**, considers the strength of a given password against a trawling, non-targeted attack to be proportional to the number of other passwords that would be guessed before it [7]–[9]. We use the term **guess number** to refer to the number of passwords guessed before a given password in an attack.

B. Password-Cracking Algorithms

Large corpora of passwords are the best source of guesses. However, attacks can make billions of guesses per second [5], yet only a few billion real passwords have been leaked [30]. Thus, after guessing previously seen passwords in descending order of frequency, attackers turn to password-cracking algorithms. Academic studies of password cracking focus on the following probabilistic algorithms, which construct models mapping passwords to probabilities based on training data:

1) Markov Models: In 2005, Narayanan and Shmatikov first proposed using Markov models of natural language to guess passwords [31]. Researchers subsequently proposed adaptive Markov Models trained on passwords [6] and mechanisms for more efficiently enumerating guesses [12]. Ma et al. analyzed Markov models' ability to guess different password sets, finding 6-gram Markov models with additive smoothing to guess English-language passwords most effectively [11].

2) Probabilistic Context-Free Grammars: In 2009, Weir et al. proposed using a probabilistic context-free grammar (PCFG) to guess passwords [14]. Training on sets of passwords, a PCFG models a password's probability as the probability of its character-class (or semantic [32]) structure multiplied by the probabilities of its component strings.

3) Neural Networks: In 2016, Melicher et al. proposed recurrent neural networks composed of long short-term memory (LSTM) units to guess passwords [13]. Like Markov models, this approach calculates the probability of a subsequent character in a candidate password based on the preceding characters, yet does so using a multi-layer neural network.

C. Efficiently Analyzing Password-Cracking Algorithms

Because calculating a guess number requires running or simulating a password-cracking algorithm, it is crucial to do so efficiently. Researchers have developed efficient techniques for the three main probabilistic algorithms. Kelley et al. proposed fast lookups of a password's guess number by accepting a time-space tradeoff, precomputing very large lookup tables of probabilities of structures and terminal strings for PCFGs [7]. Dell'Amico et al. relied on Monte Carlo methods to estimate the mapping between a given password's probability and the number of passwords with higher probability [10], efficiently computing guess numbers for Markov models and PCFGs. Melicher et al. extended this Monte Carlo approach to neural networks, further leveraging a very small model size to perform client-side password-strength evaluation [13]. Finally, Ma et al. introduced probability-threshold graphs, which can be computed efficiently and enable comparisons of the relative strength of password sets [11]. This technique, however, cannot be used directly to calculate guess numbers.

While these techniques enable efficient analysis of probabilistic models, real attackers rarely use such models [5]. Sadly, these techniques cannot be applied to the software attackers actually use because such tools do not model probabilities [15]. We fill this gap by introducing analytical techniques to reason efficiently about password-cracking software.

The password-cracking community has developed anecdotal and informal best practices for configuring cracking software [5], [15]. Little work, however, has attempted to optimize, or even reason about, these tools in a principled or scientific way. Chrysanthou presents a multi-stage attack for Hashcat based on numerous empirical experiments on one test set [33]. These experiments require enumerating guesses and thus preclude efficient analysis at scale. Absent methods of reasoning about such tools, many researchers use them in their default configuration [9], [27], [34]–[36]. Ur et al. [8] found these configurations to severely underestimate passwords' vulnerability to attacks. In Section VIII, we show that Ur et al.'s approach itself underestimates vulnerability.

III. Background

Best practice dictates storing passwords salted and hashed using a computationally expensive hash function (e.g., scrypt, Argon2), though fast hash functions (e.g., MD5) are widely used [30]. When data breaches occur, attackers generate likely candidates, hash them, and compare these hashes to those revealed in the breach. A similar process occurs when cracking password-protected files. In targeted attacks, which are out of scope, the best sources for guesses are the user's personal information [37] and passwords associated with the same username in prior breaches [20]. For the trawling attacks we model, the best sources of guesses are any real passwords that have been previously revealed in descending order of frequency [8]. To date, over 5 billion accounts have been breached [30]. Attacking fast hashes with GPUs, however, takes under one second to make 5 billion guesses [15]. To generate vastly more guesses, attackers use data-driven approaches perturbing leaked data. In this section, we explain JtR and Hashcat's transformation-based approaches. Our techniques enable the first principled analysis of such approaches.

A. Transformation-Based Mangled-Wordlist Attacks

Password-cracking software is most frequently and canonically used to perform what we term **mangled-wordlist attacks** ("wordlist mode" in JtR and "rule-based attacks" in Hashcat). Our techniques focus on these attacks, which take both a wordlist and a rule list as input. **Wordlists** typically contain

passwords from data breaches, words from dictionaries, and natural-language data (e.g., phrases) [15]. **Rule lists** contain rules, which are compositions of individual transformations (including conditional logic) written in tool-specific languages that we detail in the following section.

In a mangled-wordlist attack, each word from the wordlist is mangled, or transformed, as specified by each rule from the rule list. Note that JtR and Hashcat order guesses differently. JtR proceeds in rule-major order, applying a given rule from the rule list to all words on the wordlist before proceeding to the next rule. Hashcat conceptually follows word-major order, applying many rules from the rule list to a given word before proceeding to the next word to minimize disk I/O. In practice, Hashcat follows a more complex and hardware-dependent batching strategy (see Section IV-D) to further improve performance. To guess a larger fraction of passwords more quickly, both wordlists and rule lists should be ordered in descending order of likelihood for generating a successful guess. Because of the ordering, however, JtR's effectiveness is particularly sensitive to the ordering of the rule list, whereas Hashcat's is more sensitive to the order of the wordlist.

Both JtR and Hashcat come with preliminary rule lists, sample configuration files, and short sample wordlists. As they gain experience, attackers augment and refine their own lists. From the results of contests [17] and real-world hacks [15], these closely guarded personal lists are far more effective than these preliminary lists, or even others released publicly. Most rules contained in rule lists were initially created by human experts using their intuition about how humans create passwords by transforming natural language or by manually examining patterns within released password sets [18]. There have been some limited attempts to automate this process. For example, Marechal proposed ordering rules based on the number of passwords they cracked in a given evaluation set [38], while Hashcat's `generated2.rule` was created by randomly generating rules and testing their effectiveness cracking sets of passwords [8]. Taking a data-driven approach, Marechal automatically generated potential prepend and append rules by searching for words on the wordlist as substrings of passwords in evaluation sets [38]. Kacherginsky built on this work, using the Reverse Levenshtein Path between wordlist entries and passwords in evaluation sets to generate rules [39]. The techniques we introduce provide a more generalized framework for data-driven rule creation (Section X).

B. Language of Transformation Rules

Password-cracking software tools have well-defined, well-documented rule languages that enable rule creators to express potential ways in which natural language or previously seen passwords might have been transformed to create a password. We use the term **transformations** to refer to the individual commands that, when applied to a wordlist in a mangled-wordlist attack, change an input word to generate a new candidate guess or express conditional logic related to whether or not to make that guess. Each individual transformation has a unique function name. JtR supports 52 individual transformations [40], while Hashcat supports 55 individual transformations [41]. The two tools have 32 transformations in common. Some transformations do not take parameters. For example, `C` in both JtR and Hashcat lowercases the first character of the input and uppercases the rest. Other transformations take parameters that are either characters or numerical (a position, length, or count). For example, both tools' `sXY` transformation replaces all instances of the character X with the character Y.

Each rule in a rule list consists of one or more transformations and is parsed left to right. Table I gives our categorization of the transformations each tool supports, in addition to whether those transformations can be reasoned about analytically (Section IV). As detailed below, some transformations modify the input in ways that are straightforward to describe, and many (but not all) of these are straightforward to reason about. Some transformations express logic related to *rejection* (choosing not to make a guess if certain conditions are not met), and much of the complexity in the design of our analytical techniques derives from how this conditional rejection logic interacts with other transformations in a rule.

Some example JtR rules follow. `Az"[0-9]"` appends a digit to the end of the input word, thus making ten guesses (one per digit) for each input word. Slightly more complex, `/a saA >5 /?d /?a` immediately rejects the guess unless the input word contains "a," and otherwise replaces every "a" with "A." It then rejects the guess unless it contains more than 5 characters, as well as both a digit and a letter (uppercase or lowercase). Accounting for parameterizations (e.g., which characters are being substituted), there are over 15,000 JtR valid rules containing only a single transformation (see Section X). This increases exponentially with length; there are hundreds of millions of possible length-2 rules.

1) Straightforward Transformations: The transformations most straightforward to describe directly modify some aspect of the input word. These include transformations that shift the case of letters in the input word (e.g., `l` lowercases the word), insert or delete either a character or a set of characters (e.g., `$c` appends "c" while `D6` deletes the 7th character, zero-indexed). Other transformations substitute characters (e.g., `R` shifts each character one key right on a QWERTY keyboard) or rearranges them (e.g., `r` reverses the input). Other JtR transformations use heuristics to simulate English grammar rules for pluralization and putting words into either the past tense or gerund form.

2) Rejection Transformations: To avoid making guesses that have not modified the input word, that (after transformation) do not comply with a password-composition policy, or that are otherwise redundant, both tools include rejection transformations (not making a guess unless conditions are met). For example, if an attacker previously brute-forced all passwords of lengths 1–5 or if they knew all passwords contained at least 6 characters, they might include `>5` in rules to reject guesses containing five or fewer characters. Similarly, the transformation `sa@` (replacing every 'a' with '@') could be prefixed with `/a` to reject inputs without an "a." Other rejection transformations examine the n^{th} character of a guess (e.g., `=NX`) or count the instances of a given character or character

TABLE I: Categorization of JtR and Hashcat transformation rules and whether they can be reasoned about analytically.

Transformations	Fully Invertible, Countable	Regex-Invertible, Countable	Fully Invertible, Uncountable	Uninvertible, Uncountable
John the Ripper				
Shift Case	8	0	0	0
Insertion/Deletion	8	2	0	2
Substitution	5	0	0	0
Rearrangement	6	0	0	0
English Grammar	3	0	0	0
Rejection	14	0	0	0
Memory	0	0	0	4
Total	**44**	**2**	**0**	**6**
Hashcat				
Shift Case	6	0	2	0
Insertion/Deletion	7	2	0	1
Substitution	8	0	0	0
Rearrangement	15	0	0	0
Rejection	9	0	0	0
Memory	0	0	0	5
Total	**45**	**2**	**2**	**6**

class (e.g., `%NX`). While JtR and Hashcat both support rejection transformations, for performance reasons Hashcat only does so in special cases (e.g., CPU-based hashcat-legacy). JtR also minimizes redundancy by only making guesses that differ from their immediate predecessor guess.

3) Memory Transformations: Both JtR and Hashcat provide memory transformations that enable setting or querying variables in memory, enabling complex modifications to an input and helping to avoid redundant guesses. Note that variables can represent either substrings or numbers (e.g., positions or lengths). While we support one frequent special case, we do not support memory commands generally (see Appendix A).

4) JtR's Rule Preprocessor: While Hashcat's transformations generally cause an input word to generate a single guess (or none, based on rejections), JtR has a rule preprocessor that allows multiple similar rules to be expressed compactly as a single rule. For example, the JtR rule `r $[0-9]` reverses the input and then appends a number. Writing the equivalent in Hashcat's notation requires either ten rules or a hybrid attack combining a mangled wordlist with selective brute-forcing.

C. Password-Cracking Software's Other Attack Modes

While mangled-wordlist attacks are the most commonly used and intellectually interesting, JtR and Hashcat support other attacks. Brute-force attacks test all characters in a given keyspace, while mask attacks are a subset in which the keyspace is constrained. These are trivial to analyze by checking if a password is in the given keyspace. Combinator attacks concatenate two words and are trivial to reason about because a given password can be split into parts, checking each part for membership in the wordlist. Hybrid attacks combine these approaches and can be reasoned about similarly.

IV. Our Tools: Rule Inversion and Guess Counting

In this section, we describe the two main components of our analytical tools: *rule inversion* and *guess counting*. We start with an overview of their interfaces and purposes, then discuss their algorithmic design and implementation. We also present how these components are combined into the higher-level application of computing *guess numbers* for passwords. We completed implementations for both JtR and Hashcat, but we focus on JtR, discussing our Hashcat tool at the end.

