Modeling Password Guessing with Neural Networks

Leo de Castro, Hunter Lang, Stephanie Liu, Cristina Mata
{ldecastr, hjl, stliu, cfmata}@mit.edu

***Abstract*— Passwords still dominate the authentication space, but they are vulnerable to many different attacks; in recent years, guessing attacks in particular have notably caused a few high-profile information leaks. Password strength checkers, which attempt to guard against guessing attacks by enforcing heuristics like length and character variety, only model resistance to these brute-force attacks by proxy. In this paper, we reproduce the work of Melicher et. al [1], designing a strength checker that *directly* bases its ratings on resistance to guessing attacks. We provide a qualitative comparison of the ratings output by this system to those of a popular heuristic one, zxcvbn.**

I. INTRODUCTION

Even with recent innovations in authentication protocols, passwords remain one of the most common forms of authentication online, despite their flaws. Unfortunately, passwords are vulnerable to brute-force guessing attacks like the one used to steal celebrity photos from Apple's iCloud in 2014 [2]. The systems designed to counter these brute-force attacks—password strength checkers—are largely based on heuristics, such as length and character variety, and do not directly model the resistance of a password to an adversarial guessing attack.

With this motivation in mind, we aim to create a better password strength checker that *directly* models how an adversary might try to guess a user's password. In a brute-force guessing attack, a smart adversary tries to guess passwords by iterating through a list of passwords in descending order of probability rather than uniformly at random. To model this "worst-case" (best at guessing) adversary, our strength meter bases a password's rating on the number of guesses it would take such an adversary to arrive at that password given his probability ordering.

We focus on guessing attacks here because they are relevant to almost every application that uses passwords as the main form of authentication, and they require no knowledge about the user. Moreover, guessing attacks are being used in the real world to crack passwords, as the Apple leak demonstrates. Encouraging users to choose passwords that occur late in an adversarial ordering is therefore an important step to securing online accounts.

We generate these guess numbers in two parts. First, we use an LSTM recurrent neural network trained on one million leaked passwords to learn a generative model for passwords. The model outputs the probability of a given string being a password, and sampling from the model can generate passwords that did not occur in the training data. This allows us to generate a probability distribution over the universe of possible passwords. We then convert those password probabilities into guess number estimates using a Monte Carlo technique discussed in more detail in Section III. Finally, we threshold those guess numbers to assign passwords to different buckets of password strength and compare our assignments to those of other password strength meters. Our goal, then, is to reproduce the main strength-checking component of [1].

The remainder of this paper is organized as follows. Section II gives a brief and by no means exhaustive summary of previous work on strength checkers and guess number estimation. Section III provides an overview of the two-part strength checker design, and Sections III-A and III-B explain the implementations of each part. Section IV evaluates our checker by qualitatively comparing our strength ratings to those of zxcvbn, a popular heuristic-based password strength meter. Finally, Sections V and VI discuss shortcomings of this method and opportunities for future work, respectively.

II. PREVIOUS WORK

The password strength assessment literature has developed a precise definition of an adversarial guessing attack. This attack is performed by an adversary who has access to a universe of possible passwords as well as a probability distribution over all passwords in this universe. This probability distribution defines an ordering of the passwords, and the adversary uses the ordering to guess passwords in probabilistic order. Intuitively, this definition models an attacker's inclination to guess common passwords before uncommon passwords in a brute-force attack. To accurately model a password's strength against such an attack, a password-strength checker must accurately predict how many guesses the adversary is likely to make before the user's password is guessed. We refer to this quantity as a password's *guess number*.

When creating a new password, users often check their password against simple heuristics like the number or variety of characters. Although common, this technique is a poor model of a real-world guessing attack. The main advantage of this heuristic technique is that it can be run on the client-side of a web application. Other password-strength assessors such as Probabilistic Context-Free Grammars, Markov Models, and Mangled Word Lists can provide accurate estimations of password guess numbers, but they can take gigabytes of disk space and days to produce their results. Even efficient implementations of these models are infeasible to run on the client-side, and once the password leaves the client's machine it is no longer assumed to be secure.

