Exploring the impact of language differences on GANs for Password Cracking

Roskilde University
Computer Science Department

Author
Eugenio Maria Capuani

Supervisor
Niels Jørgensen

June 2, 2019

Computer Science Department
Roskilde University
http://www.ruc.dk

Title:
Exploring the impact of language differences on GANs for Password Cracking

Theme(s):
IT Security, Artificial Intelligence

Thesis Period:
Spring Semester 2019

Author(s):
Eugenio Maria Capunai

Supervisor(s):
Niels Jørgensen

Number of characters (inc. spaces):
91835

Number of normal pages:
38

Date of Completion:
June 2, 2019

Abstract

The goal of this thesis is to explore what impact language has on password cracking (if any) when using Generative Adversarial Networks (GANs) to crack passwords. In it we build on existing research by taking an existing GAN-based password cracker, and testing it with a dataset of leaked user passwords from Italy. By comparing the performance of the resulting model with widely-used wordlists based primarily on English-language sources, as well as state-of-the-art rule-based tools, we hope to get insights into the impact that different grammatical structures have on both the performance of the GAN model, and by extension on its performance as a password cracking tool.

In our testing we were able to crack almost 80% of the Italian passwords in our database using GANs, while other state-of-the art tools only managed 71%. Furthermore, we recovered a number of unique language-specific passwords that traditional tools would not have been able to find.

We also explored the impact of training GANs with natural language data in addition to password data, but found performance was about the same as without the extra natural language data.

We conclude that language differences do have an impact on GAN-based tools, and that these tools are expressive enough to adapt to such differences. Overall GANs can be an effective tool for attacking non-english passwords, both in addition to state-of-the art tools or by themselves.

Contents

1 Introduction

1.1 Motivation

The aim of this thesis is to evaluate the paper "PassGAN: A Deep Learning Approach for Password Guessing"[8], by testing the Deep Learning system described in the paper with a different dataset of leaked passwords from Italy[12] (referred to as the Libero dataset in this thesis). The aim of the thesis is to test whether PassGAN can be used effectively to crack passwords from a non-english source.

This dataset is composed of real user passwords belonging to the Italian email provider Libero Mail, that were leaked in 2016. We consider their use ethical as this data has been publicly available for a number of years[1].

The underlying thought behind our work is to test whether differences in grammar and language have any noticeable effect when training the system: perhaps GANs are expressive enough to account for the different provenance of the data without any significant change, or maybe some adjustments should be made such as the inclusion of Natural Language corpora in the training data.

The inclusion Natural language corpora might present its own challenges, such as the fact that some prior studies with Recurrent Neural Networks indicate that the inclusion of such data ends up creating a lot of noise rather than improving performance[13].

One might make the case that linguistic differences are not that relevant when it comes to passwords, that the use of grammatical constructs is often trumped by patterns of user behaviour in password creation that are international and well represented in rule-based password crackers. Thus, it should not matter where a neural network learns these patterns from.

On the other hand, in our early attempts to train PassGAN on the Libero dataset we observed that most of the passwords generated by PassGAN seem to attempt to mimic the sound and structure of Italian words; this leads us to speculate that the inclusion on natural language corpora might help the system to generate grammatically correct words, and thus perhaps improve performance.

In conclusion, we believe this research might contribute some insight in the role that language has in password cracking, when this task is approached using Deep Neural Networks: many papers on the subject (including [8] and [13]) ask the question of whether Deep Learning systems are expressive enough to generate novel passwords and thus yield results that are not achievable with traditional password cracking methods, and we see this research as part of this ongoing quest into exploring the capabilities and limits of Deep Learning-based password crackers.

[1] For more information on the dataset the reader can refer to section 3

We believe this research is relevant because Deep Learning-based password crackers might represent a paradigm shift in the way we think of and approach the issue of protecting users' passwords. While most of the work in this field seems to happen in an academic context right now, we believe that Deep Learning-based password crackers will present a real threat in the future, especially when in the hands of highly motivated actors with access to vast quantities of data to train and optimize such systems with (large corporations or governments come to mind).
By furthering our understanding of their capabilities and limitations, we can better protect against them and inform our security policy decisions going forward.

1.2 Problem formulation

This Thesis aims to answer the following question:

Can a Deep Learning system such as PassGAN efficiently crack passwords coming from a non-english source? If so, what is the impact of introducing natural language data when training such a system?