1) Rule Inversion: Recall that JtR applies a rule to every wordlist entry to generate a possibly large number of guesses. To determine if particular target passwords are among these guesses, one can run the tool and store the guesses, but this is expensive in both computation and storage. We designed a more efficient, analytical method that works for most, but not all, rules. Specifically, we design and implement an efficient function `invert_rule` with the signature:

$$\texttt{regex} \leftarrow \texttt{invert\_rule}(\texttt{rule}, \texttt{pw}).$$

The function `invert_rule` takes as input a rule `rule` and a target password `pw`. It outputs a regular expression `regex` that decides the set of *preimages of `pw` under `rule`* (i.e., words `w` that generate `pw` when mangled with `rule`). This set may be empty (`regex` matches nothing). For uninvertible rules we allow an error mode indicating no regex was computed.

Two types of regular expressions may be output. A *simple regex* does not use $*$ or $+$ operators, consisting only of a sequence of character classes. For example, `s[3eE]cr[3eE]t` is a simple regular expression (matching 9 total strings), but `secret(0*)` is not (matching an infinite set, as it allows any number of trailing zeros). A *non-simple regex* uses $*$ and $+$. When our tool can compute a simple regex for a rule, we term the rule *fully invertible*. When a non-simple regular expression is computed, we say the rule is *regex-invertible* to indicate a general expression must be used in future computations. If no regex can be output, we say the rule is *uninvertible*.

As discussed below and indicated in Table I, we compute simple regexes for (possibly very complex) rules built from 44 of the 52 possible JtR transformations. For such rules, we determine if a rule would guess a given password quickly (i.e., without a linear scan of the wordlist). For two transformations (truncation and substrings), we produce non-simple regexes. For six uninvertible transformations (two forms of character purging and four memory commands), we do not produce a regex. As detailed in Appendix A, for these we fall back to enumerating guesses out of performance considerations (non-simple regexes) and necessity (uninvertible transformations).

The primary usage of rule inversion is to determine if any entry of a wordlist `wlist` would guess `pw` under `rule`. For a non-simple regex, we test if the regex recognizes each entry of `wlist` via a linear scan. A simple regex enables faster checking, however, because we can enumerate the recognized strings by filling in the possible choices and checking (via a hash table) whether each possible string is a member of `wlist`. For most simple regexes generated for practical rule lists, the regex recognizes only a small number of possibilities, so this enumeration is far faster than passing over `wlist`.

Unfortunately, in some corner-cases that depend on both `pw` and `rule`, our tool will generate a regex that is simple, yet enumerates a large number of values. For example, inverting the password `@@@@@@@@@@` under the rule "substitute @ for a" results in the regex `[a@]{10}`, which enumerates 2^{10}

strings, all of which must be checked for membership in wlist. In these cases, we avoid brute force enumeration of the strings matched by the simple regex. Instead, we represent wlist using a trie [42]. Because only a small number of the matched strings will typically be in the wordlist, and the strings will tend to share prefixes, most of those strings can be skipped. That said, the theoretical worst-case complexity is not changed. For example, the rule "delete the first four characters" results in a regex like [anychar]{4}input_password, which matches any string that has four arbitrary characters prefixing the input password, causing either approach to be slow. To address this, we built a proactive toolkit to find such rules and mark these rules as uninvertible. In principle, there exist pathological rules causing all three approaches to fail, though we observed none in practical lists. For such rules, human-in-the-loop tuning would be required.

2) Guess Counting: Our guess counting component answers the question: *Given wordlist wlist and rule rule, how many guesses will JtR generate?* An analogous question is posed for Hashcat, which differs by guessing in word-major order. We say a rule is *countable* if our tool can count guesses faster than running the software, and *uncountable* otherwise. Guess counting is significantly more complex than rule inversion. We have factored it into three pieces:

$$\texttt{feat\_grps} \leftarrow \texttt{extract\_features}(\texttt{rlist}),$$
$$\texttt{aux\_info} \leftarrow \texttt{precompute}(\texttt{feat\_grps}, \texttt{wlist}),$$
$$\texttt{num} \leftarrow \texttt{guess\_count}(\texttt{aux\_info}, \texttt{rule}).$$

We describe these starting with guess_count. Using some precomputed information, it outputs the number of guesses JtR generates running rule on wlist. This function itself is fast, running in time independent of the wordlist size.

Functions extract_features and precompute perform precomputation to make guess_count fast. extract_features takes as input a rule list rlist and outputs feat_grps, a data structure representing which abstract word properties (e.g. length, the presence of a digit) are relevant for counting the guesses made by rules in rlist. Function precompute takes as input feat_grps and a wordlist wlist and computes auxiliary information aux_info. This information enables us to quickly count words with combinations of properties via a lookup table instead of making multiple passes on the entire wordlist.

guess_count estimates the runtime of JtR on rules without executing them. Moreover, each run will be in time sublinear in the size of wlist. To this end, we allow one-time expensive precomputation in extract_features and precompute. The precomputation of aux_info is done up-front and only once, encoding information about the properties of wlist, such as the number of words of a given length and the number of those that contain a digit. Computing feat_grps is fast, but aux_info can take on the order of hours, and it can be large (on the order of a few GB). This is often still far faster and smaller than running JtR. With this data, evaluating guess_count typically only takes seconds per rule, which is crucial when there are thousands of rules. In our higher-level applications, we only need the output of guess_count, so we can delete aux_info after using it.

While many rules are easy to count analytically, they quickly become complicated when rules (especially those with rejections) are composed. Our tool handles the complexity of arbitrary compositions of invertible transformations.

A. Rule Inversion

We describe how invert_rule works on single transformations, and then how it is extended to composed rules.

1) Single Transformations: For most rules consisting of a single transformation, invert_rule(rule,pw) is straightforward. For instance, if rule appends a character d, then it outputs the empty regular expression if pw does not end in d, and otherwise outputs the regular expression matching exactly pw with the trailing d deleted. In either case, this regular expression is *simple* (using our definition above).

For some transformation classes, this approach produces more complicated, but still simple, regular expressions. Consider the rule sXY that substitutes all X characters for Y. To compute invert_rule, we replace each occurrence of Y in pw with the regular expression [XY]. For example, if pw = aYbY, invert_rule outputs regular expression a[XY]b[XY], which is simple by our definition.

We applied this analysis to handle 44 of the 52 JtR single transformations (see Table I). We omit tedious details from the paper, but our open-source implementation fully specifies the process. For two other transformations (truncate and substring), our tool computes non-simple regexes. To see why this was necessary, consider a rule that truncates a word down to 4 characters (or takes a substring of length 4). We can represent the preimage set for such a rule using a general regular expression, but there may be a huge number of preimages.

For the two *purging* rules (@X and @?C which purge all "X" characters or characters from class C, respectively), our tool does not produce a regex. The difficulty with these becomes apparent when considering composition and we return to this below. Finally, we do not compute regexes for the four memory access commands as doing so would require analyzing what amounts to arbitrary programs (see Appendix A).

2) Composed Rules: We handle composed rules by adapting our single-transformation ideas to work on regexes, rather than strings of literals only. This allows us to feed the regex output of one single-transformation computation into another single-transformation computation. Moreover, if two stages individually produce a simple regular expression, then their composition will also produce a simple regular expression.

For example, consider the composed rule {sXY, which rotates the string one character left and then substitutes occurrences of X with Y. For aYbY, invert_rule proceeds:

1) Invert sXY on pw, getting a[XY]b[XY].
2) Invert { on a[XY]b[XY], getting [XY]a[XY]b.

The key observation is the second stage can unambiguously manipulate the regex to arrive at the correct answer.

This process works for arbitrary compositions of all single transformations that produce simple regexes. We can now explain why we do not produce regexes for purge commands.

Consider the seemingly innocent rule `$Y@X` ("append `Y` then purge all `X`"). Attempted inversion on `pw = Y` might proceed:

1) Invert `@X` on `pw`, getting `(X*)Y(X*)`.
2) Invert `$Y` on `(X*)Y(X*)`, which requires case analysis.

The second step is complicated in the sense that the inversion depends on the actual string matched by `(X*)Y(X*)` and thus requires a regex that handles cases separately. Even if we handled one step of composition, the complexity grows exponentially as rules are composed. Thus, we opted to limit our inversion computations to simple regular expressions.

B. Guess Counting

We start by sketching `guess_count` for single transformations. Guess counting for composed rules is far more complicated. Similar to rule inversion, the four memory commands are uncountable because they amount to arbitrary computation. Both purge commands are also uncountable. Appendix A gives further details. Note that truncation and substring are only regex-invertible, yet countable.

1) Single Transformation Rules: After initial precomputation on `wlist`, we can count single transformations in time independent of the size of `wlist`. For example, consider `$[0-9]`, which appends one digit. For this rule, `guess_count` outputs $\texttt{num} = 10 \cdot |\texttt{wlist}|$.

A slightly more complicated example is `/?d`, which rejects if the input word does not a contain a digit. This rule will generate *at most* $|\texttt{wlist}|$ guesses since it is filtering words from the list. This is of course easy to calculate by making a pass on `wlist`. Our strategy, however, is to precompute the needed information about such properties to avoid making multiple passes. The 46 JtR transformations we handle were done similarly, sometimes using auxiliary data (see below).

2) Feature Groups Extraction: To enable fast counting, we first collect in `extract_features` the combinations of features that need to be indexed for later counting. Examples include the number of words of each length, the number that contain a digit (or a particular character), and others. Based on the rules, we will later want to quickly look up counts for combinations of these features (e.g. how many words of a given length contain the letter "N," yet no digits). The output of feature extraction is effectively a list of groups of features. The next stage generates a lookup table for constant-time counting of the number of words satisfying any combination of features in the group. A table will be exponentially large in the number of features in a group, so we must limit the size of groups.

Our actual implementation is a compromise between two extremes. The first extreme is to include all features in one group, but for real-world rule lists this produces a table too large to store. The other extreme is to create a group for each rule on its own. Each table will be very small (often between 2 and 16 entries), but populating that many tables is slow. We instead create groups greedily until reaching a threshold size (usually 20 in our experiments), at which point we close that group and start another. As we iterate over rules, we check if the needed feature combination is already contained in some group. If so, we note this and move on without modifying the groups. This intermediate approach generates a small number of groups (e.g. 44, versus thousands in the second extreme) that induce moderately sized look-up tables (e.g. 64MB per table). We use heuristics to process rules containing "popular" transformations first, resulting in more frequent table reuse.

3) Auxiliary Precomputation: For each group $G_1, \ldots, G_n$, we include in the output `aux_info` a multidimensional array $\texttt{A}_i$ of dimension equal to the number of features in G_i. Each dimension of $\texttt{A}_i$ corresponds to one feature. We populate the arrays by iterating over words and incrementing the cell of each table corresponding to the features satisfied by the word.

For example, suppose a feature group has features `length`, `hasdigit`, and `hasB`. We create an array $\texttt{A}_1$, then for each possible length $\ell \in \{1, \ldots, 32\}$, and each possible boolean values $b_{\texttt{hasdigit}}, b_{\texttt{hasB}} \in \{0, 1\}$ we populate $\texttt{A}_1[\ell][b_{\texttt{hasdigit}}][b_{\texttt{hasB}}]$ with the number of words in `wlist` that match that combination of features. When we iterate over words, we check its length and whether it contains a digit and a `B`, incrementing the corresponding cell (e.g., `T9A` increments cell $\texttt{A}_1[3][1][0]$).

4) Full Guess Counting: Guess counting for composed rules is simple only for isolated cases. For example, `$[0-9] $[a-z]` ("append a digit then append a letter") is easy to guess count via the "product rule" as $\texttt{num} = 10 \cdot 26 \cdot |\texttt{wlist}|$. Typical rules, though, have more complicated dependencies and do not obey the product rule. Consider the composed rule `$1 >4 /?d`, which appends 1 and then rejects the guess unless it has length greater than 4 and a digit. Appending 1 obviously generates `wlist` guesses, and we could easily handle `>4` and `/?d` individually via (compact) precomputed data about `wlist`. Reasoning about the composed rule requires tracking how they affect the composition of `wlist`. Intuitively, the process works "backwards" as follows:

1) `/?d`: Remember that only guesses with a digit will count.
2) `>4`: Remember that only guesses of length 5+ will count.
3) `$1`: Modify the previous requirements. Since we append a digit, `/?d` will be satisfied and `>4` becomes `>3`.

