Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks

William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/melicher

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks

William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri,
Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor
Carnegie Mellon University

Abstract

Human-chosen text passwords, today's dominant form of authentication, are vulnerable to guessing attacks. Unfortunately, existing approaches for evaluating password strength by modeling adversarial password guessing are either inaccurate or orders of magnitude too large and too slow for real-time, client-side password checking. We propose using artificial neural networks to model text passwords' resistance to guessing attacks and explore how different architectures and training methods impact neural networks' guessing effectiveness. We show that neural networks can often guess passwords more effectively than state-of-the-art approaches, such as probabilistic context-free grammars and Markov models. We also show that our neural networks can be highly compressed—to as little as hundreds of kilobytes—without substantially worsening guessing effectiveness. Building on these results, we implement in JavaScript the first principled client-side model of password guessing, which analyzes a password's resistance to a guessing attack of arbitrary duration with sub-second latency. Together, our contributions enable more accurate and practical password checking than was previously possible.

1 Introduction

Text passwords are currently the most common form of authentication, and they promise to continue to be so for the foreseeable future [53]. Unfortunately, users often choose predictable passwords, enabling password-guessing attacks. In response, proactive password checking is used to evaluate password strength [19].

A common way to evaluate the strength of a password is by running or simulating password-guessing techniques [35,59,92]. A suite of well-configured guessing techniques, encompassing both probabilistic approaches [37,65,93] and off-the-shelf password-recovery tools [74, 83], can accurately model the vulnerability of passwords to guessing by expert attackers [89]. Unfortunately, these techniques are often very computationally intensive, requiring hundreds of megabytes to gigabytes of disk space, and taking days to execute. Therefore, they are typically unsuitable for real-time evaluation of password strength, and sometimes for any practically useful evaluation of password strength.

With the goal of gauging the strength of human-chosen text passwords both more accurately and more practically, we propose using artificial neural networks to guess passwords. Artificial neural networks (hereafter referred to as "neural networks") are a machine-learning technique designed to approximate highly dimensional functions. They have been shown to be very effective at generating novel sequences [49,84], suggesting a natural fit for generating password guesses.

In this paper, we first comprehensively test the impact of varying the neural network model size, model architecture, training data, and training technique on the network's ability to guess different types of passwords. We compare our implementation of neural networks to state-of-the-art password-guessing models, including widely studied Markov models [65] and probabilistic context-free grammars [59, 93], as well as software tools using mangled dictionary entries [74, 83]. In our tests, we evaluate the performance of probabilistic models to large numbers of guesses using recently proposed Monte Carlo methods [34]. We find that neural networks guess passwords more successfully than other password-guessing methods in general, especially so beyond 10^{10} guesses and on non-traditional password policies. These cases are interesting because password-guessing attacks often proceed far beyond 10^{10} guesses [44,46] and because existing password-guessing attacks underperform on new, non-traditional password policies [79, 80].

Although more effective password guessing using neural networks is an important contribution on its own, we also show that the neural networks we use can be highly compressed with minimal loss of guessing ef-

fectiveness. Our approach is thus far more suitable than existing password-guessing methods for client-side password checking. Most existing client-side password checkers are inaccurate [33] because they rely on simple, easily compressible heuristics, such as counting the number of characters or character classes in a password. In contrast, we show that a highly compressed neural network more accurately measures password strength than existing client-side checkers. We can compress such a neural network into hundreds of kilobytes, which is small enough to be included in an app for mobile devices, bundled with encryption software, or used in a web page password meter.

To demonstrate the practical suitability of neural networks for client-side password checking, we implement and benchmark a neural-network password checker in JavaScript. This implementation, which we have released as open-source software,[1] is immediately suitable for use in mobile apps, browser extensions, and web page password meters. Our implementation gives real-time feedback on password strength in fractions of a second, and it more accurately measures resistance to guessing than existing client-side methods.

In summary, this paper makes three main contributions that together substantially increase our ability to detect and help eliminate weak passwords. First, we propose neural networks as a model for guessing human-chosen passwords and comprehensively evaluate how varying their training, parameters, and compression impacts guessing effectiveness. In many circumstances, neural networks guess more accurately than state-of-art techniques. Second, leveraging neural networks, we create a password-guessing model sufficiently compressible and efficient for client-side proactive password checking. Third, we build and benchmark a JavaScript implementation of such a checker. In common web browsers running on commodity hardware, this implementation models an arbitrarily high number of adversarial guesses with sub-second latency, while requiring only hundreds of kilobytes of data to be transferred to a client. Together, our contributions enable more accurate proactive password checking, in a far broader range of common scenarios, than was previously possible.

2 Background and Related Work

To highlight when password strength matters, we first summarize password-guessing attacks. We then discuss metrics and models for evaluating password strength, as well as lightweight methods for estimating password strength during password creation. Finally, we summarize prior work on generating text using neural networks.

[1] https://github.com/cupslab/neural_network_cracking

2.1 Password-Guessing Attacks

The extent to which passwords are vulnerable to guessing attacks is highly situational. For phishing attacks, keyloggers, or shoulder surfing, password strength does not matter. Some systems implement rate-limiting policies, locking an online account or a device after a small number of incorrect attempts. In these cases, passwords other than perhaps the million most predictable are unlikely to be guessed [39].

Guessing attacks are a threat, however, in three other scenarios. First, if rate limiting is not properly implemented, as is believed to have been the case in the 2014 theft of celebrities' personal photos from Apple's iCloud [50], large-scale guessing becomes possible. Second, if a database of hashed passwords is stolen, which sadly occurs frequently [20, 23, 27, 45, 46, 67, 73, 75, 87], an offline attack is possible. An attacker chooses likely candidate passwords, hashes them, and searches the database for a matching hash. When a match is found, attackers can rely on the high likelihood of password reuse across accounts and try the same credentials on other systems [32]. Attacks leveraging password reuse have real-world consequences, including the recent compromise of Mozilla's Bugzilla database due to an administrator reusing a password [76] and the compromise of 20 million accounts on Taobao, a Chinese online shopping website similar to eBay, due to password reuse [36].

Third, common scenarios in which cryptographic key material is derived from, or protected by, a password are vulnerable to large-scale guessing in the same way as hashed password databases for online accounts. For instance, for password managers that sync across devices [52] or privacy-preserving cloud backup tools (e.g., SpiderOak [82]), the security of files stored in the cloud depends directly on password strength. Furthermore, cryptographic keys used for asymmetric secure messaging (e.g., GPG private keys), disk-encryption tools (e.g., TrueCrypt), and Windows Domain Kerberos Tickets [31] are protected by human-generated passwords. If the file containing this key material is compromised, the strength of the password is critical for security. The importance of this final scenario is likely to grow with the adoption of password managers and encryption tools.

2.2 Measuring Password Strength

Models of password strength often take one of two conceptual forms. The first relies on purely statistical methods, such as Shannon entropy or other advanced statistical approaches [21, 22]. However, because of the unrealistically large sample sizes required, we consider these types of model out of scope. The second conceptual approach is to simulate adversarial password guess-

ing [34, 65, 89]. Our application of neural networks follows this method. Below, we describe the password-guessing approaches that have been widely studied in academia and used in adversarial password cracking, all of which we compare to neural networks in our analyses. Academic studies of password guessing have focused on probabilistic methods that take as input large password sets, then output guesses in descending probability order. Password cracking tools rely on efficient heuristics to model common password characteristics.

Probabilistic Context-Free Grammars One probabilistic method uses probabilistic context-free grammars (PCFGs) [93]. The intuition behind PCFGs is that passwords are built with template structures (e.g., 6 letters followed by 2 digits) and terminals that fit into those structures. A password's probability is the probability of its structure multiplied by those of its terminals.

Researchers have found that using separate training sources for structures and terminals improves guessing [59]. It is also beneficial to assign probabilities to unseen terminals by smoothing, as well as to augment guesses generated by the grammar with passwords taken verbatim from the training data without abstracting them into the grammar [60]. Furthermore, using natural-language dictionaries to instantiate terminals improves guessing, particularly for long passwords [91].

Markov Models Using Markov models to guess passwords, first proposed in 2005 [70], has recently been studied more comprehensively [37, 65]. Conceptually, Markov models predict the probability of the next character in a password based on the previous characters, or context characters. Using more context characters can allow for better guesses, yet risks overfitting. Smoothing and backoff methods compensate for overfitting. Researchers have found that a 6-gram Markov model with additive smoothing is often optimal for modeling English-language passwords [65]. We use that configuration in our analyses.

Mangled Wordlist Methods In adversarial password cracking, software tools are commonly used to generate password guesses [44]. The most popular tools transform a wordlist (passwords and dictionary entries) using mangling rules, or transformations intended to model common behaviors in how humans craft passwords. For example, a mangling rule may append a digit and change each 'a' to '@'. Two popular tools of this type are Hashcat [83] and John the Ripper (JtR, [74]). While these approaches are not directly based on statistical modeling, they produce fairly accurate guesses [89] quickly, which has led to their wide use [44].

2.3 Proactive Password Checking

Although the previously discussed password-guessing models can accurately model human-created passwords [89], they take hours or days and megabytes or gigabytes of disk space, making them too resource-intensive to provide real-time feedback to users. Current real-time password checkers can be categorized based on whether they run entirely client-side. Checkers with a server-side component can be more accurate because they can leverage large amounts of data. For instance, researchers have proposed using server-side Markov models to gauge password strength [26]. Others have studied using training data from leaked passwords and natural-language corpora to show users predictions about what they will type next [61].