In the search for an efficient, client-side model of a password guessing attack, researchers turned to neural networks. Neural networks have been shown to be effective at generating novel sequences that contain complex dependencies (e.g. passwords), making them ideal candidates for probabilistic password sampling. Neural networks can also be compressed to sizes easily runnable on client-side machines. With both accurate and efficient password strength estimation, neural networks are a promising new direction in password strength assessment technology. In their paper [1], Melicher et al. develop a neural network password strength checker that directly models an adversarial guessing attack. They also showed how this strength checker can be compressed to mere kilobytes without loss of accuracy. In our work, we focus on the use of neural networks as a novel password strength checker, re-implementing and comparing the neural-network-based checker of [1] to other common systems.

When trained on millions of leaked passwords, neural networks are effective at simulating the probability distribution over passwords that an adversary would likely use during a guessing attack. To convert these password probabilities into guess numbers, Melicher et al. use a Monte Carlo technique described in [3]. This technique allows for accurate and efficient estimations of a password's guess number. The Monte Carlo method uses a black-box model that samples passwords from a universe according to the probabilities assigned by the model. The two parts of our system—the neural model for password probabilities and the Monte Carlo guess number estimator—are the focus of the following section.

III. METHODS

In this section, we detail the two parts of the neural strength checker, following the work of [1]. Section III-A explains the role of the neural network in the system and describes our implementation and the training process. Section III-B covers the Monte Carlo simulation method that goes from probabilities output by the neural model to guess numbers for passwords and ultimately to strength scores. Figure 1 summarizes the different components of our system.

A. Neural Network Password Model

1) Sampling and Probabilities: We use a neural network to create a probabilistic model for passwords. Neural networks are computational models that loosely emulate biological neurons. Given a large amount of training data, neural networks can learn to predict patterns and even generate new patterns. In our project we use a Long Short Term Memory (LSTM) neural network, which is a recurrent architecture that can "remember" values over short or long periods of time (where time is measured in input sequence steps). Because recurrent neural networks can generate previously unseen sequences, even passwords that do not appear in the training data will be assigned nonzero probabilities by our model. The model we use operates at the *character level*, meaning that it takes as input a string of characters and outputs a probability distribution over the next character. An example is illustrated in Figure 2.

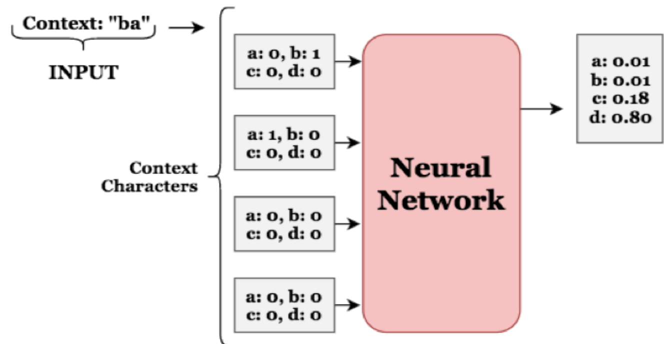

Fig. 2. Character-level NN.

Here, the context characters are "ba", and the output of the model is a conditional probability distribution over the next possible characters. If we consider a simple universe where the only possible characters are `a, b, c, d`, the corresponding distribution is `a: 0.1, b: 0.01, c: 0.18, and d: 0.80`. The most likely next character is "d", which makes sense in our application because "bad" is a common English word. To get the probability of the string "bad", we could keep track of the conditional probabilities of "a" and "b" (which occurred earlier in the string), along with the probability of "d" given context "ba", and multiply them all together.

As this example suggests, given conditional probabilities for characters, our model can take a full string as input and return the probability of that string being a password. The process for computing the total probability of a string is described in detail in Algorithm 1 (it is a simple application of the "chain rule" for conditional probability).

Algorithm 1: Computing the probability of a password.

Data: password, model
Result: P_{model}(password)
$i = 0$;
context = `"\n"`;
runningProb = 1;
while $i <$ *len(password)* **do**
  p = model.getProbabilityOfNext(context);
  runningProb *= p[password[i]];
  context.append(password[i])
end
return runningProb;

Typically, a special start character s is included in the set of all possible characters and is used to signify the start of a string. In our example above, the probability of the first character "b" would actually be the conditional probability of "b" occurring given the context character s. In our model we use the newline character as the start character.