Thus, `guess_count` looks up in the appropriate array $\texttt{A}_i$ of `aux_info` how many words in `wlist` are 4+ characters long as each such word generates one guess by this rule.

Our selection of auxiliary data enables efficient guess counting for arbitrary composed rules. Our implementation directly extends the example above. Starting with a rule's rightmost transformation, we modify the state to record how the list will be manipulated by that rule and then propagate left. Finally, we perform a fast table lookup for the counting.

We encoded exactly how every supported transformation will update state. For most rules this was simple, but for others it was moderately complex. For example, the "reflect" rule `f` appends a reversed string to itself (e.g. `Frog` is mapped to `FroggorF`). Applying `f` updates length and the presence and location of characters, both of which our tool handles.

5) Time/Space Analysis: To justify this design, we compare its asymptotic runtime to naive brute-forcing (ignoring log factors coming from data structure implementations and assuming every rule is invertible and countable). The brute-force approach to counting all of `rlist` takes time:

$O(|\texttt{wlist}| \cdot \Sigma_{\texttt{rule}} \texttt{size}(\texttt{rule})) \gg O(|\texttt{wlist}| \cdot |\texttt{rlist}|)$,

where $\texttt{size}(\texttt{rule})$ is the number of guesses `rule` generates per word. In aggregate, this will typically be larger than 1.

In our approach, `extract_features` runs in time $O(|\texttt{rlist}|)$ as the number of relevant features per rule is almost always small. The slowest part of our pipeline is `precompute`, producing data structure `aux_info` of size

$$O(\Sigma_{i=1}^{t} 2^{|G_i|}),$$

assuming the features are binary, where t is the number of tables. After allocating the tables, we spend additional time $O(|\texttt{wlist}| \cdot t)$ to populate their entries (a single pass on the wordlist that increments one cell in each table for each word).

Finally `guess_count` performs a constant-time lookup of the number of transformations in the rule being counted, which we assume is constant (it is typically small). Thus, the end-to-end runtime to compute guess counts for a rule list is

$$O(|\texttt{rlist}| + |\texttt{wlist}| \cdot t + \Sigma_{i=1}^{t} 2^{|G_i|}).$$

When the last term is smaller than $O(|\texttt{rlist}| \cdot |\texttt{wlist}|)$ and t is smaller than $|\texttt{rlist}|$, our method is faster. This rough analysis is only meant to highlight the trade-offs in our design and does not apply when rules are uninvertible. Another inaccuracy comes from cases when `guess_count` does more than a constant-time lookup in a table. For instance, it may need to sum over the table. However, these operations did not dominate processing in practice.

C. Putting it Together in a Guess-Number Calculator

We conclude by showing how to combine rule inversion and guess counting to quickly compute a tight bound on the number of guesses JtR issues before guessing a particular password. We define a function `guess_number` with signature:

`num` ← `guess_number(rlist, cnts, pw)`.

Input `rlist` is a rule list, `pw` is a password for which we want to compute the guess number, and `cnts` is a precomputed array indexed by rules, populated with their guess counts: `cnts[rule]` is the output of `guess_count` for that rule.

Following precomputation of `cnts`, `guess_number` is easy to implement. It initializes `num` ← 0 and iterates over rules in the order specified by `rlist`. For each rule, it uses rule inversion to test if the rule guesses `pw`. If not, it adds the corresponding guess count to `num`. If so, then we know the guess number is between the current value of `num` and that value plus the current rule's guess count. We can either output the range or estimate where in that range the guess occurs.

D. Hashcat Implementation

We implemented `invert_rule` for Hashcat with minor modifications. Since Hashcat guesses in word-major order, knowing which word guesses a password is more important than knowing which rule does. For uninvertible rules, we extend a C implementation of Hashcat's rule engine [43] to enumerate guesses. Hashcat's `guess_count` differs greatly from JtR, though, due to this ordering. Conceptually, Hashcat first applies all rules to the first word. In practice, for I/O efficiency it batches this process, taking A words and B rules at a time. A and B are autotuned based on the hardware.

Because we need the number of guesses induced by each word `w` when B rules are applied, our matrix-based approach to JtR does not apply. We instead compute the number of guesses induced by a word `w` using only `extract_features`, creating one group per rule. We create bit strings to determine whether or not a guess is made by a rule, given a word. If the bit string is all 1s, then a guess is made, and we increment a counter for the word: c_{w_i}. To count the total guesses made by all rules in a batch, we sum the guesses over those rules. Conceptually, this approach is slower than for JtR. However, Hashcat's standard mode does not support rejection rules except in special cases [41], and Hashcat does not support character classes (e.g., JtR's `?d` representing any digit) except in hybrid mode. Thus, guess counting is often trivial as each rule makes $|\texttt{wlist}|$ guesses.

V. Evaluation Data Sets

We analyze our techniques on three wordlists, six rule lists, and six evaluation sets of passwords. We use these evaluation sets alongside our analytical engine to tune the configuration (both order and completeness) of the rule lists and wordlists.

a) Wordlists: While JtR and Hashcat include sample wordlists, they are far smaller than those used in typical attacks. Therefore, we selected three wordlists containing password data and natural-language dictionaries that are more typical of those used by experienced attackers [15]. **XATO** is a set of 10 million passwords (5,189,378 unique entries) sampled from thousands of leaked sets and released by a security consultant in 2015 [44]. It has been used previously in research [22]. **PGS** (19,436,159 entries) is a combination of passwords and dictionaries used in CMU's Password Guessability Service [8]. Lastly, **LinkedIn** is a set of passwords (60,169,992 unique entries) from the LinkedIn data breach [45]. While hashes, not plaintext, were leaked, over 97% have been cracked. For XATO and LinkedIn, we took the initial set, cleaned non-ASCII characters, and sorted them by frequency, removing duplicates. PGS was already ordered.

b) Rule Lists: Since JtR and Hashcat do not use the same rule language, we selected three typical rule lists for each. The first three lists are for JtR. **John** (151 rules) represents JtR's default rules. **SpiderLabs** (5,146 rules) is a version of a sample list KoreLogic released for a password-cracking contest reordered by a human expert [16] in order of anticipated effectiveness [18]. **Megatron** (15,329 rules) combines the two sets of m3g9tr0n rules from Openwall [19]. The next three lists are for Hashcat, and all come with the Hashcat software. **Best64** (77 rules) functions as Hashcat's default best rules and was created and refined in community contests. **T0XlC** (4,085 rules) and **Generated2** (65,117 rules) are more extensive sets created by members of Team Hashcat.

c) Evaluation Sets: We evaluate our techniques on six sets of passwords (Table II). We chose four sets that represent password-composition policies and characteristics typically seen in leaked password sets. We also chose to include one

TABLE II: Description of evaluation sets.

Source	# Passwords	Apparent Requirements
000webhost [3]	13 million	length $\geq$ 6 with letter & digit
Battlefield Heroes [49]	500,000	length $\geq$ 6
Brazzers [50]	800,000	length $\geq$ 6
Clixsense [51]	2.2 million	length $\geq$ 6
CSDN [4]	6 million	length $\geq$ 8
Neopets [52]	70 million	length $\geq$ 6

set with a more strict composition policy (000webhost) and one non-English, Chinese set (CSDN), whose characteristics have been found to differ from English-language sets [46]. All sets were leaked in plaintext except for Battlefield Heroes, of which over 99% has been cracked. Because UTF-8 support in cracking software introduces subtleties [47] or is missing entirely [48], we converted non-ASCII characters to ASCII. Because cracking software limits the length of guesses, we removed lines longer than 32 characters. This cleaning removed at most 0.6% of passwords per set. We also removed lines that did not comply with the composition policy, which may be legacy accounts or errors. For all sets besides Neopets, this removed under 2% of passwords. So sets would be of equal size, we randomly sampled 25,000 passwords from each set.

VI. Ethics

Our evaluation sets and some of our wordlists contain passwords that were previously stolen and then leaked online. Using this data raises ethical questions. We clean the data of everything other than passwords, meaning that there is no identifying information in the data we analyze. Furthermore, these password lists are already available publicly online, so the harm already caused to users is not exacerbated by our use of the data. Lastly, the guess-number calculator we develop (Section VII) enables real-time password checking, which we anticipate will help users make more secure passwords.

Our techniques enable data-driven optimization of the order and completeness of rule lists and wordlists (Sections VIII–XI). While attackers could use our techniques to improve attacks, we do not believe that releasing our tools substantially advantages attackers. Members of the cracking community have already invested massive amounts of computation in developing and refining their own rule lists and wordlists [38]. Experienced attackers' curated lists are closely guarded secrets and are rarely shared publicly. In password-cracking contests [17], prior academic evaluations [8], and media articles [5], they substantially outperform lists released publicly. While some attackers might benefit from our tools, we expect our tools primarily to better align the academic community's models with experts' non-public lists and configurations.

VII. Evaluation of a Guess-Number Calculator

As detailed in Section IV-B, our analytical tools can be applied directly to generate a given password's approximate guess number (the number of guesses it would take that approach in that configuration to guess that password). Here, we evaluate the proportion of rules in our six evaluation rule lists that can be reasoned about analytically. We then present performance benchmarks showing orders-of-magnitude improvement in the time it takes to compute guess numbers for common rule lists. With less than a day of pre-processing on a commodity machine, we can generate guess numbers for a given password in under one second for most lists we evaluate.

TABLE III: Fraction of rule lists that are invertible/countable.

Rule List	Rules	Invertible, Countable	Invertible, Uncountable	Brute-force Enumerated
John the Ripper				
SpiderLabs	5,146	5,146 (100%)	0 (0%)	0 (0%)
Megatron	15,329	14,840 (97%)	467 (3%)	22 (0%)
John	145	89 (61%)	0 (0%)	56 (39%)
Hashcat				
T0XlC	4,085	3,980 (97%)	0 (0%)	105 (3%)
Generated2	65,117	50,781 (78%)	831 (1%)	13,505 (21%)
Best64	77	62 (81%)	0 (0%)	15 (19%)

A. Breadth of Application

Rules that are both fully invertible and countable provide the greatest benefit for our approach because they can be analyzed completely using our analytical tools, without enumerating any guesses. Table III shows the fraction of each of our six evaluation rule lists that are both invertible and countable. The SpiderLabs rule list for JtR contains 5,146 rules, all of which are both fully invertible and countable. Similarly, Megatron contains 14,840 rules, of which over 97% are both fully invertible and countable. These two large lists benefit substantially from our approach. John, however, contains only 145 rules, and only 61% are both fully invertible and countable.

For Hashcat, the T0XlC rule list contains 4,085 rules. Because 97% of these are both invertible and countable, one would again expect to see notable performance benefits from our analytical approach. In contrast, only 78% of the 65,117 Generated2 rules are both invertible and countable, and only 81% of the 77 Best64 rules are both invertible and countable, so the benefits of our analytical approach are more muted.

B. Performance Benchmarks

To gauge whether our analytical approach's conceptual advantages translate to reality, we benchmarked a Python implementation of our approach and compared it with estimates of naively enumerating guesses. We used a commodity server with an Intel Core i7-4770 CPU (3.40GHz, 4 cores), 32 GB of RAM, and 7200 RPM hard disks in RAID 1+0. We used John the Ripper 1.8.0 and Hashcat v3.6.0. We calculated guess numbers for all passwords in the 000webhost evaluation set using a small (XATO) and a large (LinkedIn) wordlist.