Unfortunately, a server-side component introduces substantial disadvantages for security. In some cases, sending a password to a server for password checking destroys all security guarantees. For instance, passwords that protect an encrypted volume (e.g., TrueCrypt) or cryptographic keys (e.g., GPG), as well as the master password for a password manager, should never leave the user's device, even for proactive password checking. As a result, accurate password checking is often missing from these security-critical applications. In cases when a password is eventually sent to the server (e.g., for an online account), a real-time, server-side component both adds latency and opens password meters to powerful side-channel attacks based on keyboard timing, message size, and caching [81].

Prior client-side password checkers, such as those running entirely in a web browser, rely on heuristics that can be easily encoded. Many common meters rate passwords based on their length or inclusion of different character classes [33, 88]. Unfortunately, in comprehensive tests of both client- and server-side password meters, all but one meter was highly inaccurate [33]. Only `zxcvbn` [94, 95], which uses dozens of more advanced heuristics, gave reasonably accurate strength estimations. Such meters, however, do not directly model adversarial guessing because of the inability to succinctly encode models and calculate real-time results. In contrast, our approach models adversarial guessing entirely on the client side.

2.4 Neural Networks

Neural networks, which we use to model passwords, are a machine-learning technique for approximating highly dimensional functions. Designed to model human neurons, they are particularly adept at fuzzy classification problems and generating novel sequences. Our method of generating candidate password guesses draws heavily on previous work that generated the probability of

the next element in a string based on the preceding elements [49, 84]. For example, in generating the string *password*, a neural network might be given *passwor* and output that *d* has a high probability of occurring next.

Although password creation and text generation are conceptually similar, little research has attempted to use insights from text generation to model passwords. A decade ago, neural networks were proposed as a method for classifying passwords into two very broad categories (weak or strong) [30], but that work did not seek to model the order in which passwords would be guessed or other aspects of a guessing attack. To our knowledge, the only proposal to use neural networks in a password-guessing attack was a recent blog post [71]. In sharp contrast to our extensive testing of different parameters to make neural networks effective in practice, that work made few refinements to the application of neural networks, leading the author to doubt that the approach has "any practical relevance." Additionally, that work sought only to model a few likely password guesses, as opposed to our use of Monte Carlo methods to simulate an arbitrary number of guesses.

Conceptually, neural networks have advantages over other methods. In contrast to PCFGs and Markov models, the sequences generated by neural networks can be inexact, novel sequences [49], which led to our intuition that neural networks might be appropriate for password guessing. Prior approaches to probabilistic password guessing (e.g., Markov models [26]) were sufficiently memory-intensive to be impractical on only the client-side. However, neural networks can model natural language in far less space than Markov models [68]. Neural networks have also been shown to transfer knowledge about one task to related tasks [97]. This is crucial for targeting novel password-composition policies, for which training data is sparse at best.

3 System Design

We experimented with a broad range of options in a large design space and eventually arrived at a system design that 1) leverages neural networks for password guessing, and 2) provides a client-side guess estimation method.

3.1 Measuring Password Strength

Similarly to Markov models, neural networks in our system are trained to generate the next character of a password given the preceding characters of a password. Figure 1 illustrates our construction. Like in Markov models [34, 65], we rely on a special password-ending symbol to model the probability of ending a password after a sequence of characters. For example, to calculate the probability of the entire password 'bad', we would start with an empty password, and query the network for the probability of seeing a 'b', then seeing an 'a' after 'b', and then of seeing a 'd' after 'ba', then of seeing a complete password after 'bad'. To generate passwords from a neural network model, we enumerate all possible passwords whose probability is above a given threshold using a modified beam-search [64], a hybrid of depth-first and breadth-first search. If necessary, we can suppress the generation of non-desirable passwords (e.g., those against the target password policy) by filtering those passwords. Then, we sort passwords by their probability. We use beam-search because breadth-first's memory requirements do not scale, and because it allows us to take better advantage of GPU parallel processing power than depth-first search. Fundamentally, this method of guess enumeration is similar to that used in Markov models, and it could benefit from the same optimizations, such as approximate sorting [37]. A major advantage over Markov models is that the neural network model can be efficiently implemented on the GPU.

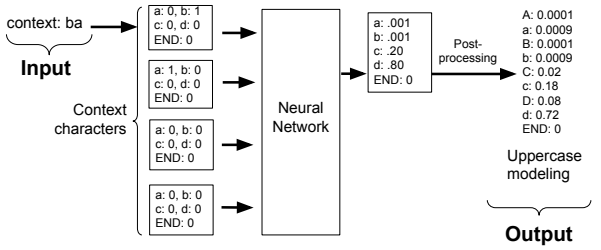


Figure 1: **An example of using a neural network to predict the next character of a password fragment.** The network is being used to predict a 'd' given the context 'ba'. This network uses four characters of context. The probabilities of each next character are the output of the network. Post processing on the network can infer probabilities of uppercase characters.

Calculating Guess Numbers In evaluating password strength by modeling a guessing attack, we calculate a password's *guess number*, or how many guesses it would take an attacker to arrive at that password if guessing passwords in descending order of likelihood. The traditional method of calculating guess numbers by enumeration is computationally intensive. For example, enumerating more than 10^{10} passwords would take roughly 16 days in our unoptimized implementation on an NVidia GeForce GTX 980 Ti. However, in addition to guess number enumeration, we can also estimate guess numbers accurately and efficiently using Monte Carlo simulations, as proposed by Dell'Amico and Filippone [34].

3.2 Our Approach

There are many design decisions necessary to train neural networks. The design space forces us to decide on

the modeling alphabet, context size, type of neural network architecture, training data, and training methodology. We experiment along these dimensions.

Model Architectures In this work, we use recurrent neural networks because they have been shown to be useful for generating text in the context of character-level natural language [49, 84]. Recurrent neural networks are a specific type of neural network where connections in the network can process elements in sequences and use an internal memory to remember information about previous elements in the sequence. We experiment with two different recurrent architectures in Section 5.1.

Alphabet Size We focus on character-level models, rather than more common word-level models, because there is no established dictionary of words for password generation. We also complement our analysis with exploratory experiments using syllable-level models in Section 5.1. We decided to explore hybrid models based on prior work in machine learning [68]. In the hybrid construction, in addition to characters, the neural network is allowed to model sub-word units, such as syllables or tokens. We chose to model 2,000 different tokens based on prior work [68] and represent those tokens the same way we would characters. A more thorough study of tokenized models would explore both more and fewer tokens. Using tokenized structures, the model can then output the probability of the next character being an 'a' or the token 'pass'. We generated the list of tokens by tokenizing words in our training set along character-class boundaries and selecting the 2,000 most frequent ones.

Like prior work [26], we observed empirically that modeling all characters unnecessarily burdens the model and that some characters, like uppercase letters and rare symbols, are better modeled outside of the neural network. We can still create passwords with these characters by interpreting the model's output as templates. For example, when the neural network predicts an 'A' character, we post-process the prediction to predict both 'a' and 'A' by allocating their respective probabilities based on the number of occurrences of 'a' and 'A' in the training data—as shown in Figure 1. The intuition here is that we can reduce the amount of resources consumed by the neural network when alternate heuristic approaches can efficiently model certain phenomena (e.g., shifts between lowercase and uppercase letters).

Password Context Predictions rely on the context characters. For example, in Figure 1, the context characters are 'ba' and the target prediction is 'd'. Increasing the number of context characters increases the training time, while decreasing the number of context characters could potentially decrease guessing success.

We experimented with using all previous characters in the password as context and with only using the previous ten characters. We found in preliminary tests that using ten characters was as successful at guessing and trained up to an order of magnitude faster, and thus settled on this choice. When there are fewer than ten context characters, we pad the input with zeros. In comparison, best-performing Markov models typically use five characters of context [34, 65]. While Markov models can overfit if given too much context, neural networks typically overfit when there are too many parameters.

Providing context characters in reverse order—e.g., predicting 'd' from 'rowssap' instead of 'passwor'—has been shown to sometimes improve performance [48]. We empirically evaluate this technique in Section 5.1.

Model Size We must also decide how many parameters to include in models. To gauge the effect of changing the model size on guessing success, we test a large neural network with 15,700,675 parameters and a smaller network with 682,851 parameters. The larger size was chosen to limit the amount of time and GPU memory used by the model, which required one and a half weeks to fully train on our larger training set. The smaller size was chosen for use in our browser implementation because it could realistically be sent over the Internet; compressed, this network is a few hundred kilobytes. We evaluate the two sizes of models with a variety of password policies, since each policy may respond differently to size constraints, and describe the results in Section 5.1.

Transference Learning We experimented with a specialized method of training neural networks that takes advantage of *transference learning*, in which different parts of a neural network learn to recognize different phenomena during training [97]. One of the key problems with targeting non-traditional password policies is that there is little training data. For example, in our larger training set, there are 105 million passwords, but only 2.6 million satisfy a password policy that requires a minimum of 16 characters. The sparsity of training samples limits guessing approaches' effectiveness against such non-traditional policies. However, if trained on all passwords, the learned model is non-optimal because it generates passwords that are not accurate for our target policy even if one ignores passwords that do not satisfy the policy. Transference learning lets us train a model on all passwords, yet tailor its guessing to only longer passwords.

