A Reproduction of the Novel *PassGAN* Paper

Tanay Bapat
University of Virginia
1826 University Ave,
Charlottesville, VA 22904
tb6xr@virginia.edu

Taylor Barmak
University of Virginia
1826 University Ave,
Charlottesville, VA 22904
twb4tr@virginia.edu

Rachel Benton
University of Virginia
1826 University Ave,
Charlottesville, VA 22904
rmb6xz@virginia.edu

Nithin Vijayakumar
University of Virginia
1826 University Ave,
Charlottesville, VA 22904
nv2ba@virginia.edu

1. INTRODUCTION

In 2017, *PassGAN: A Deep Learning Approach for Password Guessing* was introduced by Hitaj, Briland, et al. The paper was revised in 2019. The paper uses generative adversarial networks (GANs) to build, test, and analyze state-of-the-art password guessing. PassGAN, the name of this approach, is a tool that uses deep learning to guess passwords without a-priori knowledge about any passwords [1]. In seeing the efficacy of the tool and the importance of password security, we have decided to reproduce the tool and its results as described in the paper.

2. BACKGROUND & RELATED WORK

2.1 Background

Password cracking is the process of generating and hashing passwords and comparing them against a database of hashes. PassGAN is a tool that can generate passwords based on an input dataset. PassGAN uses a Generative Adversarial Network, which utilizes Generator/Discriminator subnetworks that constantly evaluate the other, resulting in each becoming more adept at their task. The Generator gets trained on how to generate passwords that are similar to the input distribution, to "trick" the Discriminator, which tries to distinguish between generated and real passwords. The end tool would be a Generator Network, which can generate billions of passwords, which, after being hashed, can be compared to the database of hashes.

2.2 Related Work

Two popular rule-based password guessing tools are John The Ripper and HashCat. They use multiple types of password guessing strategies, including dictionary-based attacks and rule-based attacks. Markov models were used to generate password guesses by Narayanan. Markov models use manually defined password rules. This technique was improved on with Probabilistic Context-Free Grammars in a paper by Weir. These PCFGs demonstrated how to learn rules from password distributions. Ma and Durmuth have extended the work on PCFG. The first paper using neural networks on passwords is in 2006 by Ciaramella. Recently, Melicher et al. wrote a paper about using recurrent neural networks on passwords. This, however, was used to assess the strength of a password. This method was intended to be lightweight and provide a password strength estimator that could be used in web browsers.

3. MOTIVATION & TARGET TASK

3.1 Motivation

Password cracking is a useful technique to prove the validity and security of passwords. It encourages users not to reuse passwords and to select strong and lengthy passwords by demonstrating what would happen otherwise.

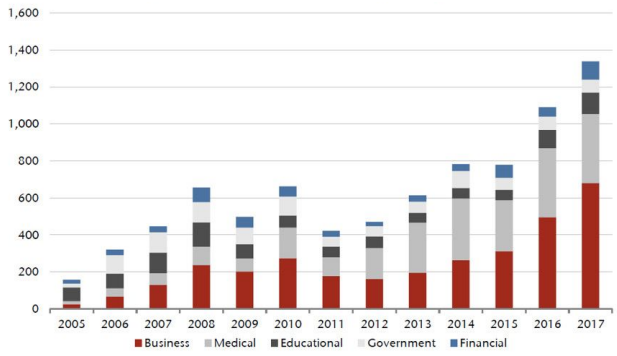
The motivation in implementing and improving this more unconventional method in password cracking is to improve on existing tools such as HashCat and John the Ripper, make the case for using stronger passwords, and experiment with an original application of GANs in a new domain.

In addition, a strong motivation is to reproduce the results seen in the paper to validate their claim. If this type of result can be achieved by simply downloading a pre-trained model, then it is critical that the latest developments are being shared so the public can know the extent of password cracking.

3.2 Target Task

The target task is to reproduce the findings of *PassGAN: A Deep Learning Approach for Password Guessing* and implement PassGAN, and attempt to find areas for improvement. Furthermore, this paper will show the importance of not using common passwords or the same password for multiple website logins. Figure 1 demonstrates the previously mentioned importance by highlighting the increase of data breaches in recent years [2].