We observed a performance benefit of many orders of magnitude for the JtR SpiderLabs rule list. Using the LinkedIn wordlist, this configuration incurred a one-time cost of 16 hours of pre-processing. Subsequently, calculating a given password's guess number took 0.367 seconds on average, and a maximum of 2.074 seconds. In contrast, enumerating the 3.01×10^{14} guesses this configuration makes would have taken approximately 4.7 years based on our benchmarked throughput of ~12 million guesses per second piping JtR's debug mode

TABLE IV: Performance comparison between naively enumerating guesses and our analytical approach. For enumeration, we estimate the uncompressed *size* on disk for storing all guesses, as well as the *time* in seconds to pipe all guesses to stdout. For the analytical approach, we present the *pre-processing* time (a one-time cost), as well as mean and max times for computing a guess number in our tests. We also give the total *size* on disk of the auxiliary data, wordlist, and any enumerated guesses.

			Enumerating Guesses		Our Analytical Approach			
Rule List	**Wordlist**	**# Guesses**	**Size**	**Time (s)**	**Size**	**Pre-Processing (s)**	**Mean Lookup (s)**	**Max Lookup (s)**
SpiderLabs (JtR)	LinkedIn	3.01×10^{14}	3.3 PB	1.51×10^{8}	10.2 GB	5.85×10^{4}	0.367	2.074
SpiderLabs (JtR)	XATO	2.79×10^{13}	306.9 TB	1.39×10^{7}	4.9 GB	5.03×10^{3}	0.367	2.011
Megatron (JtR)	LinkedIn	7.27×10^{11}	8.0 TB	3.64×10^{5}	12.8 GB	1.04×10^{4}	0.718	2.536
Megatron (JtR)	XATO	5.63×10^{10}	619.3 GB	2.82×10^{4}	1.1 GB	9.69×10^{2}	0.712	1.623
John (JtR)	LinkedIn	3.57×10^{10}	392.7 GB	1.78×10^{4}	145.7 GB	3.08×10^{4}	0.133	2.846
John (JtR)	XATO	3.05×10^{9}	33.6 GB	1.53×10^{3}	12.9 GB	2.71×10^{3}	0.117	0.406
T0XlC (HC)	LinkedIn	2.46×10^{11}	2.7 TB	2.46×10^{4}	112.6 GB	1.79×10^{3}	0.073	0.908
T0XlC (HC)	XATO	2.12×10^{10}	233.4 GB	2.12×10^{3}	8.7 GB	1.25×10^{2}	0.071	0.388
generated2 (HC)	LinkedIn	3.92×10^{12}	43.2 TB	3.92×10^{5}	9.8 TB	2.24×10^{5}	13.604	27.940
generated2 (HC)	XATO	3.38×10^{11}	3.7 TB	3.38×10^{4}	754.1 GB	1.75×10^{4}	13.005	26.175
Best64 (HC)	LinkedIn	4.03×10^{9}	44.4 GB	4.03×10^{2}	15.0 GB	5.77×10^{1}	0.039	0.802
Best64 (HC)	XATO	3.48×10^{8}	3.8 GB	3.48×10^{1}	1.2 GB	5.00×10^{0}	0.038	0.120

to stdout. While one could imagine enumerating guesses once, writing them to disk, and sorting them to enable fast lookups, doing so requires ~3.3 petabytes of disk (uncompressed).

We also observed performance benefits, albeit smaller in magnitude, for JtR's Megatron rule lists and all three Hashcat rule lists. Using the LinkedIn wordlist, our approach calculated guess numbers in an average of 0.718 seconds for JtR's Megatron list and 0.073 seconds for Hashcat's T0XlC list. These approaches required pre-processing of 2.9 hours and 0.5 hours, respectively. This cost is amortized over looking up guess numbers for many passwords. In contrast, writing enumerated guesses to disk would require 8.0 terabytes and 2.7 terabytes (uncompressed), respectively. For fast lookups in this naive approach, this data would also need to be sorted.

The benefits of the analytical approach were not fully universal, however. For the smallest JtR rule list (John), the pre-processing time exceeded the time to enumerate (but not sort) guesses. The analytical approach nonetheless enables fast lookups (mean of 0.133 seconds for LinkedIn) with smaller storage requirements. At a high level, the analytical approach provided substantial performance benefits for large rule lists in which the vast majority of rules were invertible and countable, particularly when paired with large wordlists.

Finally, our approach is highly accurate. We verified this by randomly generating rules and comparing our analytical evaluation to enumerated guesses, as detailed in Appendix B.

VIII. Optimization 1: Ordering Rule Lists

Because JtR generates guesses in rule-major order (first applying the first rule in its rule list to all words), the order of rules is critical for JtR. Many publicly released rule lists have been ordered through a combination of human intuition [18] and empirical experiments [38] that are mostly limited and undocumented. In such experiments, one collects or creates large sets of rules, reordering the rules in descending order of observed success against an evaluation set. Doing so by enumerating guesses is highly time- and computation-intensive. We use our novel analytical tools to do so efficiently.

TABLE V: Using the PGS wordlist, a comparison of the original position of the first 15 John rules (excluding guessing words verbatim) and their position after reordering based on 000webhost (*000wh*), CSDN, and the four English-language sets with identical password-composition policies (*Others*).

JtR Rule	Original	000wh	CSDN	Others
`-s x**`	1	93	85	109 – 110
`-c (?a c Q`	2	9	33	12 – 16
`-c l Q`	3	4	5	1 – 3
`-s-c x** /?u l`	4	94	86	110 – 111
`>6 '6`	5	5	87	3 – 7
`>7 '7 l`	6	6	88	8 – 12
`-c >6 '6 /?u l`	7	3	89	2 – 25
`>5 '5`	8	95	90	111 – 112
`/?d @?d >4`	9	96	3	6 – 23
`/?d @?d M @?A Q >4`	10	97	10	6 – 22
`/?d @?d >4 M [lc] Q`	11	98	29	33 – 38
`/?d @?d M @?A Q >4 M [lc] Q`	12	99	91	1 – 92
`@?D Q >4`	13	100	1	1 – 4
`/?d @?d >3 <* $[0-9] Q`	14	23	31	21 – 30
`/?d @?d M >3 <* [lc] Q $[0-9] Q`	15	54	54	44 – 70

A. Approach

We reorder rules in descending order of *success density*, defined as the ratio of a rule's successful guesses (those matching a password in evaluation set S) to the total number of guesses. To avoid prioritizing rules whose successful guesses overlap with those of a previously prioritized rule, we reorder iteratively. We assume attackers first guess all items in the wordlist verbatim, which is often the best strategy [8]. We then calculate the success density for each rule against the evaluation set S, placing the rule with the highest success density next. We remove all passwords guessed by that rule from S, recalculating the success density for all remaining rules. We repeat this process until all rules have been ordered. In the case of ties, we prioritize the rule that made fewer guesses. If the guesses made by rules are fully disjoint, this strategy is provably optimal in maximizing the area under the guessing curve. If the guesses are not disjoint, one can construct pathological cases where this strategy is not optimal, yet it is far more computationally tractable than alternatives.

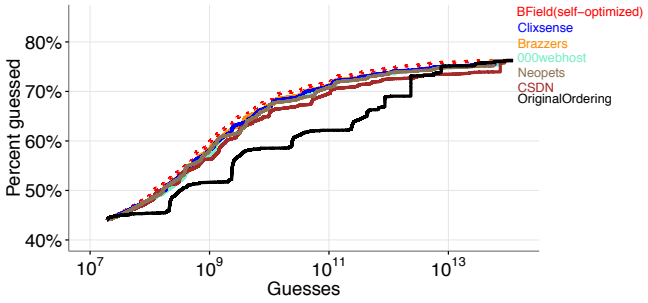


Fig. 1: The guessability of Battlefield Heroes under JtR using the PGS wordlist and the SpiderLabs rule list in its original order, reordered artificially based on itself (*BField*), and reordered based on each of the five other evaluation sets. Each reordering substantially improved on the original ordering.

B. Evaluation Procedure and Results

Using each of our three wordlists, we applied this approach to reorder each of the three JtR rule lists using each of the six evaluation sets in turn. Comparing rules' original positions to their reordered position, we found a number of rules whose positions after reordering diverged consistently from their original position in these widely distributed rule lists.

As Table V demonstrates, our reordering process consistently suggests that some rules that appear early in the 145-rule John list likely belong late in the list, and vice versa. For example, three of the first ten rules never appear earlier than the 85th position after reordering based on success density for any of our six evaluation sets. Notably, John is widely distributed in this non-optimal form as JtR's default rule list.

We found trends that were similar, if not more stark, for the other rule lists. For example, in the 5,146-rule SpiderLabs rule list, none of the first 23 rules still appear within the top 100 rules after reordering based on any of our six evaluation sets. This is particularly notable because SpiderLabs was already manually reordered based on a password-cracking expert's intuition [18]. Appendix E shows the original and reordered positions of the first 100 rules for two of the JtR rule lists.

Across the three rule lists, we observed that using any of our four English-language evaluation sets with identical password-composition policies – Battlefield Heroes, Brazzers, Clixsense, and Neopets – resulted in rule reorderings that were similar to each other. We therefore group these four sets together in our tables. The reorderings tended to be distinct, however, for both the 000webhost set, whose composition policy required a digit, and the Chinese-language CSDN set.

Crucially, we found that our reordering procedure enables many passwords to be guessed much earlier in an attack for both the SpiderLabs and John rule lists. Figure 1 shows the guessability of the Battlefield Heroes evaluation set using the SpiderLabs rules in their original order (black line), artificially reordered on itself as an upper bound (dashed red line), and reordered based on each of the five other evaluation sets. Reordering the SpiderLabs rule list based on any of the four other English-language sets provides substantial improvements in guessing. While rule reordering does not change which passwords are guessed, a substantially larger fraction of passwords are guessed earlier in the attack compared to the original ordering. Notably, reordering based on any of the other four English-language sets results in guessing success approaching the artificial self-optimized ordering, highlighting that reorderings generalize well across sets. While reordering based on the Chinese-language CSDN set also provides substantial improvements over the original ordering, it performs less well than any of the English-language sets. We observed very similar trends for each of the five other evaluation sets.

We also observed similar trends reordering the much smaller John rule list for all six evaluation sets. For the Megatron rule list, however, we observed a different trend. As shown in Figure 4a in Appendix E, the original ordering of the Megatron rule list (black) is already relatively close to the artificial self-optimized reordering (red). Guessing Battlefield Heroes passwords using any of the four other English-language evaluation sets results in an attack that performs very similarly to the original ordering, while reordering based on the Chinese-language CSDN set results in a less effective attack. While Megatron's original ordering already seems near optimal, our approach let us demonstrate this scientifically.

IX. Optimization 2: Ordering Wordlists

Given the performance benefits we observed reordering rule lists for JtR, which guesses in rule-major order, we then used our analytical tools to reorder wordlists for Hashcat, which guesses in word-major order. We had hoped reordering wordlists based on evaluation sets would improve guessing performance on other sets, yet instead found this process to worsen performance. Wordlists are typically already ordered in descending frequency based on myriad prior password leaks, and our data-driven optimization seemed to overfit. We nonetheless describe our process because it might prove more effective if extremely large evaluation sets were used.

A. Approach

We begin with a wordlist `wlist` containing words `w`. We term `wlist` in its initial order $\texttt{wlist}_{original}$. Given an evaluation set S, we run `invert_rule` on each password `pw` in S to identify which passwords would be guessed, and by which words. We split the wordlist `wlist` in two: one wordlist, $\texttt{wlist}_{success}$, containing words $\texttt{w}_i$ that would guess at least one password `pw` in S, and $\texttt{wlist}_{failure}$, containing the remaining words. We rearrange $\texttt{wlist}_{success}$ in descending order of the number of passwords in S that each would guess, breaking ties arbitrarily. We append $\texttt{wlist}_{failure}$, maintaining the order from the original `wlist`. This combined wordlist, optimized on evaluation set S, is termed $\texttt{wlist}_S$.

B. Evaluation Procedure and Results

We used this approach to reorder both the XATO wordlist, which contains only passwords, and the PGS wordlist, which contains passwords followed by natural language dictionaries. We did so for each of the six evaluation sets. Compared

to the original, reordering a wordlist based on one evaluation set decreased guessing performance substantially for all other evaluation sets (Figure 4b in Appendix E). Data-driven wordlist reordering for Hashcat appears to overfit and would not be recommended, at least for small evaluation sets.