When using transference learning, the model is first trained on all passwords in the training set. Then, the lower layers of the model are frozen. Finally, the model is retrained only on passwords in the training set that fit

the policy. The intuition is that the lower layers in the model learn low-level features about the data (e.g., that 'a' is a vowel), and the higher layers learn higher-level features about the data (e.g., that vowels often follow consonants). Similarly, the lower layers in the model may develop the ability to count the number of characters in a password, while the higher level layers may recognize that passwords are typically eight characters long. By fine-tuning the higher-level parameters, we can leverage what the model learned about all passwords and retarget it to a policy for which training data is sparse.

Training Data We experimented with different sets of training data; we describe experiments with two sets of passwords in Sections 4.1 and 5.2, and also with including natural language in training data in Section 5.1. For machine-learning algorithms in general, more training data is better, but only if the training data is a close match for the passwords we test on.

3.3 Client-Side Models

Deploying client-side (e.g., browser-based) password-strength-measuring tools presents severe challenges. To minimize the latency experienced by users, these tools should execute quickly and transfer as little data as possible over the network. Advanced guessing tools (e.g., PCFG, Markov models, and tools like JtR and Hashcat) run on massively parallel servers and require on the order of hundreds of megabytes or gigabytes of disk space. Typically, these models also take hours or days to return results of strength-metric tests, even with recent advances in efficient calculation [34], which is unsuitable for real-time feedback. In contrast, by combining a number of optimizations with the use of neural networks, we can build accurate password-strength-measuring tools that are sufficiently fast for real-time feedback and small enough to be included in a web page.

3.3.1 Optimizing for Model Size

To deploy our prototype implementation in a browser, we developed methods for succinctly encoding it. We leveraged techniques from graphics for encoding 3D models for browser-based games and visualizations [29]. Our encoding pipeline contains four different steps: weight quantization, fixed-point encoding, ZigZag encoding, and lossless compression. Our overall strategy is to send fewer bits and leverage existing lossless compression methods that are natively supported by browser implementations, such as `gzip` compression [41]. We describe the effect that each step in the pipeline has on compression in Section 5.3. We also describe encoding a short wordlist of passwords in Bloom filters.

Weight Quantization First, we quantized the weights of the neural network to represent them with fewer digits. Rather than sending all digits of the 32-bit floating-point numbers that describe weights, we only send the most significant digits. Weight quantization is routinely used for decreasing model size, but can increase error [68]. We show the effect of quantization on error rates in Section 5.3. We experimentally find that quantizing weights up to three decimal digits leads to minimal error.

Fixed-point Encoding Second, instead of representing weights using floating-point encoding, we used fixed-point encoding. Due to the weight-quantization step, many of the weight values are quantized to the same values. Fixed-point encoding allows us to more succinctly describe the quantized values using unsigned integers rather than floating point numbers on the wire: one could internally represent a quantized weight between -5.0 and 5.0 with a minimum precision of 0.005, as between -1000 and 1000 with a precision of 1. Avoiding the floating-point value would save four bytes. While lossless compression like `gzip` partially reduces the need for fixed-point encoding, we found that such scaling still provides an improvement in practice.

ZigZag Encoding Third, negative values are generally more expensive to send on the wire. To avoid sending negative values, we use ZigZag encoding [8]. In ZigZag encoding, signed values are encoded by using the last bit as the sign bit. So, the value of 0 is encoded as 0, but the value of -1 is encoded as 1, 1 is encoded as 2, -2 is encoded as 3, and so on.

Lossless Compression We use regular `gzip` or `deflate` encoding as the final stage of the compression pipeline. Both `gzip` and `deflate` produce similar results in terms of model size and both are widely supported natively by browsers and servers. We did not consider other compression tools, like LZMA, because their native support by browsers is not as widespread, even though they typically result in slightly smaller models.

Bloom Filter Word List To increase the success of client-side guessing, we also store a word list of frequently guessed passwords. As in previous work [89], we found that for some types of password-cracking methods, prepending training passwords improves guessing effectiveness. We stored the first two million most frequently occurring passwords in our training set in a series of compressed Bloom filters [69].

Because Bloom filters cannot map passwords to the number of guesses required to crack, and only compute

existence in a set, we use multiple Bloom filters in different groups: in one Bloom filter, we include passwords that require fewer than 10 guesses; in another, all passwords that require fewer than 100 guesses; and so on. On the client, a password is looked up in each filter and assigned a guess number corresponding to the filter with the smallest set of passwords. This allows us to roughly approximate the guess number of a password without increasing the error bounds of the Bloom filter. To drastically decrease the number of bits required to encode these Bloom filters, we only send passwords that meet the requirements of the policy and would have neural-network-computed guess numbers more than three orders of magnitude different from their actual guess numbers. We limited this word list to be about 150KB after compression in order to limit the size of our total model. We found that significantly more space would be needed to substantially improve guessing success.

3.3.2 Optimizing for Latency

We rely on precomputation and caching to make our prototype sufficiently fast for real-time feedback. Our target latency is near 100 ms because that is the threshold below which updates appear instantaneous [72].

Precomputation We precompute guess numbers instead of calculating guess numbers on demand because all methods of computing guess numbers on demand are too slow to give real-time feedback. For example, even with recent advances in calculation efficiency [34], our fastest executing model, the Markov model, requires over an hour to estimate guess numbers of our test set passwords, with other methods taking days. Precomputation decreases the latency of converting a password probability to a guess number: it becomes a quick lookup in a table on the client.

The drawback of this type of precomputation is that guess numbers become inexact due to the quantization of the probability-to-guess-number mapping. We experimentally measure (see Section 5.3) the accuracy of our estimates, finding the effect on accuracy to be low. For the purpose of password-strength estimation, we believe the drawback to be negligible, in part because results are typically presented to users in more heavily quantized form. For instance, users may be told their password is "weak" or "strong." In addition, the inaccuracies introduced by precomputation can be tuned to result in safe errors, in that any individual password's guess number may be an underestimate, but not an overestimate.

Caching Intermediate Results We also cache results from intermediate computations. Calculating the probability of a 10-character password requires 11 full computations of the neural network, one for each character and one for the end symbol. By caching probabilities of each substring, we significantly speed up the common case in which a candidate password changes by having a character added to or deleted from its end. We experimentally show the benefits of caching in Section 5.3.

Multiple Threads On the client side, we run the neural network computation in a separate thread from the user interface for better responsiveness of the user interface.

3.4 Implementation

We build our server-side implementation on the Keras library [28] and the client-side implementation on the neocortex browser implementation [5] of neural networks. We use the Theano back-end library for Keras, which trains neural networks faster by using a GPU rather than a CPU [17, 18]. Our implementation trains networks and guesses passwords in the Python programming language. Guess number calculation in the browser is performed in JavaScript. Our models typically used three long short-term memory (LSTM) recurrent layers and two densely connected layers for a total of five layers. On the client side, we use the WebWorker browser API to run neural network computations in their own thread [10].

For some applications, such as in a password meter, it is desirable to conservatively estimate password strength. Although we also want to minimize errors overall, on the client we prefer to underestimate a password's resistance to guessing, rather than overestimate it. To get a stricter underestimate of guess numbers on our client-side implementation, we compute the guess number without respect to capitalization. We find in practice that our model is able to calculate a stricter underestimate this way, without overestimating many passwords' strength. We don't do this for the server-side models because those models are used to generate candidate password guesses, rather than estimating a guess number. After computing guess numbers, we apply to them a constant scaling factor, which acts as a security parameter, to make the model more conservative at the cost of making more errors. We discuss this tradeoff more in Section 5.3.

4 Testing Methodology

To evaluate our implementation of neural networks, we compare it to multiple other password cracking methods, including PCFGs, Markov models, JtR, and Hashcat. Our primary metric for guessing accuracy is the guessability of our test set of human-created passwords. The guessability of an individual password is measured by how many guesses a guesser would take to crack a

password. We experiment with two sets of training data and with five sets of test data. For each set of test data, we compute the percentage of passwords that would be cracked after a particular number of guesses. More accurate guessing methods correctly guess a higher percentage of passwords in our test set.

For probabilistic methods—PCFG, Markov models, and neural networks—we use recent work to efficiently compute guess numbers using Monte Carlo methods [34]. For Monte Carlo simulations, we generate and compute probabilities for at least one million random passwords to provide accurate estimates. While the exact error of this technique depends heavily on each method, guess number, and individual password, typically we observed 95% confidence intervals of less than 10% of the value of the guess-number estimate; passwords for which the error exceeded 10% tended to be guessed only after more than 10^{18} guesses. For all Monte Carlo simulations, we model up to 10^{25} guesses for completeness. This is likely an overestimate of the number of guesses that even a well-resourced attacker could be able to or would be incentivized to make against one password.

To calculate guessability of passwords using mangling-rule-based methods—JtR and Hashcat—we enumerate all guesses that these methods make. This provides exact guess numbers, but fewer guesses than we simulate with other methods. Across our different test sets, the mangling-rule-based methods make between about 10^{13} and 10^{15} guesses.

4.1 Training Data

To train our algorithms, we used a mixture of leaked and cracked password sets. We believe this is ethical because these password sets are already publicly available and we cause no additional harm with their use.