In order to answer this question we have broken it down into four sub-questions:

- How does PassGAN perform when cracking Italian passwords?
- Does difference in language have an impact on the passwords found by PassGAN and state-of-the-art password crackers?
- How does the inclusion of natural language data during training affect PassGAN's performance?
- Ultimately, can PassGAN be a useful tool when approaching password data from a particular language area?

2 Related Work

Both Password Cracking and Deep Learning are active areas of research developing at a rapid pace.

In section 2, we aim to give an overview of the relevant knowledge in these areas as it relates to our thesis: section 2.1 below will cover the relevant knowledge concerning passwords, while section 2.2 will cover Deep Learning.

2.1 Password Cracking

Password cracking has been around for a long time, and while technology has evolved greatly and continues to do so, the basic concepts have remained relatively similar: back in 1979, Morris and Thompson were already aware of different ways to attack passwords and defend against such attacks [16].

At the base of all password attacks is the concept of *key space*, i.e the set of all possible passwords of a certain length that use a specific character set [16, 7].

Key space can be formally defined as the set of all possible keys for some given parameters. The two parameters in question are: the Alphabet of the key (the set of all symbols or characters used in the key), and the length of the key. Key space can be theoretically infinite, but in practice it is bound by both the alphabet and the key length: in the case of an hexadecimal key of length 10, the key space can be quantified as 16^{10} possibilities; on the other hand in the case of the password `password` (written with lowercase characters only), the key space can be expressed as 26^8.

Key space is important because it forms the basis from which password cracking techniques work off: if the password is sufficiently complex, an attacker will have to do an exhaustive search of the key space in order to find the password. It follows then, that one of principles of password strength is to make the search space big enough to be impractical to search through with current hardware. This is the reason why users are commonly advised to use a variety of different characters classes in their passwords, and also to use passwords of a certain minimum length.

To ground our discussion of search space in the context of cracking, we should address how passwords are stored in the first place: as Morris and Thompson explain in their paper, the simplest approach is to store the users' passwords in a file or database as they are entered. This is a bad idea as any software bug that causes an accidental disclosure will leave the users exposed, and also because any privileged user can simply look up other users' passwords.

A better approach would be to somehow encrypt the user's password, and store the cypher text in the database: when a user logs in, the string they typed is encrypted, compared with the cypher text and access is granted if it matches. This process is

commonly achieved through a one-way cryptographic hash function, while Morris and Thompson use the DES algorithm in their paper.

Simply put, a cryptographic hash function is a mathematical function that encodes a string in a predictable way, mapping an arbitrary amount of input (a message of arbitrary length) to a string of a fixed length (a hash).

Cryptographic hash functions adhere to certain security standards (hence the adjective "cryptographic") but they are different from encryption algorithms: one of the main differences between symmetric encryption and hashing algorithms is that an hashing algorithm is a one-way function, meaning there is no mechanism to decrypt or otherwise reverse the process once the input data has been hashed.

Another difference can be found in the purpose of hashing algorithms: broadly speaking they are used to compress information about an arbitrary amount of data into a string of a certain length; their main application is to act as "fingerprints" for a given message, ensuring the integrity and/or authenticity of the data rather than its confidentiality.

We should note that DES - the algorithm used by Morris and Thompson [16] to hash passwords, is in fact a symmetric encryption algorithm: the authors used it to encrypt a known constant using the user's password as a key and stored the resulting cypher text. This in effect gave them the same results as a hashing algorithm: the massage text itself has no value, but the system can encrypt this constant with the string provided by the user and see if the resulting cypher text matches the contents of the database. Morris and Thompson chose to use DES mostly due to the fact that the algorithm ran quite slowly on computers of the period: it wouldn't be much of an issue to encrypt and check small numbers of passwords, but this slow software implementation would add a lot of time to any cracking attempt.

The reason why hashing algorithms are used with passwords is that in a scenario like the one mentioned before (where whatever string the user types is hashed and compared with the hash in the database) there is no need for decryption: the hash algorithm obscures the input, but its main role its to make sure the password the user enters at login is the same as the one that is stored in the database. Additionally, the mere possibility of decrypting a user's password might represent a security risk: if a user forgets a password we do not need to decrypt it and show it to the user, we can simply delete the password and ask him to make a new one.

Crucially, in order to accomplish this task a hashing algorithm needs to be predictable i.e always output the same string for a given input: this is a propriety an attacker can exploit as we will see shortly.

When a strong hashing algorithm is used, an attacker will theoretically have to do a key space search in order to crack the users' password: this process will take even more time, as every candidate string/password must be first hashed with the same

algorithm and then looked up in the database.