X. Optimization 3: Rule List Completeness

Some prior work has automatically generated new rules and then enumerated their guesses to test their effectiveness cracking evaluation sets [38], [39]. We similarly generate new rules, yet use our analytical tools to reason efficiently about their effectiveness. Adding these potentially "missing" rules enable more passwords to be guessed, as well as more quickly.

A. Approach

The space of possible rules is huge. By randomly or comprehensively generating rules, we extend a wordlist `wlist` with rules it does not already contain. We then reorder this extended list based on other evaluation sets as in Section VIII.

B. Evaluation Procedure and Results

Using each of our three wordlists in turn with the JtR SpiderLabs rule list, we tested on the four English-language evaluation sets with identical composition policies. First, we reordered SpiderLabs based on the three other evaluation sets (termed *reordered*). We then generated the 15,085 JtR rules that consist of a single transformation and are both invertible and countable. Adding those to the SpiderLabs rule list (*extended*), we followed the same reordering procedure and cut off guessing at the previous number of guesses.

This procedure identified rules that are both new and effective. Figure 2 shows the guessability of both Battlefield Heroes and Brazzers. Compared to the original ordering (light colors), reordering based on other evaluation sets (mid colors) leads to passwords being guessed more quickly, echoing Section VIII. However, extending SpiderLabs and then reordering it (dark colors) leads to passwords being guessed even more quickly, as well as previously unguessed passwords being guessed. Results were similar across the evaluation sets.

We then analyzed the new rules. Cutting off at the same number of guesses as the original SpiderLabs with Battlefield Heroes as the evaluation set, we observed 3,495 new rules having been executed in the extended attack. While the original SpiderLabs contained only 5,146 rules, many of them utilized JtR's rule preprocessor to make a large number of guesses in a single rule (e.g., appending a digit). We found that 178 of the newly identified rules were strict subsets of an existing SpiderLabs rule (e.g., appending a specific digit) that had a higher success density than the superset rule. Another 115 new rules were either contained verbatim in John or Megatron, or they were strict subsets of a rule in those lists. The remaining 3,202 rules were completely new to our three JtR rule lists.

XI. Optimization 4: Wordlist Completeness

Our `invert_rule` process moves backwards from passwords to the preimages that, when transformed, guess that password. We modified this process to identify what we term *missing words*, or words that perhaps should have been in the wordlist based on a given evaluation set. The intuition is to leverage "cache misses" to improve wordlist completeness.

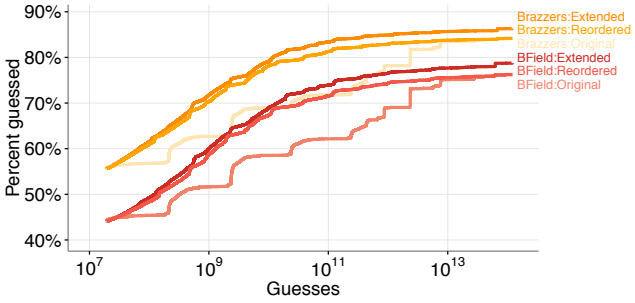


Fig. 2: The guessability of Battlefield Heroes and Brazzers using the PGS wordlist and the JtR SpiderLabs rule list in its original order (light), reordered based on the other sets (mid), and extended with "missing" rules and then reordered (dark).

A. Approach

Given an evaluation set S, a wordlist `wlist`, and a (reordered) rule list `rlist`, we use `invert_rule` to invert each password `pw` $\in S$ to generate preimages p_i. Each preimage not in the wordlist (i.e., $p_i \notin$ `wlist`) is a potential missing word. To identify preimages likely to generalize, for each unique password `pw` $\in S$, we assign a credit $c \in [0, 1]$ to each potential preimage $p_i \notin$ `wlist` inversely proportional to the rule's position in `rlist`. A preimage identified with the first rule in `rlist` will receive $c = 1$, while one identified with the middle rule will receive $c = 0.5$. This approach prioritizes preimages used early in an attack. For a particular password, credit for a particular preimage is given only once.

After following this process for all unique passwords in set S, we rank preimages in descending order of credit summed across passwords, keeping those above a threshold.

B. Evaluation Procedure and Results

We follow this procedure for all six evaluation sets using the fully invertible SpiderLabs rule list and both the LinkedIn and PGS wordlists. We use the rule ordering self-optimized for each evaluation set (Section VIII). To emphasize new cracks, we use only the passwords in each evaluation set that would not otherwise be guessed by a given wordlist and rule list.

Table VI presents a manual thematic categorization of the 100 preimages with the highest credit for the 000webhost, Battlefield Heroes, and Neopets evaluation sets. This process produces site-specific words (e.g., "bfheroes"), meaningful strings unrelated to the site, and short (2–3 character) strings.

To understand whether this procedure results in more effective guessing in realistic scenarios (i.e., testing on sets not used for optimization), we first used the four English-language evaluation sets with identical password policies to identify words potentially missing from the PGS wordlist, which includes both passwords and natural language. To test the impact on guessing, we used a random sample of 500,000

TABLE VI: A manual categorization of the top 100 preimages identified as potentially "missing" from the PGS wordlist.

Category	Examples	000webhost	BField	Neopets
Set-specific	bfheroes; ilovmyneopets""""	9	7	3
Meaningful	la la la; Son-gouku; MaSterBrain	34	73	68
Short strings	a2; a23; 7a; b2; q2; 2k;	18	6	3
Unidentified	gawabint1; kur0=ud1	39	14	26

passwords from each of the other three sets to generate the top million words "missing" from PGS. We modeled an attack using SpiderLabs rules and the missing words as the wordlist.

In each case, this attack made 1.7×10^{13} extra guesses, successfully guessing 221 Clixsense passwords, 157 Battlefield Heroes passwords, 128 Brazzers passwords, and 118 Neopets passwords from our 25,000-password evaluation sets. None of these passwords would have been guessed otherwise by SpiderLabs. While the success density of such attacks is low, they are appropriate at the end of an attack when high-probability guesses have been exhausted. For comparison, the final 1.7×10^{13} guesses with the PGS wordlist and SpiderLabs rule list (reordered on the three other sets combined) resulted in zero successful guesses for any of the four test sets.

XII. Comparisons to Existing Algorithms / Meters

Our analytical techniques enable two primary applications: proactive password checking and data-driven configuration (improvement) of transformation-based attacks. Here, we analyze these applications relative to prior approaches.

First, our guess-number calculator enables real-time password checking, which is effectively a server-side password meter. Here, we highlight two experiments comparing our approach to meters using combinatoric estimates (zxcvbn) [22] and Neural Networks [13]. Appendix C expands on both.

Following best practices in comparing meters [53], we examined how meters' guess numbers for a given password were correlated with the number of times that password appeared in an evaluation set. As shown in Table VII, our analytical JtR and Hashcat approaches are better correlated with the frequency counts than existing meters. Correlations approaching 1 indicate better alignment with frequency counts. For example, for the Brazzers set, JtR had a correlation of 0.734 and Hashcat had a correlation of 0.731, compared to 0.693 and 0.702 for zxcvbn and Neural Networks, respectively. However, while existing meters estimate a guess number for every password, our approach assigns the same large guess number to any passwords unguessed by JtR or Hashcat.

Unlike *any* prior meter, ours is the first to provide real-time models of guessing attacks widely used in the wild. To evaluate whether existing meters already fully captured these attacks, we examined whether those meters made *unsafe errors*, rating guessable passwords as strong. Reflecting real attacks, we rated the 25% of each password set with the lowest guess numbers (guessed first) for Hashcat and JtR as *practically weak*. As shown in Table VIII, all meters rated at least some practically weak passwords among the 25% of hardest-to-guess passwords. While this represents only a fraction of a percent of passwords in the set, their strength estimates differed radically from real attacks.

TABLE VII: Coverage (%) and accuracy (r_w, weighted Spearman correlation) of our approach and existing meters.

Meter	000webhost		Brazzers		Neopets	
	%	r_w	%	r_w	%	r_w
Hashcat (Sec. VII)	40.1	0.512	83.3	0.731	77.8	0.806
JtR: Extended (Sec. X)	40.3	0.515	79.4	0.734	76.4	0.805
Neural Network [13]	100.0	0.507	100.0	0.702	100.0	0.795
zxcvbn [22]	100.0	0.437	100.0	0.693	100.0	0.696

TABLE VIII: The number of passwords (of 25,000) in each set that were among the 25% easiest to guess by JtR or Hashcat, yet rated among the 25% hardest to guess by a given meter.

Meter	000webhost	BField	Brazzers	Clixsense	CSDN	Neopets
Markov: Multi [53]	11	16	19	22	24	34
PCFG: 2016 [54]	15	0	0	0	18	1
Neural Network [13]	11	12	19	21	23	28
zxcvbn [22]	21	39	35	84	22	32
zxcvbn/LinkedIn-30k	19	33	14	26	22	30

Second, our optimization techniques improve the efficacy of transformation-based attacks. As detailed in Appendix D, we compared JtR both pre- and post- optimization, as well as Hashcat, to the probabilistic approaches. Echoing prior work [8], [13], Neural Networks performed best on a guess-by-guess basis, and probabilistic approaches often outperformed pre-optimization JtR and Hashcat. However, at 10^9 guesses, post-optimization JtR performed similarly to, or better than, all approaches other than Neural Networks for four of the six evaluation sets. Our results suggest that JtR and Hashcat, if configured using techniques we propose, may not lag as far behind probabilistic approaches as previously thought. Furthermore, JtR and Hashcat have practical advantages; they generate guesses far more quickly than probabilistic approaches.

XIII. Conclusions and Discussion

We have presented some of the first techniques for principled, scientific analysis of popular password-cracking software's most common attack, the mangled-wordlist attack. Our tools provide the first computationally efficient analysis of password security against the types of mangled-wordlist attacks actually performed in the wild. We also showed how our techniques enable four data-driven optimizations to improve the ordering and completeness of rule lists and wordlists, better aligning our models with experienced attackers' non-public (and effective) configurations.

Our tools, which we are releasing open-source,[2] directly enable real-time, server-side estimation of password strength. A company could deploy our guess-number calculator, disallowing passwords deemed guessable. Prior work has found that, when discouraged [29] or forbidden [7] from using a weak password, users rarely pick the next most probable password permitted. Future empirical studies are needed to capture humans' adaptation mechanisms [55]; attackers could perhaps encode these behaviors in new transformation rules.

[2]https://github.com/UChicagoSUPERgroup/analytic-password-cracking

REFERENCES

[1] J. Bonneau, "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords," in *Proc. IEEE S&P*, 2012.

[2] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring Password Guessability for an Entire University," in *Proc. CCS*, 2013.

[3] D. Goodin, "13 Million Plaintext Passwords Belonging to Webhost Users Leaked Online," October 28, 2015, https://arstechnica.com/information-technology/2015/10/13-million-plaintext-passwords-belonging-to-webhost-users-leaked-online/.

[4] X. Yang, "Chinese Internet Suffers the Most Serious User Data Leak in History," *Forcepoint Blog*, December 26, 2011, https://blogs.forcepoint.com/security-labs/chinese-internet-suffers-most-serious-user-data-leak-history.

[5] D. Goodin, "Why Passwords Have Never Been Weaker – And Crackers Have Never Been Stronger," *Ars Technica*, August 20, 2012, http://arstechnica.com/security/2012/08/passwords-under-assault/.

[6] C. Castelluccia, M. Dürmuth, and D. Perito, "Adaptive Password-Strength Meters from Markov Models," in *Proc. NDSS*, 2012.

[7] P. Kelley, S. Kom, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. López, "Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms," in *Proc. IEEE S&P*, 2012.

[8] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay, "Measuring Real-World Accuracies and Biases in Modeling Password Guessability," in *Proc. USENIX Security*, 2015.

[9] M. Weir, S. Aggarwal, M. Collins, and H. Stern, "Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords," in *Proc. CCS*, 2010.

[10] M. Dell'Amico and M. Filippone, "Monte Carlo Strength Evaluation: Fast and Reliable Password Checking," in *Proc. CCS*, 2015.

[11] J. Ma, W. Yang, M. Luo, and N. Li, "A Study of Probabilistic Password Models," in *Proc. IEEE S&P*, 2014.