We explore two different sets of training data. We term the first set the Password Guessability Service (*PGS*) training set, used by prior work [89]. It contains the Rockyou [90] and Yahoo! [43] leaked password sets. For guessing methods that use natural language, it also includes the web2 list [11], Google web corpus [47], and an inflection dictionary [78]. This set totals 33 million passwords and 5.9 million natural-language words.

The second set (the *PGS++* training set) augments the PGS training set with additional leaked and cracked password sets [1,2,3,6,7,9,12,13,14,15,16,20,23,25,42,43,55,56,57,62,63,67,75,77,85,90]. For methods that use natural language, we include the same natural-language sources as the PGS set. This set totals 105 million passwords and 5.9 million natural-language words.

4.2 Testing Data

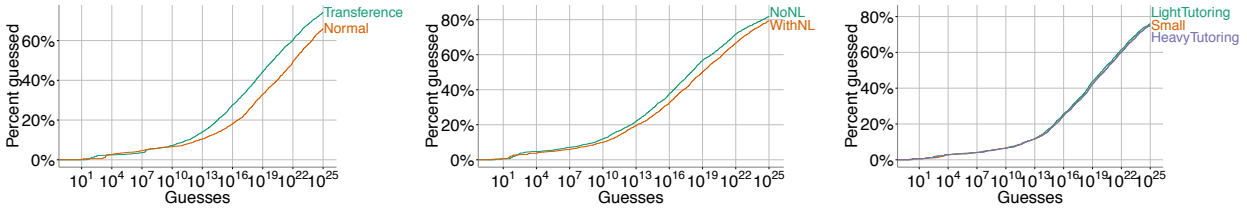
For our testing data we used passwords collected from Mechanical Turk (MTurk) in the context of prior research studies, as well as a set sampled from the leak of plaintext passwords from 000webhost [40]. In addition to a common policy requiring only eight characters, we study three less common password policies shown to be more resistant to guessing [66, 80]: 4class8, 3class12, and 1class16, all described below. We chose the MTurk sets to get passwords created under more password policies than were represented in leaked data. Passwords generated using MTurk have been found to be similar to real-world, high-value passwords [38, 66]. Nonetheless, we chose the 000webhost leak to additionally compare our results to real passwords from a recently leaked password set. In summary, we used five testing datasets:

- **1class8**: 3,062 passwords longer than eight characters collected for a research study [59]
- **1class16**: 2,054 passwords longer than sixteen characters collected for a research study [59]
- **3class12**: 990 passwords that must contain at least three character classes (uppercase letters, lowercase letters, symbols, digits) and be at least twelve characters long collected for a research study [80]
- **4class8**: 2,997 passwords that must contain all four character classes and be at least eight characters long collected for a research study [66]
- **webhost**: 30,000 passwords randomly sampled from among passwords containing at least eight characters in the 000webhost leak [40]

4.3 Guessing Configuration

PCFG We used a version of PCFG with terminal smoothing and hybrid structures [60], and included natural-language dictionaries in the training data, weighted for each word to count as one tenth of a password. We also separated training for structures and terminals, and trained structures only on passwords that conform to the target policy. This method does not generate passwords that do not match the target policy.

For PCFG, Monte Carlo methods are not able to estimate unique guess numbers for passwords that have the same probability. This phenomenon manifests in the Monte Carlo graphs with jagged edges, where many different passwords are assigned the same guess number (e.g., in Figure 5c before 10^{23}). We assume that an optimal attacker could order these guesses in any order, since they all have the same likelihood according to the model. Hence, we assign the lowest guess number to all of these guesses. This is a strict overestimate of PCFG's guessing effectiveness, but in practice does not change the results.



(a) Transference learning (b) Natural-language dictionaries (c) Tutoring

Figure 2: **Alternative training methods for neural networks.** The *x*-axes represent the number of guesses in log scale. The *y*-axes show the corresponding percentage of 1class16 passwords guessed. In (b), *WithNL* is a neural network trained with natural-language dictionaries, and *NoNL* is a neural network trained without natural-language dictionaries.

Markov Models We trained 4-, 5-, and 6-gram models. Prior work found the 6-gram models and additive smoothing of 0.01 to be an effective configuration for most password sets [65]. Our results agree, and we use the 6-gram model with additive smoothing in our tests. We discard guesses that do not match the target policy.

Mangling Wordlist Methods We compute guess numbers using the popular cracking tools Hashcat and John the Ripper (JtR). For Hashcat, we use the best64 and gen2 rule sets that are included with the software [83]. For JtR, we use the SpiderLabs mangling rules [86]. We chose these sets of rules because prior work found them effective in guessing general-purpose passwords [89]. To create the input for each tool, we uniqued and sorted the respective training set by descending frequency. For JtR, we remove guesses that do not match the target policy. For Hashcat, however, we do not do so because Hashcat's GPU implementation can suffer a significant performance penalty. We believe that this models a real-world scenario where this penalty would also be inflicted.

5 Evaluation

We performed a series of experiments to tune the training of our neural networks and compare them to existing guessing methods. In Section 5.1, we describe experiments to optimize the guessing effectiveness of neural networks by using different training methods. These experiments were chosen primarily to guide our decisions about model parameters and training along the design space we describe in Section 3.2, including training methods, model size, training data, and network architecture. In Section 5.2, we compare the effectiveness of the neural network's guessing to other guessing algorithms. Finally, in Section 5.3, we describe our browser implementation's effectiveness, speed, and size, and we compare it to other browser password-measuring tools.

5.1 Training Neural Networks

We conducted experiments exploring how to tune neural network training, including modifying the network size, using sub-word models, including natural-language dictionaries in training, and exploring alternative architectures. We do not claim that these experiments are a complete exploration of the space. Indeed, improving neural networks is an active area of research.

Transference Learning We find that the transference learning training, described in Section 3.2, improves guessing effectiveness. Figure 2a shows in log scale the effect of transference learning. For example, at 10^{15} guesses, 22% of the test set has been guessed with transference learning, as opposed to 15% without transference learning. Using a 16 MB network, we performed this experiment on our 1class16 passwords because they are particularly different from the majority of our training set. Here, transference learning improves password guessing mostly at higher guess numbers.

Including Natural-Language Dictionaries We experimented with including natural-language dictionaries in the neural network training data, hypothesizing that doing so would improve guessing effectiveness. We performed this experiment with 1class16 passwords because they are particularly likely to benefit from training on natural-language dictionaries [91]. Networks both with and without natural language data were trained using the transference learning method on long passwords. Natural language was included with the primary batch of training data. Figure 2b shows that, contrary to our hypotheses, training on natural language decreases the neural network's guessing effectiveness. We believe neural networks do not benefit from natural language, in contrast to other methods like PCFG, because this method of training does not differentiate between natural-language dictionaries and password training. However, training data could be enhanced with natural language in other ways, perhaps yielding better results.

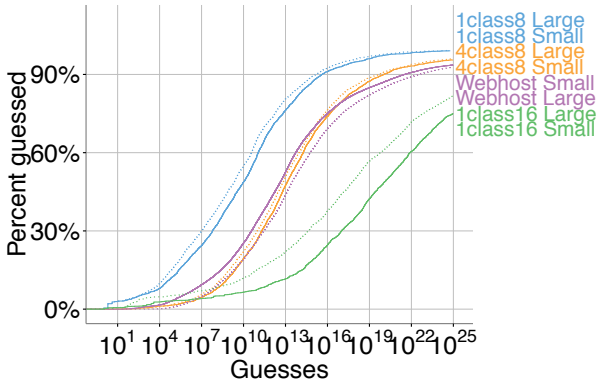


Figure 3: **Neural network size and password guessability.** Dotted lines are large networks; solid lines are small networks.

Password Tokenization We find that using hybrid, sub-word level password models does not significantly increase guessing performance at low guess numbers. Hybrid models may represent the same word in multiple different ways. For example, the model may capture a word as one token, 'pass', or as the letters 'p', 'a', 's', 's'. Because Monte Carlo simulations assume that passwords are uniquely represented, instead of using Monte Carlo methods to estimate guess numbers, we calculated guess numbers by enumerating the most probable 10^7 guesses. However, at this low number of guesses, we show this tokenization has only a minor effect, as shown in Figure 4b. We conducted this experiment on long passwords because we believed that they would benefit most from tokenization. This experiment shows that there may be an early benefit, but otherwise the models learn similarly. We consider this result to be exploratory both due to our low guessing cutoff and because other options for tuning the tokenization could produce better results.

Model Size We find that, for at least some password sets, neural network models can be orders of magnitude smaller than other models with little effect on guessing effectiveness. We tested how the following two model sizes impact guessing effectivess: a large model with 1,000 LSTM cells or 15,700,675 parameters that uses 60 MB, and a small model with 200 LSTM cells or 682,851 parameters that takes 2.7 MB.

The results of these experiments are shown in Figure 3. For 1class8 and 4class8 policies, the effect of decreasing model size is minor but noticeable. However, for 1class16 passwords, the effect is more dramatic. We attribute differences between the longer and shorter policies with respect to model size to fundamental differences in password composition between those policies. Long passwords are more similar to English language phrases, and modeling them may require more parameters, and hence larger networks, than modeling shorter passwords. The webhost test set is the only set for which the larger model performed worse. We believe that this is due to the lack of suitability of the particular training data we used for this model. We discuss the differences in training data more in Section 5.2.