In order for our Monte Carlo simulation to generate a guess number for a password, it needs to be able to sample

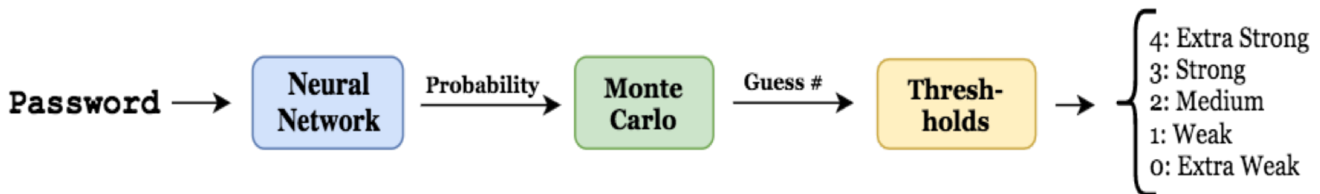

Fig. 1. Overview of the strength checker system.

from the probability distribution (learned by the neural network) over the universe of passwords. The process for sampling from this distribution is described in Algorithm 2. When a newline character is generated, the current password has ended, and the network resets the context so that the next character generated will be at the start of a new password. As explained in the section below, generating a smaller-sized sample of new passwords rather than a vast universe is sufficient for our Monte Carlo simulation.

Algorithm 2: Sampling from the learned distribution

```
Data: model, n
Result: n passwords sampled from model
i = 0;
end = "\n";
passwords = [];
probabilities = [];
while i < n do
    context = end;
    runningProb = 1;
    char = '';
    while char ≠ end do
        p = model.getProbabilityOfNext(context);
        nextChar = sampleFrom(p);
        runningProb *= p[nextChar];
        context.append(nextChar);
        char = nextChar;
    end
    probabilities.append(runningProb);
    passwords.append(context[1:len(context)-1];
end
return passwords, probabilities;
```

The Monte Carlo component described in the following section only needs to be able to compute the probability of a password and sample from the learned distribution of passwords, so Algorithms 1 and 2 are enough for it to do its work. Now we describe the training process for the neural network.

2) Data and Training: For our training data we used leaked passwords found online. However, because of the sensitive nature of the application, we initially found it difficult to find the leaked password files themselves despite the large number of news articles covering the leaks. We contacted one of the authors of Melicher et al., Saranga Komanduri, to aid us in our search, and he shared the RockYou leaked password set. This set contains around 14 million passwords—more than enough for our purposes. Each line of the file contains a password and the number of occurrences of that password. Some preprocessing was done to convert the occurrence numbers from hexadecimal exponential notation to integers. We initially split the data in half for training and testing, but realized that even training on half of the set would take quite a long time. We settled on using one million passwords to train. To have our training data reflect the number of occurrences of each password, we created a larger file with each password repeated the number of times it occurs in the RockYou set, and then took a uniform sample of one million passwords from this larger file. Using this method, passwords should appear in the training data with roughly the same frequency with which they appear in the full RockYou data. The format of the training data input to the model code was a simple newline-separated list of password strings.

To train the network itself, we used Tensorflow, an open-source deep learning library created by Google. Specifically, we used the popular `char-rnn-tensorflow` project, which implements a highly configurable character-level recurrent neural network. The network we trained on the 1 million RockYou passwords has three LSTM layers with 1024 nodes and was trained on an NVIDIA Titan X GPU for roughly 12 hours. Figure 3 shows the training loss of the model over time. Once the model was trained, we had to completely rewrite the `char-rnn-tensorflow` sampling function to work for passwords rather than raw text. The provided sample function took as input a number of characters and only sampled that many characters without ever resetting the context. This is not correct for sampling passwords, since the context needs to be reset after each newline character is predicted—as in the training data, newlines signal the end of the current password and the beginning of the next one. We implemented Algorithm 2 as the new sampling function. We also added an Algorithm 1-style probability computation to the project, since in the original code there was no way to get the probability, according to the model, of an arbitrary input string.