[12] M. Dürmuth, F. Angelstorf, C. Castelluccia, D. Perito, and A. Chaabane, "OMEN: Faster Password Guessing Using an Ordered Markov Enumerator," in *Proc. ESSoS*, 2015.

[13] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor, "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks," in *Proc. USENIX Security*, 2016.

[14] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek, "Password Cracking Using Probabilistic Context-Free Grammars," in *Proc. IEEE S&P*, 2009.

[15] D. Goodin, "Anatomy of a Hack: How Crackers Ransack Passwords Like "qeadzcwrsfxv1331"," *Ars Technica*, May 27, 2013, http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/.

[16] Trustwave SpiderLabs, "SpiderLabs/KoreLogic-Rules," Sep. 2012, https://github.com/SpiderLabs/KoreLogic-Rules.

[17] KoreLogic, "Crack me if you can," 2018, https://contest.korelogic.com/.

[18] G. Picchioni, "Hey, I Just Met You, and This is Crazy, But Here's My Hashes, So Hack Me Maybe?" *SpiderLabs Blog*, Sept. 25, 2012, https://www.trustwave.com/Resources/SpiderLabs-Blog/Hey,-I-just-met-you,-and-this-is-crazy,-but-here-s-my-hashes,-so-hack-me-maybe-/.

[19] m3g9tr0n, "Cracking Story - How I Cracked Over 122 Million SHA1 and MD5 Hashed Passwords," *Thireus' Bl0g*, August 28, 2012, https://blog.thireus.com/cracking-story-how-i-cracked-over-122-million-sha1-and-md5-hashed-passwords/.

[20] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The Tangled Web of Password Reuse," in *Proc. NDSS*, 2014.

[21] A. Greenberg, "The police tool that pervs use to steal nude pics from Apple's iCloud," *Wired*, September 2, 2014, https://www.wired.com/2014/09/eppb-icloud/.

[22] D. L. Wheeler, "zxcvbn: Low-Budget Password Strength Estimation," in *Proc. USENIX Security*, 2016.

[23] H.-C. Chou, H.-C. Lee, H.-J. Yu, F.-P. Lai, K.-H. Huang, and C.-W. Hsueh, "Password Cracking Based On Learned Patterns From Disclosed Passwords," *IJICIC*, vol. 9, no. 2, pp. 821–839, 2013.

[24] M. Dürmuth, A. Chaabane, D. Perito, and C. Castelluccia, "When Privacy Meets Security: Leveraging Personal Information for Password Cracking," *CoRR*, vol. abs/1304.6584, pp. 1–19, Apr. 2013.

[25] R. Shay, S. Komanduri, A. L. Durity, P. S. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor, "Can Long Passwords Be Secure and Usable?" in *Proc. CHI*, 2014.

[26] Y. Zhang, F. Monrose, and M. K. Reiter, "The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis," in *Proc. CCS*, 2010.

[27] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password Strength: An Empirical Analysis," in *Proc. INFOCOM*, 2010.

[28] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor, "How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation," in *Proc. USENIX Security*, 2012.

[29] B. Ur, F. Alfieri, M. Aung, L. Bauer, N. Christin, J. Colnago, L. F. Cranor, H. Dixon, P. E. Naeini, H. Habib, N. Johnson, and W. Melicher, "Design and Evaluation of a Data-Driven Password Meter," in *Proc. CHI*, 2017.

[30] T. Hunt, "Have I been pwned?" 2018, https://haveibeenpwned.com/.

[31] A. Narayanan and V. Shmatikov, "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff," in *Proc. CCS*, 2005.

[32] R. Veras, C. Collins, and J. Thorpe, "On the Semantic Patterns of Passwords and their Security Impact," in *Proc. NDSS*, 2014.

[33] Y. Chrysanthou, "Modern Password Cracking: A Hands-On Approach to Creating an Optimised and Versatile Attack," Master's thesis, Royal Holloway, University of London, 2013.

[34] A. Forget, S. Chiasson, P. C. van Oorschot, and R. Biddle, "Improving Text Passwords Through Persuasion," in *Proc. SOUPS*, 2008.

[35] S. Fahl, M. Harbach, Y. Acar, and M. Smith, "On the Ecological Validity of a Password Study," in *Proc. SOUPS*, 2013.

[36] X. de Carné de Carnavalet and M. Mannan, "From Very Weak to Very Strong: Analyzing Password-Strength Meters," in *Proc. NDSS*, 2014.

[37] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted Online Password Guessing: An Underestimated Threat," in *Proc. CCS*, 2016.

[38] S. Marechal, "Automatic Mangling Rules Generation," *Passwords '12*, 2012, http://www.openwall.com/presentations/Passwords12-Mangling-Rules-Generation/.

[39] P. Kacherginsky, "Smarter Password Cracking with PACK," *Passwords '13*, 2013, http://thesprawl.org/research/automatic-password-rule-analysis-generation/.

[40] Openwall, "Wordlist Rules Syntax," 2018, https://www.openwall.com/john/doc/RULES.shtml.

[41] Hashcat, https://hashcat.net/wiki/doku.php?id=rule_based_attack.

[42] Y. Baburov, "python-chartrie," https://github.com/buriy/python-chartrie.

[43] Llamasoft, https://github.com/llamasoft/HashcatRulesEngine.

[44] M. Burnett, "Ten Million Passwords FAQ," February 10, 2015, https://xato.net/ten-million-passwords-faq-3b2752ed3b4c.

[45] S. Perez, "117 Million LinkedIn Emails and Passwords From a 2012 Hack Just Got Posted Online," *TechCrunch*, May 18, 2016, http://tcrn.ch/23Xcd6R.

[46] Z. Li, W. Han, and W. Xu, "A Large-Scale Empirical Analysis of Chinese Web Passwords," in *Proc. USENIX Security*, 2014.

[47] Hashcat Forum, https://hashcat.net/forum/thread-5486.html.

[48] Openwall, "John the Ripper's Cracking Modes," http://www.openwall.com/john/doc/MODES.shtml.

[49] J. Walker, "LulzSec Over, Release Battlefield Heroes Data," *Rock Paper Shotgun*, June 26, 2011, https://www.rockpapershotgun.com/2011/06/26/lulzsec-over-release-battlefield-heroes-data/.

[50] J. Cox, "Nearly 800,000 Brazzers Porn Site Accounts Exposed in Forum Hack," *Vice Motherboard*, September 5, 2016, https://motherboard.vice.com/en_us/article/vv7pgd/nearly-800000-brazzers-porn-site-accounts-exposed-in-forum-hack.

[51] D. Goodin, "6.6 Million Plaintext Passwords Exposed as Site Gets Hacked to the Bone," *Ars Technica*, September 13, 2016, https://arstechnica.com/information-technology/2016/09/plaintext-passwords-and-wealth-of-other-data-for-6-6-million-people-go-public/.

[52] J. Cox, "Another Day, Another Hack: Tens of Millions of Neopets Accounts," *Vice Motherboard*, May 5, 2016, https://motherboard.vice.com/en_us/article/ezpvw7/neopets-hack-another-day-another-hack-tens-of-millions-of-neopets-accounts.

[53] M. Golla and M. Dürmuth, "On the Accuracy of Password Strength Meters," in *Proc. CCS*, 2018.

[54] S. Komanduri, "Modeling the Adversary to Evaluate Password Strength with Limited Samples," Ph.D. dissertation, CMU, 2016.

[55] M. Wei, M. Golla, and B. Ur, "The Password Doesn't Fall Far: How Service Influences Password Choice," in *Proc. WAY*, 2018.

APPENDIX

A. *Detailed Explanations of Handling Particular Transformation Rules*

In Table I, we claimed that a handful of transformation rules are not fully invertible or are not countable. Here, we detail why this is the case. We also justify our decision to enumerate guesses for rules that are only regex-invertible.

1) The Complexity of Memory Commands: JtR supports memory commands that memorize a string (`M`), query the memory and reject the word if unchanged (`Q`), insert substrings from memory into a guess at a given position (`XNMI`), and perform numeric operations to calculate differences (`vVNM`) [40]. Hashcat supports five similar memory commands.

Our tool handles the common special case when the `Q` command is both the only memory command in a rule and appears as the final transformation. Skipping a guess if the input has not changed is often desirable, so this special case is commonly used. We handle this special case by removing the target password from the set of possible preimages after inversion.

While this common case is covered, our analytical techniques cannot efficiently invert or count rules containing memory commands in their full generality. Consider the fourth command above, `vVNM`. By concatenating several such commands, a rule can express any straight-line arithmetic program, and hence any computation expressible in a program with a fixed number of instructions. If JtR were to allow arbitrarily long rules with any number of variables, then in principle any computation would be possible (so, for example, one could write a rule that checks if the length of the input word is prime, or represents a three-colorable graph under some unary encoding, etc.). The variables used by `vVNM` can be used by `XNMI`. Hence, its behavior depends on some arbitrary computation. We observe that JtR does not fully support completely *arbitrary* computation like this, and only allows for 11 variables, along with other length restrictions. However, it appears impossible to efficiently analyze such general computation simulated with JtR rules.

2) Performance Tradeoffs of Enumeration / Non-Simple Regexes: While JtR and Hashcat transformations for truncation and substring extraction are regex-invertible and our tool supports comparing such regexes individually against each wordlist entry, by default we enumerate and sort guesses for such rules as if they were uninvertible. We do so for performance.

We sampled 100 Hashcat regex-invertible rules and compared the wall-clock time of enumerating guesses versus evaluating the regex against each wordlist entry. We separately tested XATO and LinkedIn as the wordlist. We found the total time (both precomputation and evaluating 25,000 passwords from 000webhost) for the larger LinkedIn wordlist was 1.98×10^4 seconds for guess enumeration and 8.96×10^9 seconds for regex matching. For the smaller XATO wordlist, these values were 1.09×10^4 seconds and 6.88×10^8 seconds, respectively. Thus, we enumerate regex-invertible rules by default.

3) The Complexity of Hashcat Shift Case Commands: There are two Hashcat-specific transformations we can invert, yet cannot count. These transformations "lowercase the whole line, then uppercase the first letter and every letter after a space" (`E`) and "lowercase the whole line, then uppercase the first letter and every letter after a custom separator character" (`eX`). Similar to purge, supporting these transformations in the general case would require detailed case analysis. Consider a rule that lowercases the whole line, then uppercases the first letter and every letter after a space, and then requires the resultant word to contain "a." If the input word has only one "a," but the character before "a" is a space, then the word is rejected. To correctly count guesses, one has to know the *relative* locations of all "a"s and spaces, which is inconsistent with our approach.

B. *Evaluating Guess Number Accuracy*

The lower and upper bounds on a password's guess number are computed by summing the guesses made by the rules up to and including (respectively) the rule that first guesses that password. Thus, we must ensure that `invert_rule` and `guess_count` are accurate. We did so by creating unit tests of single transformations, as well as by creating a random-rule generator, crafting 100,000 invertible rules each for JtR and Hashcat. We did not observe any false positives or negatives comparing `invert_rule` with the software's output on random samples of our evaluation sets.

To test the accuracy of `guess_count`, we focused on the complexity created by rejection transformations. Therefore, we randomly generated 100,000 JtR rules and 100,000 Hashcat rules containing at least three transformations and at least three more rejection transformations in arbitrary order. The analytical `guess_count` results exactly matched the empirical results of enumerating the guesses for all 100,000 Hashcat rules. JtR discards guesses that exactly match previous guess. Muting this behavior results in a 100% match between the analytical `guess_count` results and the empirical results.

C. *Detailed Comparison to Existing Password Meters / Proactive Password Checkers*

Proactive password checking using a JtR or Hashcat attack as the metric of password strength entails computing a given password's guess number. Our techniques enable this to be done server-side in under a second for many combinations of rule lists and wordlists. Here, we compare the accuracy and coverage of our techniques to existing meters. For equality, we trained each meter using a sample of 10 million LinkedIn passwords [45] as the training data (probabilistic methods) or wordlist (software tools). We had to use a sample because of implementation limitations for many probabilistic approaches.