Figure 1 - Number of Data Breaches Over Time



4. PROPOSED SOLUTION

To reproduce the paper, we plan to recreate the tool exactly according to the paper. Their Generator and Discriminator uses a series of Residual blocks composed of 1-dimensional convolutional layers. PyTorch will be used to build a Generative Adversarial Network and the resulting Generator will be used to create passwords. We plan to use the datasets in the paper, LinkedIn and RockYou, we plan to use the tool to test two of their main claims - 1. standalone performance for PassGAN and 2. in combination with other state of the art rule-based password

guessing tools, namely HashCat and John the Ripper. These are the most important results that need to be verified.

In addition, the implementation will be tested on datasets that have been leaked since the paper's initial release. Dubsmash had a leak of over 160M passwords of which roughly 22M have been cracked so far publicly [3]. We then attempt to iteratively improve upon the PassGAN paper by implementing different architectures for the GAN.

5. CONTRIBUTIONS

(1) While the original paper showed that a GAN can generate high-quality password guesses, it was using deprecated software (Python 2) and libraries (Tensorflow 1) and thus our tool contributes a more recent and updated version of PassGAN. In addition, our implementation of this tool was applied on a more recent dataset, showing the ability of PassGAN to be generalized.

(2) PassGAN is competitive with other state of the art password guessing systems, and our most recent version of this tool is well engineered and easy to build upon. By splitting the code into separate programs, our implementation of PassGAN eliminates repeated functions, thus making the code very understandable for even someone unfamiliar with the tool. In addition to a more organized layout, our implementation improves upon importing datasets by making it a simple one line edit.

(3) Using the same password or easy to guess passwords such as the website's name or a sequence of numbers can be detrimental to one's security. Our implementation of PassGAN, like other password guessing algorithms, re-emphasizes the importance of using unique passwords by showing simpler and non complex passwords can be guessed rather easily by a GAN.

6. IMPLEMENTATION

6.1 Technical Solutions

In our project to implement PassGAN, we closely followed how the researchers created the tool. They based this tool on the Improved Training of Wasserstein GAN paper [4], referred to as IWGAN/WGAN-GP. The IWGAN paper built upon the Wasserstein GAN (WGAN) paper [5], which will not be discussed in this report. In the IWGAN paper, their main contribution over the WGAN paper was increased stability by adding a penalty term when training the discriminator. In Figure 2, it is represented by the lambda term on line 7.

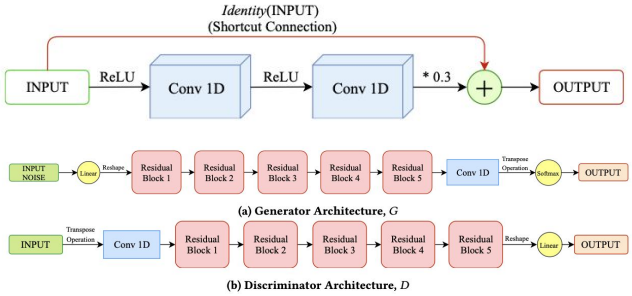
Figure 2 - Foundational Algorithms: From WGAN-GP Paper

1: **while** θ has not converged **do**
2: **for** $t = 1, ..., n_{\text{critic}}$ **do**
3: **for** $i = 1, ..., m$ **do**
4: Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
5: $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
6: $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
7: $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
8: **end for**
9: $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
10: **end for**
11: Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
12: $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
13: **end while**

We implemented the GAN in PyTorch, which provided us the capability to calculate intermediate gradients via their autograd package. This allows us to calculate the gradient penalty term easily because of PyTorch's ability to calculate gradients with respect to specific terms, without modifying the overall computation graph when applying the optimizer.

The generator and discriminator networks were constructed as the PassGAN paper discussed. This was also taken from the IWGAN paper, which featured a character level language model. This model was repeated in the PassGAN paper, which is displayed in Figure 3.

Figure 3 - Generator/Discriminator: Residual Blocks Model



The main component that made each of the generator and discriminator networks was a Residual Block. The residual block for PassGAN is composed of 2 1-dimensional convolution layers with a shortcut connection. The shortcut connection adds the input layer to 0.3 * the output from the convolutional layers. This essentially means that the network can be deeper without being affected by the traditional issues with gradient vanishing/exploding because the shortcut connection allows the identity function to easily be learned. The input to the generator is a 128 dimensional latent vector and the output is a 10 x num characters encoded. This is also the form of the input that the discriminator takes, and that network outputs a single value, [0, 1], that contextually represents if the discriminator thinks the input is real or not.