All of our code for the neural network, the Monte Carlo component explained below, and the evaluation scripts that generated the results in Section IV is written in Python and can be found at https://github.mit.edu/cfmata/6.857finalproject.

Both Algorithms 1 and 2 actually keep track of the running sum of the log probabilities to avoid underflow errors.

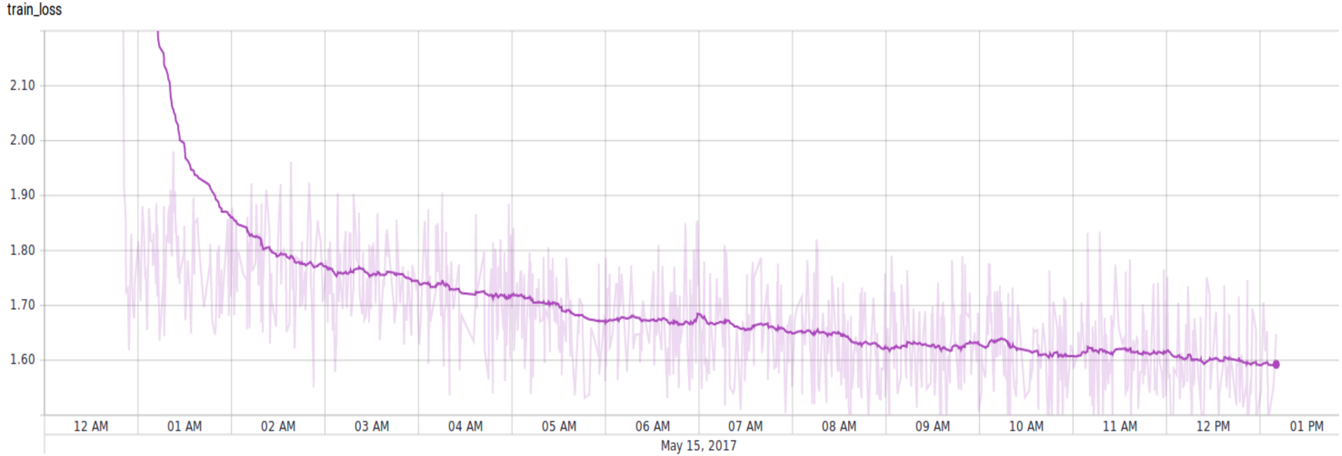

Fig. 3. Smoothed training loss at each training step.

B. Monte Carlo Guess Number Estimation

A password's guess number is defined as the number of passwords that will likely be guessed before the password. Given a universe of passwords Γ, we define a probability function $p\colon \Gamma \mapsto [0, 1)$ that takes a password $\alpha \in \Gamma$ and outputs the probability of α being used as a password. The probability function p is such that

$$\sum_{\alpha \in \Gamma} p(\alpha) = 1$$

In our execution of this technique, we used the neural network to compute the probability of a string being a password.

To compute the guess number of $\alpha \in \Gamma$, we need to estimate the number of passwords $\beta \in \Gamma$ such that $p(\beta) > p(\alpha)$. These passwords β are the passwords that will likely be guessed before α. The goal of a guess number calculation is to estimate the size of the set Δ, defined as:

$$\Delta = \{\beta \in \Gamma \;:\; p(\beta) > p(\alpha)\}$$

One way to determine the cardinality of Δ is brute-force enumeration, which is far too inefficient. Instead, the Monte Carlo method described in [3] generates a sample of n passwords Θ. Passwords are sampled with replacement as described in the previous section. The sampling method must be consistent with the probability function p such that the probability of sampling a password β must be $p(\beta)$. Sampling passwords from the neural network satisfies this condition.

Once we've sampled n passwords from the neural network, we compute the following sum:

$$C_\Delta = \sum_{\beta \in \Theta} \begin{cases} \frac{1}{n \cdot p(\beta)} & p(\beta) > p(\alpha), \\ 0 & \text{otherwise.} \end{cases}$$

It is shown in [3] that the expected value of C_Δ is $|\Delta|$, and the standard deviation of C_Δ is $O(\frac{1}{\sqrt{n}})$. Taking a sufficiently large sample from the neural network allows us to confidently estimate a password's guess number.