Hashcat used the Best64 rules followed by the T0XlC rules and the generated2 rules. **JtR** used the John rules followed by the SpiderLabs rules and the Megatron rules. As in Section VIII, we reordered these rules for each evaluation set based on the other five (**JtR Reordered**). As in Section X, we extended this list with "missing" rules prior to reordering (**JtR Extended**).

As one point of comparison, we arranged this sample of 10 million LinkedIn passwords (6 million after discarding duplicates) in descending order of frequency (**LinkedIn: 10M**), assigning the guess number 1 to the most frequent password, the guess number 2 to the second most frequent, and so on. To test an analogue appropriate for client-side password checking, we also tested the 30,000 most frequent of these (**LinkedIn: 30k**). We tested the popular **zxcvbn** client-side meter [22], which uses combinatoric heuristics to estimate guess numbers. For more equal comparison with other approaches, we also tested replacing zxcvbn's built-in dictionary of 30,000 frequent RockYou passwords with LinkedIn-30k (**zxcvbn + LinkedIn-30k**).

We also compared to probabilistic approaches, including a **Neural Network** password meter [13]. We also tested a PCFG (**PCFG: 2016**) that integrates probability smoothing and other enhancements over previous PCFG approaches [54]. Finally. **Markov: Multi** is a meter that calculates password probabilities under Markov models of different orders (n-gram sizes) [53].

To compare meters, Golla et al. recommend computing the weighted Spearman correlation (r_w) between a meter's guess number or probabilities and a ground truth source [53]. For each evaluation set, we evaluated each of the 25,000 passwords with each meter. As ground truth, we used the frequency of that password in that set (the full set, not the sample, for greater precision). Correlation values closer to 1 indicate better alignment between the meter's ranking and the frequency counts.

Our analytical techniques for JtR and Hashcat had higher correlation (better agreement with the ground truth) for each of the six evaluation sets than both the popular zxcvbn meter and any probabilistic approach we tested (Neural Networks, Markov: Multi, PCFG: 2016), as shown in Table IX. This greater accuracy, however, comes partially at a cost of the ability to relatively rank passwords that are potentially guessable, yet non-trivial to guess. Three approaches we tested — Neural Networks, Markov: Multi, and zcxvbn — estimate a guess number for every possible password. For the remaining approaches, including JtR and Hashcat, passwords that are not guessed in the attack have no true guess number, so we assigned them a guess number one past the maximum guess number. We define a meter's % *coverage* to be the percentage of passwords that are assigned a true guess number (i.e., not one past the maximum guess number). Hashcat and JtR had roughly 40% coverage for 000webhost, roughly 50% coverage for CSDN, and roughly 70%–80% coverage for the four other approaches (Table IX). Thus, these approaches cannot fully leverage the training data in rating the strength of some unseen passwords.

TABLE IX: A comparison of the coverage and accuracy of server-side (top) and client-side (bottom) password meters.

Meter	**Type**	**000webhost**		**Battlefield Heroes**		**Brazzers**		**Clixsense**		**CSDN**		**Neopets**	
		%	r_w	%	r_w	%	r_w	%	r_w	%	r_w	%	r_w
Hashcat (Sec. VII)	Server	40.1	0.512	73.6	0.641	83.3	0.731	69.6	0.695	53.0	0.730	77.8	0.806
JtR: Original (Sec. VII)	Server	39.5	0.512	69.8	0.643	78.4	0.732	68.2	0.696	51.8	0.720	75.4	0.798
JtR: Reordered (Sec. X)	Server	39.5	0.515	69.8	0.644	78.4	0.734	68.2	0.698	51.8	0.731	75.4	0.804
JtR: Extended (Sec. X)	Server	40.3	0.515	70.9	0.644	79.4	0.734	69.1	0.696	52.5	0.729	76.4	0.805
LinkedIn-10M [45]	Server	12.2	0.511	37.0	0.712	46.8	0.775	37.0	0.774	23.9	0.768	33.0	0.747
Markov: Multi [53]	Server	100.0	0.472	100.0	0.539	100.0	0.629	100.0	0.589	100.0	0.663	100.0	0.693
PCFG: 2016 [54]	Server	59.4	0.454	84.5	0.613	91.6	0.707	80.2	0.664	76.1	0.645	83.8	0.752
LinkedIn-30k [45]	Client	3.7	0.354	17.2	0.717	24.3	0.771	17.2	0.714	14.6	0.651	10.3	0.494
Neural Network [13]	Client	100.0	0.507	100.0	0.604	100.0	0.702	100.0	0.654	100.0	0.686	100.0	0.795
zxcvbn [22]	Client	100.0	0.437	100.0	0.586	100.0	0.693	100.0	0.577	100.0	0.667	100.0	0.696
zxcvbn + LinkedIn-30k	Client	100.0	0.440	100.0	0.575	100.0	0.668	100.0	0.616	100.0	0.669	100.0	0.700

Taking the tradeoff between accuracy and % coverage further, we treated a small and a medium-size list of LinkedIn passwords (LinkedIn-30k and LinkedIn-10M) as meters. For four of the six evaluation sets, LinkedIn-10M had a higher correlation (r_w) than any meter, including JtR or Hashcat. However, the % coverage for this approach ranged from 12.2% to only 46.8%, losing any ability to distinguish between any previously unseen passwords.

Many passwords in a given set are singletons, appearing once. As passwords that are frequently used should likely not be considered strong, we also evaluated when meters assigned passwords appearing frequently (≥ 5 times) in an evaluation set guess numbers ranking them among the 25% of hardest-to-guess passwords in that set. These can be considered unsafe errors. As shown in Table X, no meter consistently minimizes the number of unsafe errors, rating common passwords as strong. For some sets, the JtR or Hashcat approaches had the fewest unsafe errors. For other sets, the Neural Network had the fewest.

Using our JtR and Hashcat guess-number calculators as server-side meters thus strikes a balance between accuracy and % coverage. Most crucially, however, our approach models the approaches real attackers use in actual attacks.

D. Detailed Comparison to Existing Password-Cracking Algorithms

Our techniques can also be used to improve the expected success per guess of JtR and Hashcat. Therefore, we compared how JtR and Hashcat compare against the other major password-guessing approaches in an attack. As with the meter comparisons, we again used a sample of 10 million LinkedIn passwords as the wordlist or training data. In addition to the relevant techniques also tested as meters, we used Monte Carlo methods proposed by Dell'Amico et al. [10] to evaluate two Markov models (**Markov: 4-gram** and **Markov: Backoff**), as well as the original PCFG proposal (**PCFG: 2009**) [14]. The latter only guesses passwords whose component strings were seen verbatim in training, which is why some passwords are never guessed. We also graph an attack that has optimal **Perfect Knowledge** of the evaluation set (i.e., knows the full password distribution a priori).

TABLE X: The number of passwords that appear frequently (≥ 5 times) in a set, yet are rated by the meter as among the 25% of hardest-to-guess (or least probable) passwords.

Meter	000webhost	Battlefield Heroes	Brazzers	Clixsense	CSDN	Neopets
Hashcat (Sec. VII)	301	12	63	48	157	792
JtR: Original (Sec. VII)	322	16	61	49	216	877
JtR: Reordered (Sec. X)	294	20	66	46	111	814
JtR: Extended (Sec. X)	297	17	62	39	113	795
LinkedIn: 10M [45]	665	32	87	74	477	1611
Markov: Multi [53]	258	32	63	86	200	736
PCFG: 2016 [54]	324	16	66	52	220	950
LinkedIn: 30k [45]	967	149	270	300	854	2580
Neural Network [13]	236	19	53	51	166	502
zxcvbn [22]	311	33	72	153	200	793
zxcvbn + LinkedIn-30k	299	40	97	110	199	763

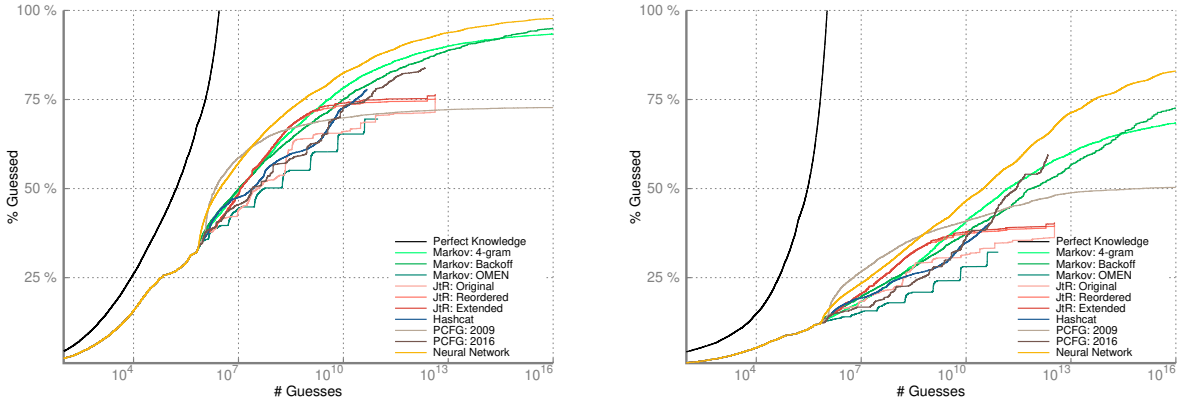
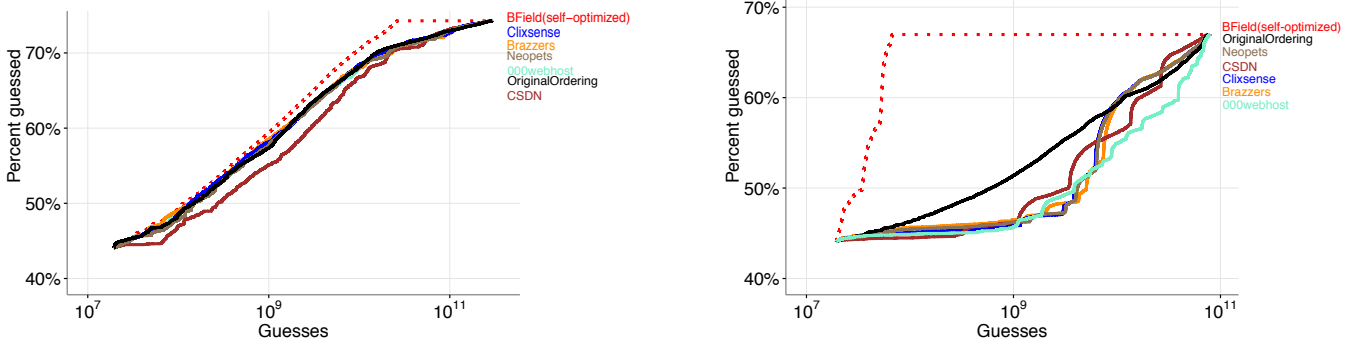


Fig. 3: Guessability of Neopets (left) and 000webhost (right) passwords against different password-cracking approaches.

Figure 3 shows this comparison among password-cracking algorithms for Neopets (left) and 000webhost (right). The graphs for the remaining four evaluation sets resembled Neopets far more closely than 000webhost. Echoing prior work [8], [13], we found probabilistic approaches (particularly Neural Networks and PCFG) often performed best on a guess-by-guess basis.

Surprisingly, though, JtR Extended (using our optimization techniques to add "missing" rules and reorder the rules) performed about as well as, or even better than, probabilistic approaches other than Neural Networks at 10^9 guesses for four of the six evaluation sets. That is, a billion-guess attack will guess about as many passwords using JtR Extended as any approach other than Neural Networks. This result is particularly important because all probabilistic approaches inherently incur substantial computational costs to generate guesses in descending probability order. Therefore, our results somewhat contradict prior work and suggest that JtR and Hashcat, if configured using the techniques we propose, may not lag as far behind probabilistic approaches as previously thought. That said, after 10^9 guesses, JtR Extended performance tended to plateau.

E. Additional Figures and Tables



(a) **Reordering JtR rule lists:** PGS wordlist, Megatron rule list.

(b) **Reordering Hashcat wordlists:** PGS wordlist, T0XlC rule list.