We used all the hyperparameters listed in the PassGAN paper, such as batch size, learning rate, kernel size, and others. However, the paper ran for 199,000 iterations, but we instead trained for 158,000 iterations.

Due to our need for GPU's to speed up computation, we relied on Google Cloud Platform compute instances. We were able to record checkpoints periodically after a certain number of iterations occurred, and could resume at any time.

6.2 Technical Challenges

The limitations of this project were mainly due to the scale of the model and data. Training the 158,000 iterations took over 36 hours, using a Nvidia Tesla T4 16GB GPU on Google Cloud Platform. In addition to the training taking an extended amount of time, generating the billion passwords took approximately 4 hours and 11 GB of storage. To process the generated passwords, it was also a large effort because the one billion passwords required nearly 80 GB of ram to process on one computer in Python. The researchers generated 50 billion passwords when testing, which would have scaled to more than 1 TB of data. This would have been impossible for us with our current methods of generation and analysis. We had a slight lack of domain knowledge working with

big data and could have instead sought a distributed solution and a database to more effectively generate and analyze the results.

Due to these time constraints, we were unable to train/test on multiple datasets like we had proposed to. Another result of these limitations was not being able to fine-tune the number of iterations. The paper had suggested 199,000 iterations as a hyperparameter, but due to the differences in training data, we wanted to experiment with how different checkpoints generated passwords. If the later checkpoints kept generating the same set of passwords, then it may have given us some indication of the model overfitting. This would help us make conclusions about what quality of training data is necessary to have an effective model.

6.3 Technical Novelty

The novelty of our implementation of PassGAN mainly comes from the improvement of the code for end users. Officially, there is no publicly available open-source PassGAN tool. There have been implementations in the past [6], but these have several issues. As stated in the implementation section, WGAN-GP is the original paper from which the researchers who made PassGAN referenced. WGAN-GP has available code contributed by the researchers [7]. The PassGAN open-source solution uses the code directly from the WGAN-GP repository, which is over four years old and uses Python 2 and Tensorflow 1. This is problematic, as both are not backwards compatible. Using the latest version of both Python and PyTorch will encourage users and researchers to build upon up-to-date code.

In addition, our implementation is engineered to be modular and easy to use and understand. The `main.py` file contains all tuning parameters needed for different hyperparameters, datasets, number of passwords to generate, etc. This file simply calls functions from a variety of utility files which are abstracted into different parts of the project. Data, model, and training functions are labeled accurately and clear. The same cannot be said for the other implementation of PassGAN.

6.4 Method Variations

As was mentioned in section 6.2, time limitations prevented us from testing on multiple datasets, as well as experimenting with different iterations, which in turn restricted our ability to have variations in our method compared to the original paper. The major difference was training on a different data set than what the paper had used. The effects will be discussed further in the experimental analysis section.

7. DATA SUMMARY

The dataset to which we applied our implementation of PassGAN was the Dubsmash dataset mentioned earlier in the paper. The dataset is a real-world dataset that contains 22,119,460 cracked passwords. Each line in the dataset has a hashed version of the password separated from the plaintext version by '`=:`', for example:

```
pbkdf2_sha256$12000$00Lj5R6V38Kz$+dnX5a0ILWN7vom
uw1KCtebUP3jEXVeOE1pIJgKpW8=:purple11
```

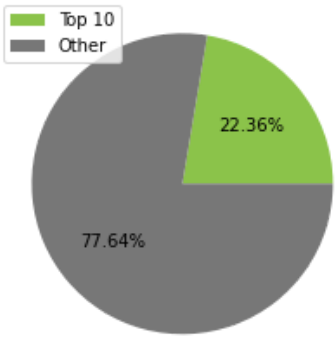
In order to process this data, we extracted the plaintext password from each row and only used the passwords that were 10 characters or less in length, which conveniently made up 92.5% of the original passwords (20,482,352 passwords). For those passwords with fewer than 10 characters, '|' characters were added to pad the lengths. For example, '`password`' would have been saved as '`password||`', '`hello`' would become '`hello|||||`', '`12345`' would become '`12345|||||`', and so on. Another notable and significant aspect of our data is that there are many identical passwords in the dataset. In fact, the ten passwords with the highest frequencies in the dataset made up 22.36% of the entire data set. Simple passwords, such as '`12345`', '`123456789`', '`dubsmash`', and '`password`' are among these most-frequent passwords. Figure 4 (below) is a table that shows the top ten passwords in the Dubsmash dataset and their respective frequencies. Figure 5 (below Figure 4) shows this distribution, with green representing the cumulative frequency of the top ten passwords within the whole Dubsmash dataset.