Hashing alone does not solve the problem however, because in reality an attacker would not have to resort to brute-forcing in a majority of cases.

Because the hashing algorithm needs to be predictable, the attacker can start comparing the hashes in the database and draw some conclusion: if some hashes appear many times, they probably hold the plain text of some very common passwords; if the attacker cracks those first and starts looking for similar patterns, he may crack a sizable number of the passwords contained in the database.

Practically, an attacker can exploit this predictability by building a table with precomputed hashes for all the most common passwords: this technique saves a lot of time since we do not need to encrypt every candidate password before comparing it with the database, and instead we merely look up the hashes from the table in the database.

These tables are commonly referred to as *Rainbow Tables*, and they are a simplified version of the rule-based techniques covered in section 2.1.1.

Rainbow tables can be defeated by using *password salting*: salting works by generating a random string that is then appended to the user's password before it is hashed (mainly when the user first creates the account). The authors in [16] use a 12-bit random number as their salt, but modern sources suggest the use of longer and more complex salts [17]; by generating a unique salt per each user (or even better, per each individual password a user creates), rainbow tables and correlation attacks will be rendered useless as its no longer possible to meaningfully compare hashes; one might attempt to generate tables that contain salts, but this is impractical as it would mean brute-forcing the key space of the salt. This can be done however in cases where the salt is very short, or alternatively when the system administrators have chosen one fixed salt string with which every password is processed. The latter mistake its particularly grave, as it defeats the purpose of having salts in the first place.

For a more current example of how these security practices have evolved since 1979, we can look at the National Institute of Standards and Technologies' standard SP 800-63-3. This standard is intended to provide security guidelines for information systems within the US government, and was released in June 2017.

Document SP 800-63-3B [17], provides guidelines on how user secrets (passwords and/or PINs) should be stored.
They suggest that user passwords should be between 8 and 64 characters, but advise against enforcing password policies concerning the composition of passwords; instead, they suggest that user passwords be checked against a list of leaked passwords and dictionary words before they are accepted, in order to avoid the use of weak passwords.

When it comes to hashing, they suggest the use of the PBKDF2 and Balloon algorithms; Other hashing algorithms listed as suitable for password storage include HMAC and SHA-3.

Furthermore, they recommend that all passwords be salted with at least a 32bit quantity; to our understanding the standard calls for a unique salt for each user, as opposed to a unique salt for each password. The salts should be stored alongside the hashed password of the user.

In order to foil dictionary attacks even in cases where the attacker has access to the users' salts alongside the hashes, the system should perform an additional iteration of the hashing algorithm using a different 112bits salt that should be secret and stored on separate hardware form the main database.
If we were to apply such process before hashing the users' passwords with the standard 32bit salt, this would effectively neutralize all dictionary attacks even when the attacker factors the salts into the process.

To conclude this discussion of password storage and security, we will briefly touch upon our main dataset used in the thesis: the Libero dataset.

The Libero dataset is a JSON formatted document that contains information on roughly 700 thousand users of the italian email provider Libero Mail [12].
Each user record contains various fields such as user ID, user name, the associated email address and the user's password in plain text. These plain text passwords served as the basis for all our experiments (detailed in section 4), in which we attempted to crack the Libero dataset with rule-based password crackers and PassGAN. Because the passwords were provided in plain text, we had to hash them with MD5 in order to crack them. The fact that our dataset did not contain hashed passwords presented a problem, and called into question the authenticity of the data in our possession: we will go into further detail with this and the choices we made regarding the cracking process in section 3.

Now that we have a better notion of how passwords are stored, the next section will address how passwords are commonly cracked: specifically, we will use state of the art rule-based password crackers in order to explore the cracking process.

2.1.1 Rule-Based password crackers

Unlike other password cracking tools made for brute-forcing passwords, rule-based tools rely on *mangling rules* in order to attack a greater variety of passwords:

Mangling rules are patters that define how a string should be modified, they can be given in either in a regular language or in a more complex form. Their purpose is to modify each string in a wordlist or dictionary so that one entry in the wordlist can match multiple, similar passwords.

The simplest kind of rules are *masks*: they are a subset of mangling rules whose use will be explained in section 2.1.1.1. Masks are patterns expressed in a regular language, that define what character classes are in a password and at what position: their main function is to shrink the key space of a password when executing brute-force attacks.

"Proper" mangling rules by contrast are more complex and more expressive, in line with a context-free or context-sensitive language; they will be covered is section 2.1.1.3. Common examples of rules might be a rule that switches the case of the first letter of the string (mainly in order to capitalize passwords, a common strategy users employ to comply with password composition requirements), or a rule that appends one or two digits to the end of a string.