Tutored Networks To improve the effectiveness of our small model at guessing long passwords, we attempted to tutor our small neural network with randomly generated passwords from the larger network. While this had a mild positive effect with light tutoring, at a roughly one to two ratio of random data to real data, the effect does not seem to scale to heavier tutoring. Figure 2c shows minimal difference in guessing accuracy when tutoring is used, and regardless of whether it is light or heavy.

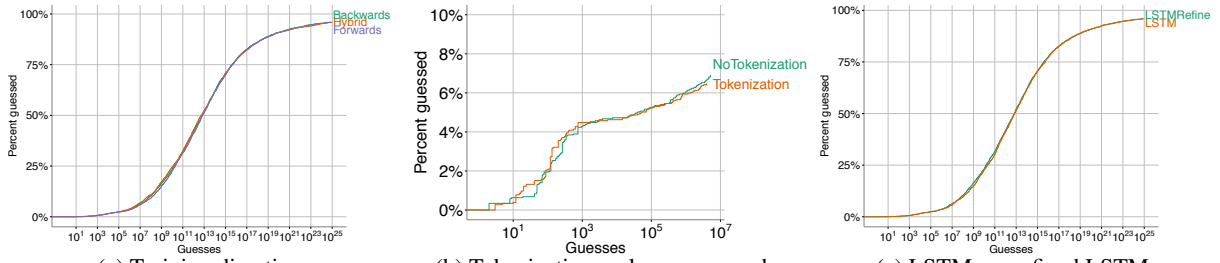
Backwards vs. Forwards Training As described in Section 3.2, processing input backwards rather than forwards can be more effective in some applications of neural networks [48]. We experiment with guessing passwords backwards, forwards, and using a hybrid approach where half of the network examines passwords forwards and the other half backwards. We observed only marginal differences overall. At the point of greatest difference, near 10^9 guesses, the hybrid approach guessed 17.2% of the test set, backwards guessed 16.4% of the test set and forwards guessed 15.1% of the test set. Figure 4a shows the result of this experiment. Since the hybrid approach increases the amount of time required to train with only small improvement in accuracy, for other experiments we use backwards training.

Recurrent Architectures We experimented with two different types of recurrent neural-network architectures: long short-term memory (LSTM) models [54] and a refinement on LSTM models [58]. We found that this choice had little effect on the overall output of the network, with the refined LSTM model being slightly more accurate, as shown in Figure 4c.

5.2 Guessing Effectiveness

Compared to other individual password-guessing methods, we find that neural networks are better at guessing passwords at a higher number of guesses and when targeting more complex or longer password policies, like our 4class8, 1class16, and 3class12 data sets. For example, as shown in Figure 5b, neural networks guessed 70% of 4class8 passwords by 10^{15} guesses, while the next best performing guessing method guesses 57%.

Models differ in how effectively they guess specific passwords. *MinGuess*, shown in Figure 5, represents an idealized guessing approach in which a password is considered guessed as soon as it is guessed by any of our



(a) Training direction. (b) Tokenization on long passwords. (c) LSTM vs. refined LSTM.

Figure 4: **Additional tuning experiments.** Our LSTM experiments tested on complex passwords with 16M parameters. We found very little difference in performance. Our experiments on tokenization examined long passwords. Our experiments on training direction involved training backwards, forwards, and both backwards and forwards with 16M parameters on complex passwords.

guessing approaches, including neural networks, Markov models, PCFG, JtR, and Hashcat. That MinGuess outperforms neural networks suggests that using multiple guessing methods should still be preferred to using any single guessing method for accurate strength estimation, despite the fact that neural networks generally outperform other models individually.

For all the password sets we tested, neural networks outperformed other models beginning at around 10^{10} guesses, and matched or beat the other most effective methods before that point. Figures 5-6 show the performance of the different guessing methods trained with the PGS data set, and Figures 7-8 show the same guessing methods trained with the PGS++ data set. Both data sets are described in more detail in Section 4.1. In this section, we used our large, 15.7 million parameter neural network, trained with transference learning on two training sets. While performance varies across guessing method and training set, in general we find that the neural networks' performance at high guess numbers and across policies holds for both sets of training data with one exception, discussed below. Because these results hold for multiple training and test sets, we hypothesize that neural networks would also performe well in guessing passwords created under many policies that we did not test.

In the webhost test set using the PGS++ training data, neural networks performed worse than other methods. For webhost, all guessing methods using the PGS++ data set were less effective than the PGS data set, though some methods, such as PCFG, were only slightly affected. Because all methods perform worse, and because, when using the PGS training data, neural networks do better than other methods—similar to other test sets—we believe that the PGS++ training data is particularly ineffective for this test set. As Figure 3 shows, this is the only data set where a smaller neural network performs significantly better than the larger neural network, which suggests that the larger neural network model is fitting itself more strictly to low-quality data, which limits the larger network's ability to generalize.

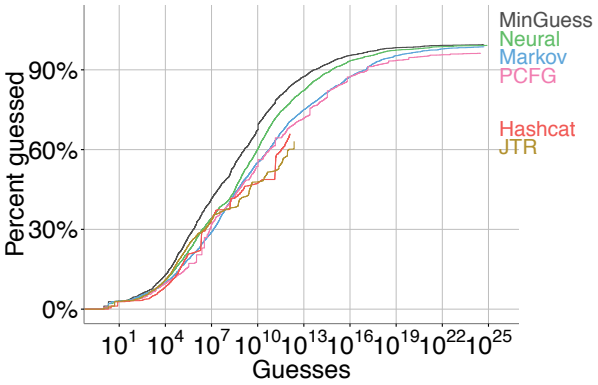
Qualitatively, the types of passwords that our implementation of neural networks guessed before other methods were novel passwords that were dissimilar to passwords in the training set. The types of passwords that our implementation of neural networks were late to guess but that were easily guessable by other methods often were similar to words in the natural-language dictionaries, or were low-frequency occurrences in the training data.

Resource Requirements In general, PCFGs require the most disk, memory, and computational resources. Our PCFG implementation stored its grammar in 4.7GB of disk space. Markov models are the second largest of our implementations, requiring 1.1GB of disk space. Hashcat and JtR do not require large amounts of space for their rules, but do require storing the entire training set, which is 756MB. In contrast, our server-side neural network requires only 60MB of disk space. While 60MB is still larger than what could effectively be transferred to a client without compression, it is a substantial improvement over the other models.

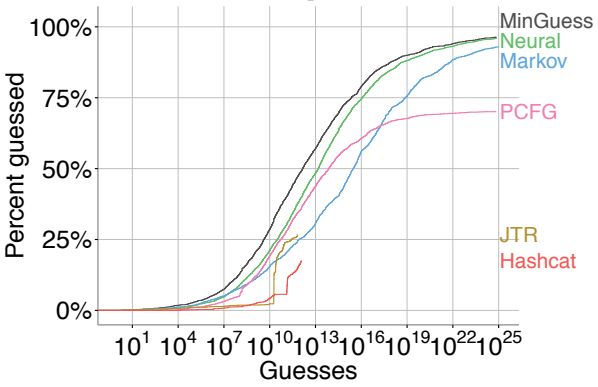
5.3 Browser Implementation

While effective models can fit into 60MB, this is still too large for real-time password feedback in the browser. In this section, we evaluate our techniques for compressing neural network models, discussed in Section 3.3, by comparing the guessing effectiveness of the compressed models to all server-side models—our large neural network, PCFG, Markov models, JtR, and Hashcat.

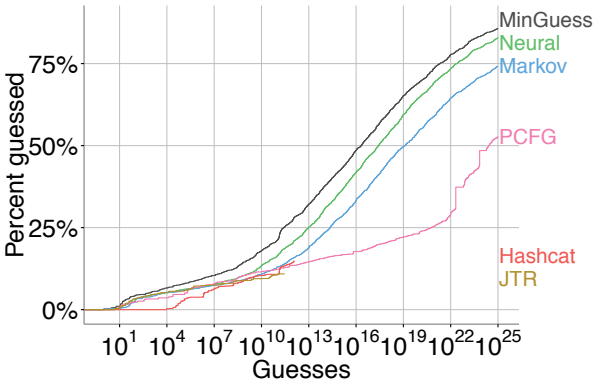
Model Encoding Our primary size metric is the `gzip`ed model size. Our compression stages use the JSON format because of its native support in JavaScript platforms. We explored using the MsgPack binary format [4], but found that after `gzip` compression, there was no benefit for encoding size and minor drawbacks for decoding speed. The effects of different pipeline stages on compression are shown in Table 1.



(a) 1class8 passwords



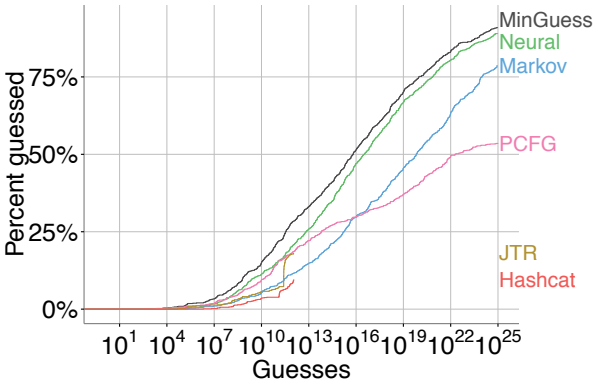
(b) 4class8 passwords



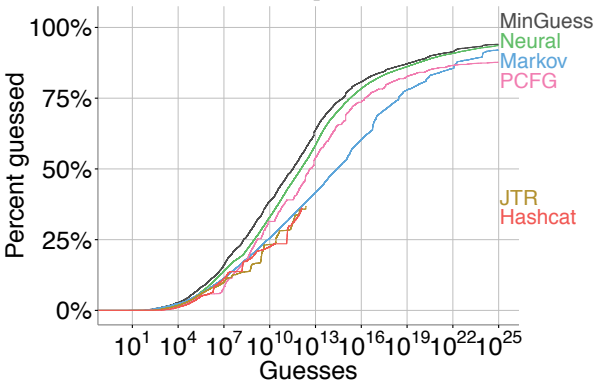
(c) 1class16 passwords

Figure 5: **Guessability of our password sets for different guessing methods using the PGS data set.** *MinGuess* stands for the minimum number of guesses for any approach. Y-axes are differently scaled to best show comparative performance.