Fig. 4: The impact of reordering rule lists and wordlists. Each graph shows the guessability of Battlefield Heroes reordered artificially based on itself (*self-optimized*) and on each of the five other evaluation sets. Unlike for other JtR rule lists, Figure 4a show that the original order of the Megatron rule list is nearly optimal, while reordering rules based on any English-language set also led to a nearly optimal ordering. Reordering words (Figure 4b), however, appears to overfit to the data.

TABLE XI: The first 100 of the 5,146 SpiderLabs rules and their final position after reordering based on each evaluation set using the PGS wordlist.

Rule	Original Position	000webhost	CSDN	Others
cAz"[0-9]"	1	547	993	501 - 609
Az"[0-9]"	2	271	246	183 - 295
cAz"[0-9][0-9]"	3	569	416	508 - 637
Az"[0-9][0-9]"	4	507	367	313 - 527
cAz"[0-9][0-9][0-9]"	5	587	425	535 - 668
Az"[0-9][0-9][0-9]"	6	568	388	506 - 637
cAz"[0-9][0-9][0-9][0-9]"	7	608	437	552 - 688
Az"[0-9][0-9][0-9][0-9]"	8	588	406	527 - 660
cA0"[0-9]"	9	584	994	616 - 1517
A0"[0-9]"	10	553	380	494 - 607
cA0"[0-9][0-9]"	11	590	1001	624 - 1529
A0"[0-9][0-9]"	12	571	384	507 - 638
cA0"[0-9][0-9][0-9]"	13	627	433	551 - 1443
A0"[0-9][0-9][0-9]"	14	581	399	518 - 655
cA0"[0-9][0-9][0-9][0-9]"	15	634	1034	567 - 710
A0"[0-9][0-9][0-9][0-9]"	16	601	413	536 - 666
/asa@[:c]	17	369	971	377 - 1456
/asa4[:c]	18	437	970	257 - 585
/AsA4[:c]	19	372	129	703 - 1508
/AsA@[:c]	20	806	1052	702 - 857
/bsb8[:c]	21	1427	775	432 - 1169
/BsB8[:c]	22	1546	1053	726 - 1599
/ese3[:c]	23	376	381	176 - 589
/EsE3[:c]	24	1547	1054	75 - 1600
/isi1[:c]	25	436	964	181 - 355
/isi![:c]	26	1548	1055	1451 - 1611
/isi|[:c]	27	1549	1056	1489 - 1612
/IsI1[:c]	28	310	1057	205 - 818
/IsI![:c]	29	1550	1058	1490 - 1613
/IsI|[:c]	30	1551	1059	1491 - 1614
/lsl1[:c]	31	538	955	367 - 1365
/lsl7[:c]	32	556	957	605 - 1517
/lsl|[:c]	33	1552	1060	1492 - 1615
/lsl![:c]	34	1553	1061	606 - 1616
/Lsl1[:c]	35	751	477	2 - 650
/Lsl7[:c]	36	752	478	651 - 810
/Lsl|[:c]	37	749	479	652 - 811
/Lsl![:c]	38	750	480	76 - 812
/oso0[:c]	39	275	961	85 - 232
/OsO0[:c]	40	131	474	55 - 649
/sss$[:c]	41	1405	1062	614 - 1514
/sss5[:c]	42	1469	958	412 - 1447
/SsS$[:c]	43	1554	1063	148 - 1515
/SsS5[:c]	44	190	1064	197 - 1605
/tst+[:c]	45	1555	1065	611 - 1606
/TsT+[:c]	46	1556	1066	1496 - 1617
/ls1![:c]	47	1442	949	433 - 1518
/ls1i[:c]	48	541	948	541 - 1471
/ls1I[:c]	49	1465	952	1362 - 1509
/ls1|[:c]	50	1557	1067	1497 - 1608
/0s0o[:c]	51	549	239	409 - 1469
/0s0O[:c]	52	1458	911	1315 - 1470
/3s3e[:c]	53	1436	819	448 - 1294
/3s3E[:c]	54	1437	820	1229 - 1609
/4s4a[:c]	55	368	179	321 - 1336
/4s4A[:c]	56	1426	770	380 - 1273
/5s5s[:c]	57	393	318	229 - 542
/5s5S[:c]	58	1425	769	1172 - 1312
/7s71[:c]	59	1420	761	1143 - 1286
/7s7L[:c]	60	1417	762	1144 - 1618
/8s8b[:c]	61	542	779	540 - 1343
/8s8B[:c]	62	1428	780	1279 - 1619
/asa@/bsb8[:c]	63	1558	1068	1499 - 1620
/asa@/BsB8[:c]	64	1559	1069	1500 - 1621
/asa@/ese3[:c]	65	1467	1070	601 - 1622
/asa@/EsE3[:c]	66	1560	1071	1501 - 1623
/asa@/isi1[:c]	67	1461	1072	594 - 1524
/asa@/isi![:c]	68	1561	1073	1317 - 1624
/asa@/isi|[:c]	69	1562	1074	1502 - 1625
/asa@/IsI1[:c]	70	1563	1075	5 - 1626
/asa@/IsI![:c]	71	1564	1076	1503 - 1627
/asa@/IsI|[:c]	72	1565	1077	1504 - 1628
/asa@/lsl1[:c]	73	1566	1078	570 - 1620
/asa@/lsl7[:c]	74	1567	1079	1505 - 1629
/asa@/lsl|[:c]	75	1568	1080	1506 - 1630
/asa@/lsl![:c]	76	1569	1081	1507 - 1631
/asa@/Lsl1[:c]	77	1570	1082	1508 - 1632
/asa@/Lsl7[:c]	78	1571	1083	1509 - 1633
/asa@/Lsl|[:c]	79	1572	1084	1510 - 1634
/asa@/Lsl![:c]	80	1573	1085	1511 - 1635
/asa@/oso0[:c]	81	1444	1086	455 - 1371
/asa@/OsO0[:c]	82	1574	1087	1512 - 1636
/asa@/sss$[:c]	83	1136	1088	1293 - 1637
/asa@/sss5[:c]	84	1575	1089	468 - 1629
/asa@/SsS$[:c]	85	1576	1090	1513 - 1638
/asa@/SsS5[:c]	86	1577	1091	1514 - 1639
/asa@/tst+[:c]	87	1578	1092	1515 - 1640
/asa@/TsT+[:c]	88	1579	1093	1516 - 1641
/asa@/ls1![:c]	89	1580	1094	467 - 1642
/asa@/ls1i[:c]	90	1581	1095	1090 - 1643
/asa@/ls1I[:c]	91	1582	1096	1091 - 1644
/asa@/ls1|[:c]	92	1583	1097	1519 - 1645
/asa@/0s0o[:c]	93	1584	1098	264 - 1646
/asa@/0s0O[:c]	94	1585	1099	1520 - 1647
/asa@/3s3e[:c]	95	1081	1100	801 - 1648
/asa@/3s3E[:c]	96	1586	1101	1522 - 1649
/asa@/4s4a[:c]	97	957	1102	748 - 1551
/asa@/4s4A[:c]	98	958	1103	1524 - 1650
/asa@/5s5s[:c]	99	905	1104	1525 - 1651
/asa@/5s5S[:c]	100	906	1105	1526 - 1652

TABLE XII: The first 100 of the 15,324 Megatron rules and their final position after reordering based on each evaluation set using the PGS wordlist.

Rule	Original Position	000webhost	CSDN	Others
: A1"1"	1	8	378	2 - 11
\]\] Q	2	3	6	1 - 1
u Q	3	77	19	16 - 56
D5 Q	4	21	501	5 - 38
D3 Q	5	20	83	7 - 43
c Q	6	7	280	3 - 17
: A1"2"	7	45	147	12 - 24
: A1"3"	8	27	119	19 - 28
: A1"7"	9	35	45	16 - 25
D2 Q	10	16	25	14 - 33
: A1"5"	11	26	268	15 - 31
D6 Q	12	17	30	2 - 8
: A1"4"	13	23	118	12 - 25
D4 Q	14	11	82	7 - 29
: A1"6"	15	51	226	18 - 60
: A1"8"	16	46	41	22 - 64
: A1"12"	17	15	23	2 - 155
D1D1 Q	18	18	15	14 - 1173
: A1"9"	19	29	106	32 - 141
: A1"123"	20	6	11	8 - 22
: A1"23"	21	49	247	34 - 299
} Q	22	36	49	37 - 54
: A1"13"	23	203	213	30 - 73
: A1"0"	24	60	28	28 - 36
: A1"11"	25	25	54	14 - 111
\]\]\] Q	26	4	9	4 - 89
: A1"07"	27	1049	397	67 - 725
: A1"01"	28	31	212	32 - 153
: A1"21"	29	39	104	53 - 173
: A1"22"	30	56	214	26 - 213
: A1"08"	31	117	334	48 - 487
: A1"06"	32	352	398	100 - 812
Xm1z Q	33	13	10	3 - 129
r Q	34	48	12	37 - 76
: A1"!"	35	340	322	25 - 5556
: A1"69"	36	95	173	137 - 428
D1 Q	37	213	84	20 - 66
: A0"1"	38	136	1214	108 - 1625
: A1"14"	39	204	283	65 - 813
: A1"10"	40	82	52	54 - 174
: A1"05"	41	353	400	138 - 339
: A1"15"	42	227	335	102 - 488
\]\] Q A1"y"	43	1187	499	143 - 1180
: A1"88"	44	167	72	56 - 287
u Q A1"1"	45	1034	1192	385 - 1529
: A1"16"	46	127	507	68 - 291
: A1"09"	47	226	153	122 - 194
: A1"s"	48	650	385	48 - 341
: A1"18"	49	327	44	77 - 195
: A1"a"	50	47	51	43 - 1377
\]\] Q A1"e"	51	1183	1086	44 - 1174
: A1"17"	52	120	248	183 - 547
: A1"24"	53	83	396	46 - 367
: A0"j"	54	110	1249	186 - 1426
: A1"".	55	176	487	190 - 5557
: A1"89"	56	139	85	47 - 490
\]\] Q A1"1"	57	107	174	233 - 703
: A1"04"	58	115	215	290 - 656
: A1"03"	59	572	336	67 - 1753
: A1"25"	60	94	123	59 - 340
D1D1D1 Q	61	12	14	9 - 72
: A1"00"	62	108	26	40 - 162
: A1"02"	63	328	285	124 - 724
c Q A1"1"	64	69	1183	52 - 311
x05d Q	65	33	8	6 - 63
: A1"20"	66	325	60	68 - 221
: A1"99"	67	52	61	28 - 264
: A1"77"	68	78	287	123 - 811
: A1"87"	69	129	252	41 - 116
: A1"19"	70	113	508	123 - 489
\]\]\]\] Q	71	5	2	7 - 16
>2 o2n	72	297	662	74 - 242
: A0"a"	73	68	1215	168 - 1718
: A1"z"	74	308	492	334 - 1777
: A1"27"	75	354	284	127 - 409
: A1"92"	76	72	349	58 - 378
/i si1 Q	77	57	387	20 - 68
>2 o2y	78	1289	1169	277 - 539
>2 o2m	79	298	1173	80 - 278
D7 Q	80	9	20	6 - 14
: A1"90"	81	269	188	44 - 375
: A1"26"	82	355	286	126 - 558
: A1"33"	83	66	57	51 - 175
: A1"93"	84	168	295	82 - 435
: A1"28"	85	119	185	78 - 368
>3 o3y	86	1279	1156	250 - 1275
: A1"86"	87	326	218	288 - 441
: A0"m"	88	1300	383	231 - 1631
/e se3 Q	89	41	449	18 - 508
: A1"91"	90	100	135	80 - 492
>4 o4y	91	291	1145	172 - 1584
D6 /a sa4 Q	92	99	1047	109 - 1141
: A1"94"	93	96	356	59 - 158
>3 o3i	94	293	1158	56 - 1590
D1 /s ss5 Q	95	240	2721	210 - 1502
: A1"95"	96	140	514	85 - 383
>3 o3a	97	76	1159	151 - 1274
: A0"k"	98	1351	693	189 - 1661
/o so0 Q	99	19	1048	7 - 32
\]\] Q A1"o"	100	1186	77	154 - 368