Two common rule-based password cracking tools are John The Ripper and HashCat[9, 5]: these tools use rules to exploit common patterns in user behaviour in order to optimize the cracking process. Both tools have a variety of techniques an attacker can employ, and we will briefly exemplify their capabilities using HashCat as an example [6]:

In both tools there are three categories of attacks that can be carried out: brute-force attacks, dictionaries attacks and rule-based attacks: these can be combined and tweaked in various ways depending on the desired result, and there is a degree of overlap between each.

- Brute-force attacks try every possible combination of characters within a defined pattern, and thus closely relate to key space search.
- Dictionary attacks use a dictionary of words or passwords as a source for password candidates, that are hashed and checked against the target to see if any of them match
- Rule-based attacks are an evolution of Dictionary attacks, in which the dictionary's effectiveness is boosted by mangling rules that change the words in ways that reflect common patterns in user-generated passwords.

HashCat distinguishes between brute-force attacks and wordlist-based attacks with two separate modes (a mode is a setting that defines the kind of attack that HashCat should carry out), but there is no such distinction between wordlist-based attacks and rule-based attacks: instead rules are an optional parameter one can use in wordlist attack mode to greatly increase the effectiveness of the wordlist. The following three sections will cover HashCat's attack strategies using the different modes present in the software.

2.1.1.1 Brute-force attacks

The simplest of these modes is *Mask attack* mode, used to carry out brute-force attacks as we mentioned previously: a mask attack is essentially an optimized version of key space search, meant to attack simpler passwords while shrinking the key space. Instead of searching the total space for a password of length x, we define a simple regular pattern expressing the what character classes are there and at which position, saving us a substantial amount of processing time. For example if we have a password like `Benjamin86` or `Iloveyou02`, we can capture both examples by defining our password mask to be a ten-character string with eight lowercase or uppercase letters and two numbers at the end.

In a classic brute-force attack we would deal with a search space of 62^{10} (or roughly 8×10^{17} combinations), but thanks to the above-mentioned mask we can reduce our search space to around 4×10^{13} possibilities if we assume that the first character in the string is the only one that can be uppercase. Furthermore we can use the `-increment` option in HashCat to apply this mask recursively, to all strings up to ten characters: this allows us to match shorter passwords that follow the same pattern.

This method is rather simplistic and not very flexible, but exemplifies some of the ways in which attackers can optimize key space attacks.

2.1.1.2 Dictionary attacks

The main mode of operation of HashCat is *straight* mode, that performs a dictionary attack: in this mode, the program is fed a wordlist/dictionary, and tries each entry the wordlist as a password candidate. Because of its simplicity, such a dictionary attack works best with a wordlist composed of leaked passwords; the aim is to target very common passwords and users that re-use passwords, but the effectiveness of such an attack can increase significantly depending on what wordlist is used.

Dictionary attacks can be further enhanced by combining them in various ways: one approach would be to use two wordlists and append/prepend each entry in the second one to each entry in the first.
The second wordlist might be a natural language dictionary or simply anther wordlist of plain-text leaked passwords. This is called *Combinator attack* mode in HashCat.

We might also want to use the output of a mask attack as our second wordlist: if we use the patterns described in the Mask Attack mode to generate strings or number sequences we combine with a wordlist, we will obtain a more targeted and effective version of the Mask Attack method.

For example if we know that a good deal of the passwords we want to extract are strings with numbers appended to the end, we might run a mask that generates

combinations of 0 to 4 digits and then combine the output of that with our wordlist. This is called *Hybrid attack* mode in HashCat.

2.1.1.3 Rule-based attacks

Finally we come to rule-based attacks. In short they are an extension of all the methods described above. As we said previously rule-based attacks do not have a dedicated mode, but instead they are often used in *straight* mode alongside a wordlist in order to boost the number of passwords found by that wordlist.

Rules are flexible and allow for more thorough definition of the patterns that may appear in a password, going beyond the capabilities of the regular language used with Masks. Patterns can be created independently of the size and characteristics of the passwords, and they are not limited to a fixed pattern; there are also flow control statements and options to apply rules only in certain conditions. There are also options to save password candidates to memory enabling more advanced processing: saved strings can be appended to each password candidate matching certain criteria, reversed and so on... These more advanced options allow an attacker to emulate the operation of both Combinator and Hybrid attacks using just mangling rules.

Rules are applied once to each entry in a wordlist in a similar way to a combinator attack, but multiple rules can also be be applied sequentially to a memorized password or string.