This method is convenient because it uses a probability distribution and sampling method as a black-box, making it easily applicable to new password models. Computing the estimate C_Δ concludes our password strength checking system–the neural network learns a probability distribution over passwords and the Monte Carlo algorithm estimates the guess number. It remains to evaluate the performance of this system on real-life password data.

IV. EVALUATION

To evaluate our strength checker, we give both qualitative and quantitative comparisons of its performance to a popular heuristic system called zxcvbn [4]. Developed during a Dropbox hackathon, this system gives passwords a strength score from zero to four, so in order to compare our system we also need to generate scores in this range by setting guess number cutoffs for each score bin. To set these cutoffs, we plotted a cumulative distribution of passwords over guess numbers. A point at $(1000000, 50)$ means 50 percent of passwords have guess number one million or lower according to our model. Using this plot, we can draw in cutoffs as vertical lines, choosing values that distribute passwords across the five bins.

We used two new sources of password data when plotting these CDFs: phpBB and FaithWriters, two sites that have suffered from password leaks. For each site, we chose 5000 passwords uniformly at random to use as our evaluation set. We chose not to use the RockYou leak since we trained on that data; our model should be able to generalize to passwords from other sites. Figure 4 shows the CDFs for our model with phpBB in black and FaithWriters in red. Figure 5 shows the corresponding CDFs for zxcvbn, which also generates its scores by thresholding a "guess number" that they use heuristics to calculate.

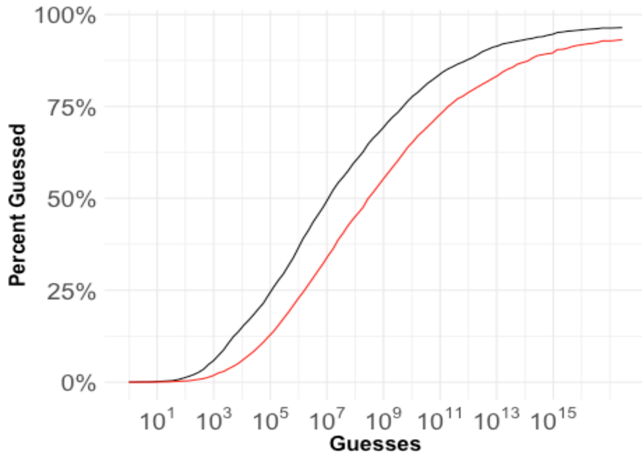

Fig. 4. Guess number CDF for our model with phpBB results in black and FaithWriters results in red.

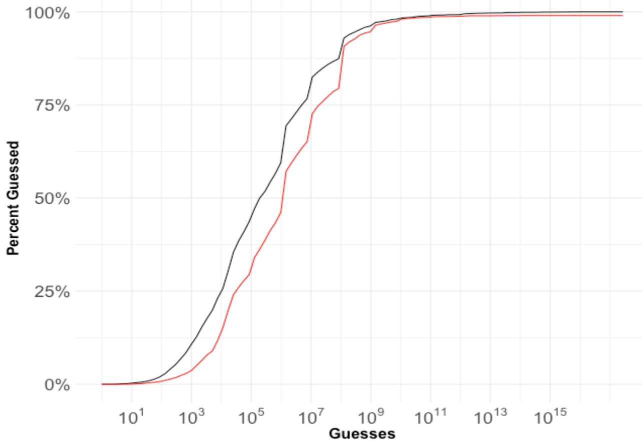

Fig. 5. Guess number CDF for zxcvbn with phpBB results in black and FaithWriters results in red.

Based on Figure 4, we settled on `[1E3, 1E6, 1E9, 1E12]` as our bin cutoffs, hoping to put roughly the same percentage of passwords in each bin as zxcvbn, which uses `[1E3, 1E6, 1E8, 1E10]`. The cutoffs are different, but the guess numbers generated by the two systems are also different, and cutoffs depend on guess numbers. We chose our thresholds so that the percentage of passwords in each bin (with our guess numbers and our cutoffs) would be close to the percentage of passwords in each bin assigned by zxcvbn (with their guess numbers and their cutoffs).