Weight and Probability Curve Quantization Because current methods of calculating guess numbers from probabilities are too slow, taking hours or days to return results, we precompute a mapping from password probability to guess number and send the mapping to the client, as described in Section 3.3.2. Such a mapping can be efficiently encoded by quantizing the probability-to-guess-number curve. Quantizing the curve incurs safe errors—i.e., we underestimate the strength of passwords.



(a) 3class12 passwords

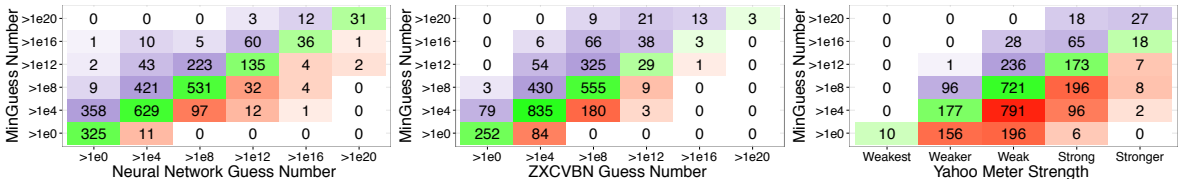


(b) Webhost passwords

Figure 6: **Guessability of our password sets for different guessing methods using the PGS data set (continued).**

We also quantize the model's parameters in the browser implementation to further decrease the size of the model. Both weight and curve quantization are lossy operations, whose effect on guessing we show in Figure 9. Curve quantization manifests in a saw-tooth shape to the guessing curve, but the overall shape of the guessing curve is largely unchanged.

Evaluating Feedback Speed Despite the large amount of computation necessary for computing a password's guessability, our prototype implementation is efficient enough to give real-time user feedback. In general, feedback quicker than 100 ms is perceived as instantaneous [72]; hence, this was our benchmark. We performed two tests to measure the speed of calculating guess numbers: the first measures the time to produce guess numbers with a semi-cached password; the second computes the total time per password. The semi-cached test measures the time to compute a guess number when adding a character to the end of a password. We believe this is representative of what a user would experience in practice because a user typically creates a password by typing it in character by character.



(a) Client-side Neural Network (b) `zxcvbn` (c) Yahoo!

Figure 10: **Client-side guess numbers compared to the minimum guess number of all server-side methods.** The number in the bin represents the number of passwords in that bin. For example, neural networks rated 358 passwords as being guessed with between 10^0 and 10^4 guesses, while server-side approaches rate them as taking between 10^4 and 10^8 guesses. The test passwords are our 1class8 set. The Yahoo! meter does not provide guess numbers and, as such, has a different x-axis. Overestimates of strength are shown in shades of red, underestimates in shades of purple, and accurate estimates in shades of green. Color intensity rises with the number of passwords in a bin.

Pipeline stage	Size	`gzip`-ed Size
Original JSON format	6.9M	2.4M
Quantization	4.1M	716K
Fixed point	3.1M	668K
ZigZag encoding	3.0M	664K
Removing spaces	2.4M	640K

Table 1: **The effect of different pipeline stages on model size.** This table shows the small model that targets the 1class8 password policy, with 682,851 parameters. Each stage includes the previous stage, e.g., the fixed-point stage includes the quantization stage. We use `gzip` at the highest compression level.

		Total	Unsafe
1class8	Neural Network	1311	164
	`zxcvbn`	1331	270
	Yahoo!	1900	984
4class8	Neural Network	1826	115
	`zxcvbn`	1853	231
	Yahoo!	1328	647

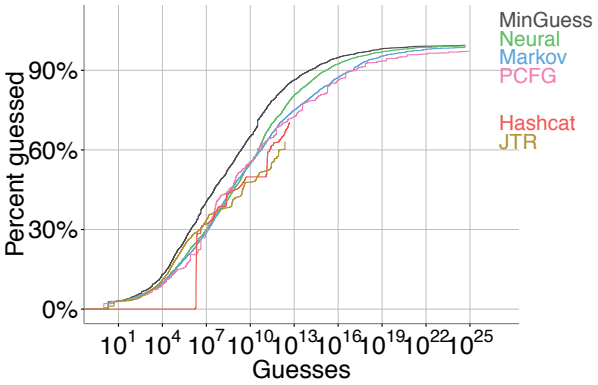
Table 2: **The number of total and unsafe misclassifications for different client-side meters.** Because the Yahoo! meter provides different binning, we pre-process its output for fairer comparison, as described in Section 5.3.

We perform both tests on a laptop running OSX with a 2.7 GHz i7 processor and using the Chrome web browser (version 48). We randomly selected a subset of 500 passwords from our 1class8 training set for these tests. In the semi-cached test, the average time to compute a guess number is 17 ms (stdev: 4 ms); in the full-password test, the average time is 124 ms (stdev: 48 ms). However, both the semi-cached test and the uncached test perform fast enough to give quick feedback to users.

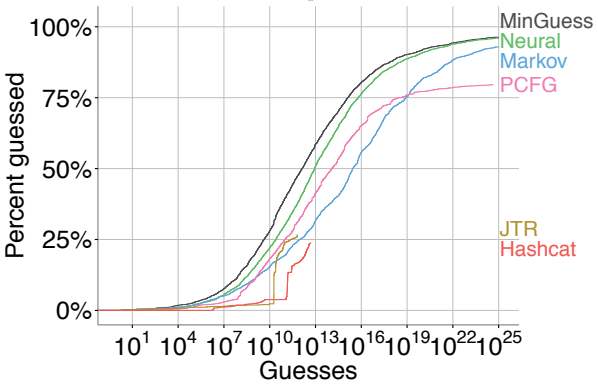
Comparison to Other Password Meters We compared the accuracy of our client-side neural network implementation to other client-side password-strength estimators. Approximations of password strength can be under- or overestimates. We call overestimates of password strength unsafe errors, since they represent passwords as harder to guess than they actually are. We show that our meter can more precisely measure passwords' resistance to guessing with up to half as many unsafe errors as existing client-side models, which are based on heuristics. Our ground truth for this section is the idealized MinGuess method, described in Section 5.2.

Prior work found nearly all proactive password-strength estimators to be inconsistent and to poorly estimate passwords' resistance to guessing [33]. The most promising estimator was Dropbox's `zxcvbn` meter [94, 95], which relies on hand-crafted heuristics, statistical methods, and plaintext dictionaries as training data to estimate guess numbers. Notably, these plaintext dictionaries are not the same as those used for our training data, limiting our ability to fully generalize from these comparisons. Exploring other ways of configuring `zxcvbn` is beyond the scope of this evaluation. We compare our results to both `zxcvbn` and the Yahoo! meter, which is an example of using far less sophisticated heuristics to estimate password strength.

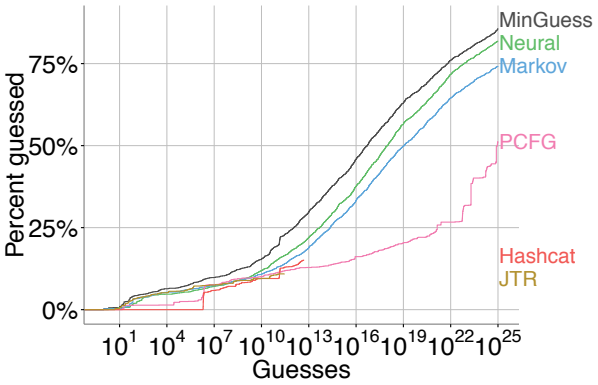
The Yahoo! meter does not produce guess numbers but bins passwords as weakest, weaker, weak, strong, and stronger. We ignore the semantic values of the bin names, and examine the accuracy with which the meter classified passwords with different guess numbers (as computed by the MinGuess of all guessing methods) into the five bins. To compare the Yahoo! meter to our minimum guess number (Table 2), we take the median actual guess number of each bin (e.g., the "weaker" bin) and then map the minimum guess number for each password to the bin that it is closest to on a log scale. For example, in the Yahoo! meter, the guess number of $5.4 \cdot 10^4$ is the median of the "weaker" bin; any password closer to $5.4 \cdot 10^4$ than to the medians of other bins on a log scale we consider as belonging in the "weaker" bin. We intend for this to be an overestimate of the accuracy of