Rules provide a more efficient way to tackle password cracking since their greater flexibility means that an attacker need not know as much about their target or about the nature of the passwords he is trying to crack.

In order to get a better idea of how rules are used we can look at the *best64* rules, which contain around 70 commonly used mangling rules. The two examples below are taken from the `best64.rule` file, a version of the best64 rules that is commonly distributed with HashCat:

```
## nothing, reverse, case... base stuff
:
r
u
T0
```

The rules above are rather simple, they simply reverse a string, toggle each character's case or toggle the case of the first character: as we can see these rules already reflect some common patters used by users, such as using common words spelled backwards or toggling a word's case as a way to comply with password composition policies.

Further down in the same file we can find some more complex rules:

```
## leetify
so0
si1
se3
```

The substitution rules above are designed to defeat a common practice in user-generated passwords, leet speak: this is the practice of changing letters in passwords with numbers that look similar in an attempt to fit within character class requirements in passwords.

2.1.1.4 Using HashCat

In this section we will give an example of how rule-based password crackers are used in practice, using our experience with HashCat and the Libero dataset as reference. To see how we used HashCat and the Libero passwords in our thesis, refer to section 4.

When running a dictionary attack with HashCat (and other rule-based password crackers), an attacker needs three main pieces of data: a target file containing hashed passwords to be cracked, a wordlist, and optionally a set of rules. Additionally, one needs to know the type of hash the passwords are encoded with.

Let us give an example of a HashCat command we used in our thesis to perform a dictionary attack using rules, and walk through the various parameters (Note that the red arrows are simply a visual aid to indicate line wrapping, to make the command fit on the page):

```
hashcat -a 0 -m 0 test_10c.hash /usr/share/hashcat/wordlists/
    ↪ rockyou.txt -r /usr/share/hashcat/rules/best64.rule
```

Starting from the left:

- The first parameter `-a` indicates the attack mode: the value 0 equals to straight mode in this case for a dictionary attack.
- The second parameter `-m` indicates the hash type, hash type number 0 is MD5. As we mentioned previously (towards the end of section 2.1), we chose to hash the plain text passwords in the Libero dataset with MD5 for the purposes of testing.
- The third parameter is the target, the set of passwords to be cracked: here we used `test_10c.hash`, which is a sub-set of the Libero dataset (further details in section 4.3).
- The fourth parameter is the wordlist to use: here we used `RockYou`, a pop-

ular wordlist composed of more than 14 million passwords from various data breaches. It often comes packaged with HashCat.

- The fifth parameter `-r` tells HashCat that we want to use rules.
- The sixth parameter is the ruleset that we want to use. In this particular instance we have used `best64.rule`, the same ruleset we have taken the example rules from.

Running this command will print the cracked passwords to standard output as they are found, in the format `hash:plaintext`. Alternatively it is also possible to save the results to a file with the `-o` option.

We should also note that in this example we did not show how to deal with salted passwords: as we explain in section 3, the Libero dataset did not come with any salts, so we do not deal with them in this paper.

2.2 Deep Learning

Deep Learning is a class of machine learning systems whose goal is to extract relevant features from a distribution of data. On an abstract level, Deep Learning systems take inspiration the structure of the human brain, with multi-layer structures of nodes (Neurons): each layer might be thought of as a section of a brain recognizing a very specific element, and when all layers work in unison they can recognize and act upon high level features and categories.

Deep Learning Systems are particularly useful because of their ability to extract high level features from data, especially features that are hard to define algorithmically by humans.

Because of the number of variables involved, it is hard to predict what results might derive from a change in the initial condition; as a consequence of this uncertainty, the process of working with Deep Learning systems is one of incremental change and experimentation.

Figure 1 shows the typical structure of a Deep Neural Network:

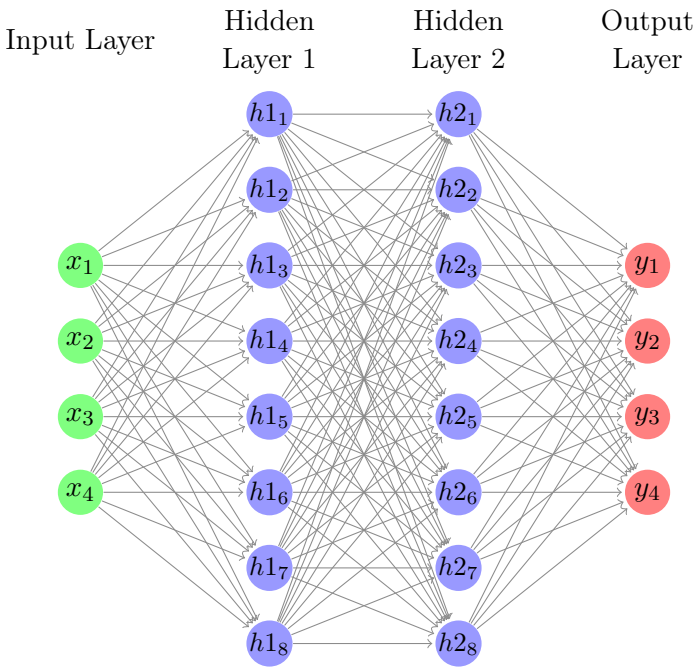


Figure 1: An example of a Deep Neural Network

As we can see in the example above, a neural network is composed of an Input Layer, a number of Hidden layers and an Output layer. Speaking in general terms, the input layer receives a chunk of data to be processed, the hidden layers operate on the input sequentially, and the output layer is where the system expresses a judgement on the

data.

The training process for a neural network is composed of two distinct phases: the *Feed Forward* phase and the *Back-Propagation* phase: in the feed forward phase the network processes the current data and comes to a result, then the networks computes the Error (or how much the result it had come to deviated from the expected result).

The network then takes the error value and propagates it backwards by adjusting the weights of the connections between each neuron in the different layers, in an effort to reduce the error. This is the back-propagation phase. Once both phases are completed, the network has completed an iteration. The next batch of data is then introduced and the process begins anew.

Note that training data is not fed to the network one piece at a time, but instead is introduced in batches. The input data is divided into batches of varying size (64 or 128 samples per batch are common values), and each iteration a new batch of data is loaded into the network; training usually ends once the system runs out of input data.

In the coming sections we will cover two types of Deep Neural Networks that are relevant in this paper: *Recurrent Neural Networks* (RNNs) and *Generative Adversarial Networks* (GANs). This following section will cover Recurrent Neural Networks, while section 2.2.2 will cover Generative Adversarial networks: we will explain the structure and operation of GANs in that section, then talk more about PassGAN specifically in section 2.2.3.

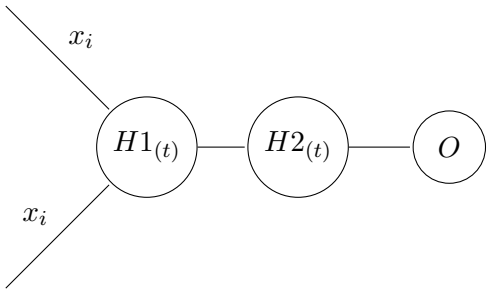
2.2.1 Recurrent Neural Networks

Recurrent Neural Networks are the simpler of the two network architectures that we will cover: they are a kind of neural network specialized in working with data that has a temporal component. For instance if the network needs to predict the next letter or word in a sentence, it needs to know what came before. Another example might be a neural network that scales or edits videos, where the action to take on any given frame might be dependent on the frames that came before.

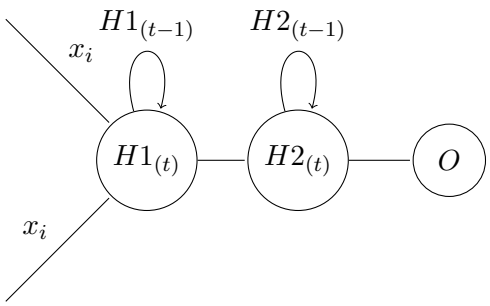
We do not use Recurrent Neural Networks directly in our thesis, but they are used heavily in Melicher et al. [13] and we cover them here as an alternative approach to Deep Learning-based password crackers; furthermore they are also relevant in the context of PassGAN, as Hitaj. et al. [8] compare the performance of PassGAN to the RNN-based system in Melicher et al.

The precise method the network uses to keep track of temporal elements is rather complex, but can be briefly summarized by saying that each neuron in the network holds a state: in each iteration a neuron's state from the previous iteration is fed as input to itself in the current iteration along side the current data to be processed. This creates a feedback loop, whereby at any given point the calculation performed

by each neuron are influenced by previous events.



Iteration number 1 $t = 1$



Iteration number 2 $t = 2$

Figure 2: A simplified representation of neurons in a Recurrent Neural Network

Figure 2 illustrates the working of a RNN neuron: taking picture 1 as a starting point we can see the feedback loop inside an RNN neuron at work. The first picture shows the state of three neurons in the network on iteration 1: during the feed forward phase the neuron in the first hidden layer receives input x from the input layer, processes it and passes it on to the consecutive layers; at the end of the feed forward phase, each neuron in the hidden layers saves its working values, and those values form the neuron's state.