This dataset covers passwords with several special characters and a mix of uppercase and lowercase letters. While this only covers passwords that are less than or equal to 10 characters in length, this data reflects enough coverage of possible data cases to show the contributions of this research.

Figure 4 - Top 10 Passwords and their Frequencies (Dubsmash)

Password	Frequency	Rel. Frequency)
12345	2,497,511	12.1935%
123456	600,495	2.93178%
123456789	504,007	2.4607%
12345678	279,592	1.365%
1234567890	173,441	0.8468%
dubsmash	144,141	0.7037%
1234567	110,885	0.5414%
password	95,766	0.4676%
qwerty	95,478	0.4661%
00000	77,946	0.3806%

Figure 5 - Top 10 vs Other Passwords in Dubsmash data



8. EXPERIMENTAL RESULTS

The performance of the neural network was measured by its ability to generate passwords that it had not seen. This was done by taking the percentage of passwords in the test set that were generated by the GAN.

In the original paper, the researchers were able to match 1,350,178 (43.6%) unique passwords out of 3,094,199 passwords from the RockYou test dataset, and 10,478,322 (24.2%) unique passwords out of 43,354,871 passwords from the LinkedIn dataset. They generated 50 billion passwords trained on 199,000 iterations. In our implementation, we had trained on 19,458,235 passwords from Dubsmash and generated 1 billion passwords. On the withheld test Dubsmash data, we matched with 9.56% unique

passwords. On the entire RockYou dataset (13,020,477 unique passwords), we matched 5.16% unique passwords. Our performance was lower than expected, and we hypothesize that this mainly is from the difference in dataset.

Figure 6 - Sample Passwords Found after N Iterations

Iteration 1	o0['qJsx3P	M63VK}],hZ	{|wY*Zn9MK
Iteration 1,000	Sacdmama||	1utn1|34||	cdbdb123||
Iteration 5,000	lucexOQOQ@	12315|||||	chschos122
Iteration 10,000	Veyminop9|	morebbet||	gasmort|||
Iteration 25,000	pugharda||	almarty8ip	1234567893
Iteration 75,000	madia|||||	brace2013|	pania|||||
Iteration 158,000	gameett!||	asheelee18	12345678on

This figure shows the sample passwords generated after a certain number of training iterations. As the researchers noted, the improvements gained from additional iterations tapers off around 125,000 iterations and tapers again around 199,000 iterations. Ideally, we would have also tracked the number of unique predictions for various iterations because then we could have identified when the model was trained enough. This is not possible to the limitations explained in Section 6.2

Figure 7 - Percentage Unique Passwords

Number of Generated Passwords	PassGAN paper	Our Implementation
10^4	97.38%	76.2%
10^5	94.4%	70.6%
10^6	85.6%	59.9%
10^7	70.6%	45.4%
10^8	52.8%	29.9%
10^9	35.6%	16.3%

Figure 7 demonstrates how training on the Dubsmash dataset may have significantly altered our results. The percentage of unique passwords is much lower than the researchers, who trained the model on the RockYou data. The fewer number of unique passwords generated, generally the fewer opportunities to match with a password in the testing set.

9. EXPERIMENTAL ANALYSIS

Compared to the paper, our implementation results varied significantly. Much of this difference can be attributed to the change in dataset. The researchers in the paper trained on the Rockyou dataset, which contained around 20 million passwords. While the Dubsmash dataset we used also contained around 20 million passwords, there was a large difference in quality. RockYou had 100% of its passwords cracked. This means large, complex passwords were also in the dataset. In the Dubsmash dataset, however, only 22 million out of 163 million were cracked. We hypothesized that this means only the more simple passwords were in the Dubsmash data set, so the GAN only fit to the simple passwords. Presumably, the passwords in the Dubsmash dataset were cracked by traditional rule-based password guessing strategies, which further biased the training data.