Given our thresholds, we can generate scores on the same scale as zxcvbn's. Figures 6 and 7 are bin plots for phpBB and FaithWriters, respectively. Starting from the bottom left corner, cell (i, j), $0 \leq i, j \leq 4$, represents the percentage of the 5000 passwords whose zxcvbn score is i and whose neural Monte Carlo score is j. The guess number cutoffs for each bin and each system are included on the axes. Green represents agreement, purple means the neural system gave the password a higher score than zxcvbn, and red the reverse. The color intensity gives a visual representation of the fraction of passwords contained in that cell; darker cells contain more of the passwords.

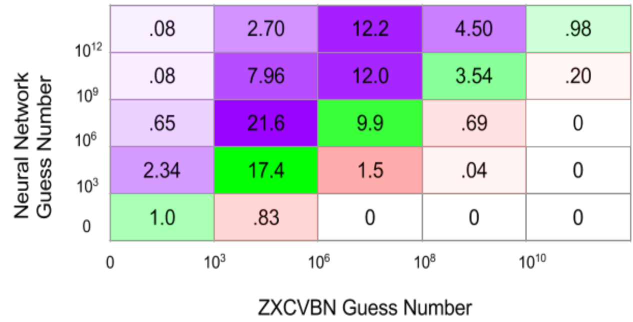

Fig. 6. Our scores vs zxcvbn, phpBB.

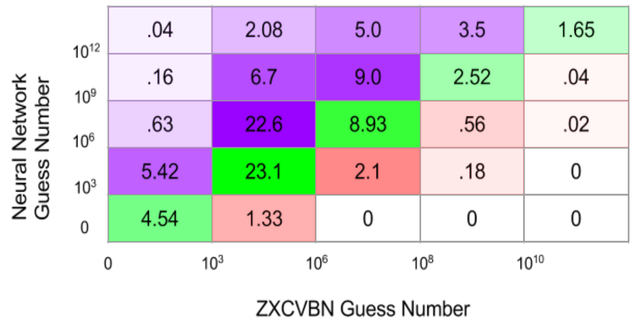

Fig. 7. Our scores vs zxcvbn, FaithWriters.

The amount of purple in these figures suggests our system may be systematically overestimating the strength of passwords compared to zxcvbn. We return to this observation in the following section.

The correlation between our strength scores and the strength scores produced by zxcvbn for the FaithWriters data is 0.659. The correlation between our strength scores and the strength scores produced by zxcvbn for the phpBB data is 0.604. Both of these correlations were computed with 5000 passwords from each source. Upon closer examination of the data, it is clear that a small minority of extreme outliers are skewing these correlations. These outliers and the general trend of the data are discussed in Section V.

V. DISCUSSION

Our neural network model computes the probability of a string being a password by taking successive probabilities of characters conditioned on all previous characters. This strategy relies heavily on the neural network learning patterns in passwords well enough to predict common patterns. As the strings grow in length, these patterns become more difficult for the network to predict, making the conditional probabilities shrink (and the guess numbers grow). This leads to a phenomenon we noticed when comparing guess numbers predicted by our network with those from zxcvbn.

For long, patterned passwords, such as those composed of several English words, our neural network predicted very

large guess numbers, whereas zxcvbn was able to detect the English words directly and assign only a moderate strength score. Some examples include:

- Password: "PROPERTYCUSTODIAN"
  Our guess number: 5.01×10^{17}
  Our strength score: 4
  zxcvbn guess number: 5.01×10^{7}
  zxcvbn strength score: 2

- Password: "ohboyohboyohboy"
  Our guess number: 4.63×10^{16}
  Our strength score: 4
  zxcvbn guess number: 64635.33
  zxcvbn strength score: 1

- Password: "larrylawrence"
  Our guess number: 8.27×10^{13}
  Our strength score: 4
  zxcvbn guess number: 16300.09
  zxcvbn strength score: 1

These examples illustrate the key differences between zxcvbn and our neural network model. In zxcvbn, a password is first analyzed for basic structure, such as English words, words with letters replaced by characters (i.e. apple to @pp13), and common names, and then assessed based on how much it conforms to a structure. Zxcvbn assumes that passwords with recognizable, natural-language structures will be more easily guessed, and therefore less secure, than more randomized passwords. Passwords that are common English words, then, will be assigned weak scores by zxcvbn.