In the next iteration (shown in the second picture) the process repeats, but this time the neuron in H1 receives not only input from x, but also its previous state as input; it processes the combined values and passes it on to the neuron in H2, which undergoes the same procedure.

Through this mechanism, the network can better retain knowledge of features from past samples of data, and yield better results. However this also highlights a potential problem: because each node is only fed its previous state with each successive iteration, as the training process continues the network will develop a bias towards

more recent samples of data. The cyclical nature of the feed-forward means that at any given iteration the network will still remember the initial stages of training, but the influence of those early iterations will wane with time as the network adapts and changes when presented with new data. This phenomenon is usually called 'Gradient Decay'
For some particular tasks such as generating natural language, gradient decay can significantly influence the final results.

2.2.2 Generative Adversarial Networks

Generative Adversarial Networks are a more complex architecture, composed of two discrete neural networks that are in competition with each other.

The system works by employing a generative network (G) and a discriminator network (D), and figure 3 shows its general structure:

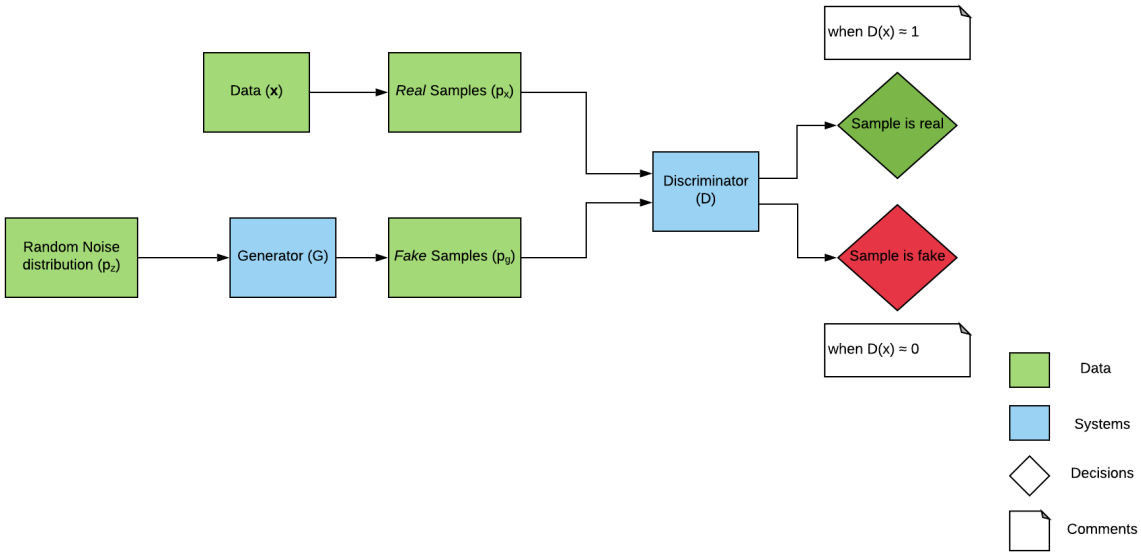


Figure 3: An illustration of the general structure of a Generative Adversarial Network

As we can see in figure 3, G generates samples from a random noise distribution p_z. We call the resulting data distribution p_g; D is fed both data from the training set (x) and the output from G (p_g), and it's task is to figure out whether any given sample comes from the data (it belongs to p_x) or has been generated from G (it belongs to p_g). In layman terms, we might say that it is D's job to tell "real" samples from "fake" ones.

Goodfellow et al.[3] explain the interplay between the two networks as a min-max game: the goal of D is to maximize the likelihood of guessing correctly, while the

goal of G is to minimize that same likelihood (or to put it another way, G wants D to make more mistakes): the reason for this is that by minimizing the likelihood of D guessing correctly, we are implicitly telling G to generate samples as close as possible to the input data.

Simply put, the two networks are trying to out-smart each other: D tries to spot the fake samples from p_g, while G tries to improve more and more to make D's job harder.

The output of D is a single value between 0 and 1 expressing the network's confidence on whether a sample came from x: values of $D(x)$ closer to 1 tend towards assigning a sample to the input data distribution. Ideally, this race/competition between the two networks comes to an end when the output $D(x)$ stabilizes on $D(x) = 0.5$, meaning that D is no longer able to tell the two distributions apart.