(a) 1class8 passwords

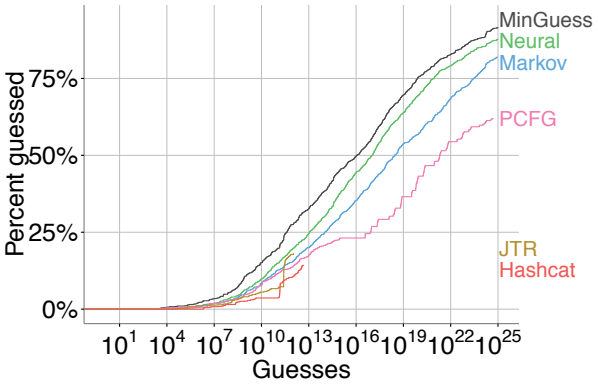


(b) 4class8 passwords

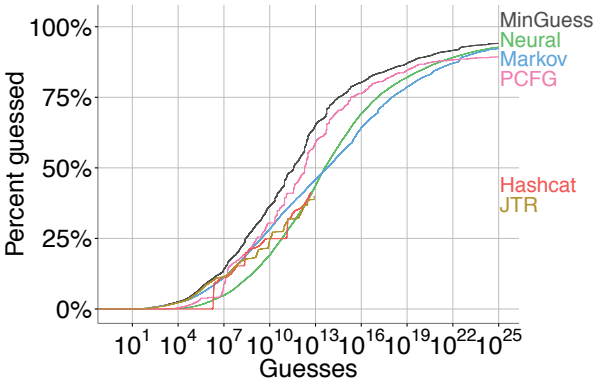


(c) 1class16 passwords

Figure 7: **Guessability of our password sets for different guessing methods using the PGS++ data set.** *MinGuess* stands for the minimum number of guesses for any approach.



(a) 3class12 passwords



(b) Webhost passwords

Figure 8: **Guessability of our password sets for different guessing methods using the PGS++ data set (continued).**

the Yahoo! meter. Nonetheless, both our work and prior work [33] find the Yahoo! meter to be less accurate than other approaches, including the `zxcvbn` meter.

We find that our client-side neural network approach is more accurate than the other approaches we test, with up to two times fewer unsafe errors and comparable safe errors, as shown in Figure 10 and Table 2. Here, we used our neural network meter implementation with the tuning described in Section 3.4. We performed the 1class8 test with the client-side Bloom filter, described in Section 3.3.1, while the 4class8 test did not use the Bloom filter because it did not significantly impact accuracy. Both tests scale the network output down by a factor of 300 and ignore case to give more conservative guess numbers. We chose the scaling factor to tune the network to make about as many safe errors as `zxcvbn`. In addition, we find that, compared to our neural network implementation, the `zxcvbn` meter's errors are often at very low guess numbers, which can be particularly unsafe. For example, for the 10,000 most likely passwords, `zxcvbn` makes 84 unsafe errors, while our neural network only makes 11 unsafe errors.

Besides being more accurate, we believe the neural network approach is easier to apply to other password policies. The best existing meter, `zxcvbn`, is handcrafted to target one specific password policy. On the other hand, neural networks enable easy retargeting to other policies simply by retraining.

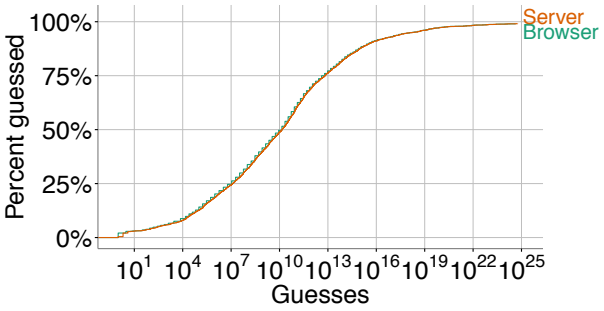


Figure 9: **Compressed browser neural network with weight and curve quantization compared an unquantized network.** *Browser* is our browser network with weight and curve quantization. *Server* is the same small neural network without weight and curve quantization.

6 Conclusion

This paper describes how to use neural networks to model human-chosen passwords and measure password strength. We show how to build and train neural networks that outperform state-of-the-art password-guessing approaches in efficiency and effectiveness, particularly for non-traditional password policies and at guess numbers above 10^{10}. We also demonstrate how to compress neural network password models so that they can be downloaded as part of a web page. This makes it possible to build client-side password meters that provide a good measure of password strength.

Tuning neural networks for password guessing and developing accurate client-side password-strength metrics both remain fertile research grounds. Prior work has used neural networks to learn the output of a larger ensemble of models [24] and obtained better results than our network tutoring (Section 5.1). Other work achieves higher compression ratios for neural networks than we do by using matrix factorization or specialized training methods [51, 96]. Further experiments on leveraging natural language, tokenized models, or other neural-networks architectures might allow passwords to be guessed more effectively. While we measured client-side strength metrics based on guessing effectiveness, a remaining challenge is giving user-interpretable advice to improve passwords during password creation.

7 Acknowledgements

We would like to thank Mahmood Sharif for participating in discussions about neural networks and Dan Wheeler for his feedback. This work was supported in part by gifts from the PNC Center for Financial Services Innovation, Microsoft Research, and John & Claire Bertucci.

References

[1] CSDN password leak. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`.

[2] Faith writer leak. `https://wiki.skullsecurity.org/Passwords#Leaked_passwords`.

[3] Hak5 leak. `https://wiki.skullsecurity.org/Passwords#Leaked_passwords`.

[4] Msgpack: It's like JSON but fast and small. `http://msgpack.org/index.html`.

[5] Neocortex Github repository. `https://github.com/scienceai/neocortex`.

[6] Perl monks password leak. `http://news.softpedia.com/news/PerlMonks-ZFO-Hack-Has-Wider-Implications-118225.shtml`.

[7] Phpbb password leak. `https://wiki.skullsecurity.org/Passwords`.

[8] Protocol buffer encoding. `https://developers.google.com/protocol-buffers/docs/encoding`.

[9] Stratfor leak. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`.

[10] Using Web workers. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`. Accessed:Feb 2016.

[11] The "web2" file of English words. `http://www.bee-man.us/computer/grep/grep.htm#web2`, 2004.

[12] Password leaks: Elitehacker. `https://wiki.skullsecurity.org/Passwords`, 2009.

[13] Password leaks: Alypaa. `https://wiki.skullsecurity.org/Passwords`, 2010.

[14] Specialforces.com password leak. `http://www.databreaches.net/update-specialforces-com-hackers-acquired-8000-credit-card-numbers/`, 2011.

[15] YouPorn password leak, 2012. `http://thepasswordproject.com/leaked_password_lists_and_dictionaries`.

[16] WOM Vegas password leak, 2013. `https://www.hackread.com/wom-vegas-breached-10000-user-accounts-leaked-by-darkweb-goons/`.

[17] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. Theano: New features and speed improvements. In *Proc. NIPS 2012 Deep Learning workshop* (2012).

[18] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: A CPU and GPU math expression compiler. In *Proc. SciPy* (2010).

[19] Bishop, M., and Klein, D. V. Improving system security via proactive password checking. *Computers & Security 14*, 3 (1995), 233–249.

[20] Bonneau, J. The Gawker hack: How a million passwords were lost. *Light Blue Touchpaper* Blog, December 2010. `http://www.lightbluetouchpaper.org/2010/12/15/the-gawker-hack-how-a-million-passwords-were-lost/`.

[21] Bonneau, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE Symp. Security & Privacy* (2012).

[22] Bonneau, J. Statistical metrics for individual password strength. In *Proc. WPS* (2012).

[23] BRODKIN, J. 10 (or so) of the worst passwords exposed by the LinkedIn hack. *Ars Technica*, June 6, 2012. http://arstechnica.com/security/2012/06/10-or-so-of-the-worst-passwords-exposed-by-the-linkedin-hack/.

[24] BUCILUǍ, C., CARUANA, R., AND NICULESCU-MIZIL, A. Model compression. In *Proc. KDD* (2006).

[25] BURNETT, M. Xato password set. https://xato.net/.

[26] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from Markov models. In *Proc. NDSS* (2012).

[27] CHANG, J. M. Passwords and email addresses leaked in Kickstarter hack attack. *ABC News*, Feb 17, 2014. http://abcnews.go.com/Technology/passwords-email-addresses-leaked-kickstarter-hack/story?id=22553952.

[28] CHOLLET, F. Keras Github repository. https://github.com/fchollet/keras.

[29] CHUN, W. WebGL Models: End-to-End. In *OpenGL Insights*. 2012.

[30] CIARAMELLA, A., D'ARCO, P., DE SANTIS, A., GALDI, C., AND TAGLIAFERRI, R. Neural network techniques for proactive password checking. *IEEE TDSC 3*, 4 (2006), 327–339.

[31] CLERCQ, J. D. Resetting the password of the KRBTGT active directory account, 2014. http://windowsitpro.com/security/resetting-password-krbtgt-active-directory-account.

[32] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *Proc. NDSS* (2014).

[33] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Proc. NDSS* (2014).

[34] DELL'AMICO, M., AND FILIPPONE, M. Monte Carlo strength evaluation: Fast and reliable password checking. In *Proc. CCS* (2015).

[35] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *Proc. INFOCOM* (2010).

[36] DUCKETT, C. Login duplication allows 20m Alibaba accounts to be attacked. *ZDNet*, February 5, 2016. http://www.zdnet.com/article/login-duplication-allows-20m-alibaba-accounts-to-be-attacked/.