Figure 8 - Top 10 Passwords and their Frequencies (RockYou)

Password	Frequency	Rel. Frequency
123456	290,732	0.9808%
12345	79,080	0.2668%
123456789	76,794	0.2591%
password	59,480	0.2007%
iloveyou	49,960	0.1685%
princess	33,369	0.1126%
1234567	21,728	0.0733009%
rockyou	20,918	0.0705684%
12345678	20,554	0.0693404%
abc123	16,648	0.0561632%

While the top-10 passwords are similar between the two datasets, it is clear that the variety of passwords is different between the two. Dubsmash had 22.36% of passwords in the top 10, while RockYou's top-10 passwords (Figure 8) made up just 2.26% of the overall passwords.

Figure 9 - Samples of Matched and Unmatched Passwords

Sample of Matched Password	mollyy, angles99, sahara123, melinda 28082003,
Sample of Unmatched Passwords	joey1021, 1607681, 9946805264, purple01, 07272001
Sample of Unmatched Generated Passwords	miay1333, to412515, Porthe608, Perkla98, Aohstriey

As one can see in Figure 9, even the passwords generated by PassGAN that did not match any of the passwords in the testing set still presented themselves as something a user could have created and thus could potentially match other real-life passwords that were not used in our testing set.

The technology behind PassGAN can be applied to generate pictures of human faces from a set of known pictures. It can also be used for text to image translation.

10. CONCLUSION & FUTURE WORK

10.1 Conclusion

What we have been able to find throughout the semester through our project is that using Generative Adversarial Networks (GANs) is an effective way to generate passwords from a set of known passwords. We've also found that the PassGAN tool can be generalized to datasets outside of the datasets used in the original paper. We have been able to demonstrate that the concepts are still state-of-the-art in higher, more updated levels of Python, as our version is built using the most modern tools available, thus reasserting the efficacy of the PassGAN method. While we have not been able to acquire as nuanced results (fine tuning model, adjusting algorithms, etc.) as we would have been able to given more time, we have been able to demonstrate success and ability to generalize using these concepts. These conclusions relate closely with our contributions. In addition to successfully implementing the tool from PassGAN using the most updated versions of the technical tools we've used and also demonstrating generalizability by using a new dataset of passwords (Dubsmash), we have also made a case for several better password practices. Because we were able to capture only about a tenth of unique

passwords, about almost half of all passwords, we can infer that choosing more unique, varying passwords would decrease the chance of password-related breaches.

10.2 Future Work

If there was more time we would have done testing on other datasets like was mentioned earlier, as well as done further testing within a single dataset by picking different splits other than 90/10. Another variation would have been using an entire dataset as the training set and a different dataset as the testing set to see how well one leak of passcodes could predict another site's passcodes. In addition, we think it would be interesting to do training on datasets that have greater requirements for their passcodes including capital letters, special symbols, or lengths greater than ten. Besides the different datasets we could use, further research could include changing the model architecture itself. Currently, PassGAN uses 1 dimensional convolution layers in the residual blocks. Experimenting with more or less residual blocks, or using an RNN instead of the convolutional layer. Generally RNNs are preferred for sequential data, so it could result in better performance.

Although there is room for improvement and further studies, we are pleased with our results and demonstrable success, despite the unanticipated limitations and computational constraints.

11. REFERENCES

[1] Hitaj, Briland, et al. "PassGAN: A Deep Learning Approach for Password Guessing." *Applied Cryptography and Network Security Lecture Notes in Computer Science*, 14 Feb. 2019, pp. 217–237., doi:10.1007/978-3-030-21568-2_11.

[2] "More than 96% of Consumers Are Sharing up to Six Passwords with Other People." SaferPass Blog, 14 July 2016, blog.saferpass.net/more-than-96-of-consumers-are-sharing-up-to-six-passwords-with-other-people/.

[3] s3in!c. "Leak 'Dubsmash.com'." Hashes.org - Leak 'Dubsmash.com', hashes.org/leaks.php?id=1793.

[4] Gulrajani, Ishaan, et al. "Improved Training of Wasserstein GANs." *ArXiv.org*, 25 Dec. 2017, arxiv.org/abs/1704.00028.

[5] Arjovsky, Martin, et al. "Wasserstein GAN." *ArXiv.org*, 6 Dec. 2017, arxiv.org/abs/1701.07875.

[6] https://github.com/brannondorsey/PassGAN

[7] https://github.com/igul222/improved_wgan_training

About the authors:

Tanay Bapat, Taylor Barmak, Rachel Benton, and Nithin Vijayakumar are third-year undergraduate Computer Science students at the University of Virginia enrolled in CS 4774 (Machine Learning) with Professor Yanjun Qi.