This is fundamentally different from the neural network approach, which attempts to learn the structure of the input from the training data. If the training data does not contain a sufficient variation of possible inputs, the neural network will assign low probability to patterns that never occur in the training data. For instance, if there are no passwords in the training data that contain upper-case letters, the neural network is unlikely to produce a sampled password with an upper case letter even though it is capable of producing a lower-case password that did not appear in the training set. In our case, though, the confounding parameter is length; as the input string becomes longer and longer, the neural network begins predicting that the ending character[1] was more likely than any other character, since not many very long passwords appear in our training data. This raises a fundamental problem of correctly assessing long passwords in models based on successive conditional probabilities.

One approach to this problem may simply be to train the neural network on larger, more diverse data sets. Our neural network was trained on about 1 million passwords, while the neural networks used in [1] were trained on millions of passwords. However, it is not clear that more training data is a scalable solution to this problem, as a 15 character password like "ohboyohboyohboy" will still likely be difficult to predict by a network trained on common passwords. Most passwords seen today do not approach this character length. As password standards become more secure and increased length becomes necessary for security, the use of common phrases or short sentences as passwords will likely grow, since passwords with natural-language structure are easier to remember.

A tool like zxcvbn will be able to scale with this trend, since searching directly for common English words will allow it to detect and properly score a highly structured password of an almost arbitrary length. More generally, a password checker that assesses a candidate password based on some predetermined structure will always be reasonably effective as long as the presence or absence of that structure is a decent proxy for the strength of the password. Proxy-based methods seem to be able to scale better to longer passwords than the neural method explained here, as long as the structure measured by the proxy-based method is maintained. Since the strength of a password is defined as its resistance to a guessing attack, an increased popularity of longer, natural-language passwords will make these passwords more likely to be guessed, which, in turn, increases the effectiveness of the proxy-based method to measure actual password strength.

In order for a neural network to accurately predict this complex yet deterministic structure of natural-language passwords, an approach more sophisticated than character analysis will likely be necessary. This is discussed further in section VI.

VI. FUTURE WORK

Our methods show that neural networks perform well at estimating password strength against adversarial guessing attacks. Due to time constraints, we were unable to experiment with as many augmented models and optimizations as we would have liked to, but there are certainly many possibilities for enhancements to our model.

For example, future work in this area might leverage different natural language dictionaries, such as common first and last names and English words, which have been shown to improve the performance of probabilistic methods using Probabilistic Context Free Grammars (PCFGs). Research has shown that using natural language dictionaries to instantiate PCFG terminals improved their performance for longer passwords, which suggests that they may improve neural network results as well [1].

In a similar vein, it might also be useful to train the neural network model with mangled strings of training data (such as bear and Be@r). These mangled strings are intended to model the way humans create passwords. In fact, mangled wordlists are a common feature of many software tools for adversarial password cracking, such as John the Ripper or Hashcat. An alternative to training the neural network on mangled wordlist data is to mangle the strings output by the model, which may give a wider range of password guesses.

[1] This is a special character added to the alphabet to delineate the end of a password. In our case, as above, that character was the newline.

Furthermore, our current model only outputs password strengths, but provides no feedback on how to make passwords stronger or what makes our model's predictions different from those of password strength meters based on heuristics. In an effort to give users accurate feedback about the strength of their passwords, it would be useful for the system to give users advice on how to generate strong passwords in the first place. This may be done by looking for common patterns between passwords with high guess numbers. As discussed above, our model assigned higher guess numbers to longer passwords, so it may be useful for users to know that a longer password length is desirable. However, length does not suffice to make a password strong, so it is important to explore combinations of password elements and their effects on guess numbers.