[37] DÜRMUTH, M., ANGELSTORF, F., CASTELLUCCIA, C., PERITO, D., AND CHAABANE, A. OMEN: Faster password guessing using an ordered markov enumerator. In *Proc. ESSoS* (2015).

[38] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On the ecological validity of a password study. In *Proc. SOUPS* (2013).

[39] FLORÊNCIO, D., HERLEY, C., AND VAN OORSCHOT, P. C. An administrator's guide to internet password research. In *Proc. USENIX LISA* (2014).

[40] FOX-BREWSTER, T. 13 million passwords appear to have leaked from this free web host. *Forbes*, October 28, 2015. http://www.forbes.com/sites/thomasbrewster/2015/10/28/000webhost-database-leak/.

[41] GAILLY, J.-L. gzip. http://www.gzip.org/.

[42] GOODIN, D. 10,000 Hotmail passwords mysteriously leaked to web. *The Register*, October 5, 2009. http://www.theregister.co.uk/2009/10/05/hotmail_passwords_leaked/.

[43] GOODIN, D. Hackers expose 453,000 credentials allegedly taken from Yahoo service. *Ars Technica*, July 12, 2012. http://arstechnica.com/security/2012/07/yahoo-service-hacked/.

[44] GOODIN, D. 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/.

[45] GOODIN, D. Why LivingSocial's 50-million password breach is graver than you may think. *Ars Technica*, April 27, 2013. http://arstechnica.com/security/2013/04/why-livingsocials-50-million-password-breach-is-graver-than-you-may-think/.

[46] GOODIN, D. Once seen as bulletproof, 11 million+ Ashley Madison passwords already cracked. *Ars Technica*, September 10, 2015. http://arstechnica.com/security/2015/09/once-seen-as-bulletproof-11-million-ashley-madison-passwords-already-cracked/.

[47] GOOGLE. Web 1T 5-gram version 1, 2006. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13.

[48] GRAVES, A. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012.

[49] GRAVES, A. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.

[50] GREENBERG, A. 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/.

[51] HAN, S., MAO, H., AND DALLY, W. J. A deep neural network compression pipeline: Pruning, quantization, Huffman encoding. arXiv preprint arXiv:1510.00149, 2015.

[52] HENRY, A. Five best password managers. *LifeHacker*, January 11, 2015. http://lifehacker.com/5529133/.

[53] HERLEY, C., AND VAN OORSCHOT, P. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy Magazine 10*, 1 (Jan. 2012), 28–36.

[54] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[55] HUNT, T. A brief Sony password analysis. http://www.troyhunt.com/2011/06/brief-sony-password-analysis.html, 2011.

[56] HUYNH, T. ABC Australia hacked nearly 50,000 user credentials posted online, half cracked in 45 secs. *Techgeek*, February 27, 2013. http://techgeek.com.au/2013/02/27/abc-australia-hacked-nearly-50000-user-credentials-posted-online/.

[57] JOHNSTONE, L. 9,885 user accounts leaked from Intercessors for America by Anonymous. http://www.cyberwarnews.info/2013/07/24/9885-user-accounts-leaked-from-intercessors-for-america-by-anonymous/, 2013.

[58] JOZEFOWICZ, R., ZAREMBA, W., AND SUTSKEVER, I. An empirical exploration of recurrent network architectures. In *Proc. ICML* (2015).

[59] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE Symp. Security & Privacy* (2012).

[60] KOMANDURI, S. *Modeling the adversary to evaluate password strengh with limited samples*. PhD thesis, Carnegie Mellon University, 2016.

[61] KOMANDURI, S., SHAY, R., CRANOR, L. F., HERLEY, C., AND SCHECHTER, S. Telepathwords: Preventing weak passwords by reading users' minds. In *Proc. USENIX Security* (2014).

[62] KREBS, B. Fraud bazaar carders.cc hacked. http://krebsonsecurity.com/2010/05/fraud-bazaar-carders-cc-hacked/.

[63] LEE, M. Hackers have released what they claim are the details of over 21,000 user accounts belonging to Billabong customers. *ZDNet*, July 13, 2012. http://www.zdnet.com/article/over-21000-plain-text-passwords-stolen-from-billabong/.

[64] LOWERRE, B. T. *The HARPY speech recognition system*. PhD thesis, Carnegie Mellon University, 1976.

[65] MA, J., YANG, W., LUO, M., AND LI, N. A study of probabilistic password models. In *Proc. IEEE Symp. Security & Privacy* (2014).

[66] MAZUREK, M. L., KOMANDURI, S., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., KELLEY, P. G., SHAY, R., AND UR, B. Measuring password guessability for an entire university. In *Proc. CCS* (2013).

[67] MCALLISTER, N. Twitter breach leaks emails, passwords of 250,000 users. *The Register*, Feb 2, 2013.

[68] MIKOLOV, T., SUTSKEVER, I., DEORAS, A., LE, H.-S., KOMBRINK, S., AND CERNOCKY, J. Subword language modeling with neural networks. Preprint (http://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf), 2012.

[69] MITZENMACHER, M. Compressed Bloom filters. *IEEE/ACM Transactions on Networking (TON) 10*, 5 (2002), 604–612.

[70] NARAYANAN, A., AND SHMATIKOV, V. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. CCS* (2005).

[71] NEEF, S. Using neural networks for password cracking. Blog post. https://0day.work/using-neural-networks-for-password-cracking/, 2016.

[72] NIELSEN, J., AND HACKOS, J. T. *Usability engineering*, vol. 125184069. Academic press Boston, 1993.

[73] PERLROTH, N. Adobe hacking attack was bigger than previously thought. *The New York Times Bits Blog*, October 29, 2013. http://bits.blogs.nytimes.com/2013/10/29/adobe-online-attack-was-bigger-than-previously-thought/.

[74] PESLYAK, A. John the Ripper. http://www.openwall.com/john/, 1996-.

[75] PROTALINSKI, E. 8.24 million Gamigo passwords leaked after hack. *ZDNet*, July 23, 2012. http://www.zdnet.com/article/8-24-million-gamigo-passwords-leaked-after-hack/.

[76] RAGAN, S. Mozilla's bug tracking portal compromised, reused passwords to blame. *CSO*, September 4, 2015. http://www.csoonline.com/article/2980758/.

[77] SCHNEIER, B. Myspace passwords aren't so dumb. http://www.wired.com/politics/security/commentary/securitymatters/2006/12/72300, 2006.

[78] SCOWL. Spell checker oriented word lists. http://wordlist.sourceforge.net, 2015.

[79] SHAY, R., BAUER, L., CHRISTIN, N., CRANOR, L. F., FORGET, A., KOMANDURI, S., MAZUREK, M. L., MELICHER, W., SEGRETI, S. M., AND UR, B. A spoonful of sugar? The impact of guidance and feedback on password-creation behavior. In *Proc. CHI* (2015).

[80] SHAY, R., KOMANDURI, S., DURITY, A. L., HUH, P. S., MAZUREK, M. L., SEGRETI, S. M., UR, B., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Can long passwords be secure and usable? In *Proc. CHI* (2014).

[81] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on SSH. In *Proc. USENIX Security Symposium* (2001).

[82] SPIDEROAK. Zero knowledge cloud solutions. https://spideroak.com/, 2016.

[83] STEUBE, J. Hashcat. https://hashcat.net/oclhashcat/, 2009-.

[84] SUTSKEVER, I., MARTENS, J., AND HINTON, G. E. Generating text with recurrent neural networks. In *Proc. ICML* (2011).

[85] TRUSTWAVE. eHarmony password dump analysis, June 2012. http://blog.spiderlabs.com/2012/06/eharmony-password-dump-analysis.html.

[86] TRUSTWAVE SPIDERLABS. SpiderLabs/KoreLogic-Rules. https://github.com/SpiderLabs/KoreLogic-Rules, 2012.

[87] TSUKAYAMA, H. Evernote hacked; millions must change passwords. *Washington Post*, March 4, 2013. https://www.washingtonpost.com/8279306c-84c7-11e2-98a3-b3db6b9ac586_story.html.

[88] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? The effect of strength meters on password creation. In *Proc. USENIX Security* (2012).

[89] UR, B., SEGRETI, S. M., BAUER, L., CHRISTIN, N., CRANOR, L. F., KOMANDURI, S., KURILOVA, D., MAZUREK, M. L., MELICHER, W., AND SHAY, R. Measuring real-world accuracies and biases in modeling password guessability. In *Proc. USENIX Security* (2015).

[90] VANCE, A. If your password is 123456, just make it hackme. *New York Times*, January 20, 2010. http://www.nytimes.com/2010/01/21/technology/21password.html.

[91] VERAS, R., COLLINS, C., AND THORPE, J. On the semantic patterns of passwords and their security impact. In *Proc. NDSS* (2014).

[92] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. CCS* (2010).

[93] WEIR, M., AGGARWAL, S., MEDEIROS, B. D., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *Proc. IEEE Symp. Security & Privacy* (2009).

[94] WHEELER, D. zxcvbn: Realistic password strength estimation. https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/, 2012.

[95] WHEELER, D. L. zxcvbn: Low-budget password strength estimation. In *Proc. USENIX Security* (2016).

[96] XUE, J., LI, J., YU, D., SELTZER, M., AND GONG, Y. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Proc. ICASSP* (2014).

[97] YOSINSKI, J., CLUNE, J., BENGIO, Y., AND LIPSON, H. How transferable are features in deep neural networks? In *Proc. NIPS* (2014).