When training our model we did not distinguish between different *password policies*—sets of passwords that follow certain rules. For example, some websites only require user passwords to be at least eight characters long, while others require a combination of numbers and uppercase and lowercase letters. We call the set of passwords that fits a certain policy a class of passwords. Training the model on a certain class of password data may yield better strength estimates for passwords within that class. However, this may become problematic for nontraditional policies. Neural networks perform better with more training data, and it is hard to find password data that fits nontraditional policies.

Transference learning is an area of further research that may help create a better model when there is a lack of training data for nontraditional policies. In transference learning, the total training set contains both general passwords and passwords that fit a specific policy. The model is first trained on all the passwords in the set, and then the lower layers of the network are frozen. The model is then retrained only on passwords that fit the policy. The intuition is that the lower layers of the network learn low-level features of the data while the higher layers learn higher-level features. For example, a low-level feature might be that 'a' is a vowel, while a higher-level feature is that 'a' is often followed by a consonant. Melicher et al. found that transference learning improved their network's guessing effectiveness, particularly for higher guess numbers over nontraditional policies. Transference learning minimizes the amount of nontraditional password data needed for the network to produce accurate results for a specific policy.

The long-term goal for accurate password strength checkers is instant feedback. One of the largest challenges with providing instant feedback on password strength is memory. Neural networks can take up megabytes or gigabytes of disk space and would not be suitable for instant feedback. However, they may be compressed by methods such as weight quantization, where the weights of the neural network are only denoted by the most significant digits to reduce the size of their representation. Melicher et al. showed that it is possible to compress an implementation of the neural network and accompanying functionality to a size that is easily portable to a browser client. Their implementation is in Javascript and is available on Github [1]. However, it still seems to be a work in progress, and although it is an extremely useful tool, users will not adopt it if it lacks an intuitive user interface. An area of further research that would be most useful to the user, then, is to create an application that is readily available to house and query the compressed version of the neural network. We found zxcvbn's interactive tool quite helpful and would suggest this as a model for future application development.

There are many reasons why having a client-side checker would be useful. The work from Melicher et. al shows that even highly compressed and simplified neural network models for password strength checking perform as well or better than other larger models, such as Markov Models and PCFGs. Another reason for having the neural network on the client is that there is no need to send data to a server. A neural network based checker that lives on the client would be able to take advantage of the large amount of data used to train the network as well as the security of not having one's password leave their device for strength checking.

VII. CONCLUDING REMARKS

We have re-implemented a password strength checker that uses a neural network to directly model an adversarial guessing attack. This password strength checker consists of two major components: the neural network and the Monte Carlo simulation. We trained an LSTM on a large number of leaked passwords to learn a probability distribution over passwords from which we can sample. Given a password as input, the neural network can also output the probability of that password occurring. Our Monte Carlo algorithm then uses the probability distribution and input password probability to estimate a guess number for the password. We gave qualitative and quantitative comparisons between this system and zxcvbn using guess number estimates and password strength scores from two sets of passwords each sampled randomly from two new datasets. We found that our model generally overestimates password strength when compared to zxcvbn, especially on long patterned passwords, as discussed in Section V.

Prior work shows that our model may be improved by incorporating natural language dictionaries, more diverse training data, and transference learning. There are many areas for future research, but the most pertinent is to create applications that compress the neural networks in probabilistic strength checkers and make them available client-side to users. With these tools, users may receive instant and accurate feedback on their password strength and increase the security of their accounts.

ACKNOWLEDGMENTS

We'd like to thank Saranga Komanduri, one of the authors of [1], for supplying us with training data, and MIT's Center for Bits and Atoms, which generously provided us with GPU time for training the neural network component of our strength checker.

References

[1] Melicher, W., Ur, Blase., Segreti, S., Komanduri, S., Bauer, L., Christin, N., and Faith-Cranor, L. Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks. In Proc. USENIX Security Symposium (2016).

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

[3] Dell'amico, Matteo, and Maurizio Filippone. "Monte Carlo Strength Evaluation." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15 (2015).

[4] Wheeler, Daniel Lowe. "zxcvbn: Low-Budget Password Strength Estimation." Proceedings of the 25th Usenix Security Symposium (2016).