This also tells us that if we had infinite resources (both in data samples and computing resources), we could train until $D(x) = 0.5$ and $p_x = p_g$, giving us an output of G that is indistinguishable from the input data.

One important thing to note is that we do not show G any of the input data at any point: all that G has access to is a random noise distribution and the error value for D; the Generator has to slowly shape that noise into data resembling the Discriminator's input, by using Gradient Descent.

The process described above can be formalized in the following general equation:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \quad (1)$$

Since the network relies on this tight feedback loop between D and G to operate, one needs to pay close attention to how quickly each individual network learns. The idea, as the authors in [3] point out, is to have a certain number of iterations of D before we go ahead and adjust the Error for G. Conceptually, we might think of this as giving D enough time to tell the real data from the fake data reliably (or try), before iterating G to counter D's current strategy.

The original paper indicates this ratio with k, and notes that increasing it grows the required computation significantly. In their experiments the authors used a k value of 1 for performance reasons, while PassGAN defaults to a k value of 10[8].

2.2.3 PassGAN

PassGAN is a generative adversarial network built to generate password candidates for the purposes of password cracking. In it, G and D compete in an attempt to generate passwords that closely mimic the real passwords fed to the network as input data.

In this section, we will go over some of the important hyper-parameters used with PassGAN and some other concepts that will become relevant is section 4, where we put PassGAN to the test.

In order to train PassGAN, we use the passwords from the Libero dataset mentioned in section 2.1. As is common with deep neural network training, we split our data into two subsets of 80% and 20%: 80% of the passwords are used for Training, while 20% are used for Testing.

The Training phase is what we covered in section 2.2.2, but we did not talk about Testing in this chapter: generally speaking testing is the practice of presenting a newly-trained deep learning system with new data it has not seen before, in order to test whether the model performs as expected outside of the limited scope of the training data.

For some simpler machine learning systems like image classifiers (a deep learning system trained to recognized hand-written digits is a very common example), all image data is labelled with what they are supposed to be (*e.g* the label might say what digit a particular image is supposed to be): if we split our data 80%-20% like we mentioned before, at the end of training we can present our network with the 20% of data we put aside and use the labels to quantify whether it can guess as accurately as we expect.

As an example of why Testing is needed, one of the risks associated with deep learning systems is *over-fitting*: over-fitting is a phenomenon wherein the network is "over trained" so to speak, it performs amazingly on the training data but is not flexible enough to maintain that performance with real data outside the narrow scope of training. Intuitively, we might think of it as the network being hyper-focused on the minute characteristics of the training data: it works very well in that specific context but it does not scale well, the features that the network picks up on might not be abstract or general enough to apply in a general case.

We are discussing the testing process in some depth because there is an important difference between PassGAN and image classifiers: the testing process described above simply does not work with PassGAN. This is because PassGAN does not simply classify existing data, it creates new and unique data with potentially unseen characteristics: because we do not know what kind of string PassGAN will generate in advance, we cannot write labels for them. Even if we were able to do so, we think that the nature of GANs is at odds with this approach since the interplay between G and D provides the same sort of feedback as labels do on image classifiers.

Because of the reasons mentioned above, PassGAN has no built-in method for testing: instead, we have to rely on external methods to test the model's performance. In this paper we want to test cracking performance, so we use the password candidates

produced by PassGAN in rule-based password crackers: this process will be covered in more detail in section 4, for now we will say that we used 20% of our password data as the target of our cracking attempts with PassGAN and HashCat.
To clarify we followed the same testing methodology as Hitaj et al. [8], just using different datasets for training and testing.

Another important concept to introduce is that of a *Model*: we have used this term already several times, but to clarify we shall introduce it properly here.
A model is what results at the end of training of a neural network: in essence it is a snap-shot of all the weights and parameters present in the network, that taken together tell the network what to do with a given input. Once we load a model into a neural network we can use it to process new data, and we tests the model with our testing data to verify that it is as accurate new data as it was with the training data.

PassGAN's approach to models is rather common, and it works as follows: by default PassGAN is programmed to take a model snap-shot every 5000 iterations (called a checkpoint in the software), potentially allowing us to use a partially trained model if we so desire; if we want to use the final model, we simply choose the latest snap-shot available.
Models are then used when generating password candidates through sampling, where we tell PassGAN what model we would like to use for generating our passwords and how many we want. Generally we will want to use the latest model available to get the best possible results, but because we have access to snap-shots from various points in the training process we may choose to use an older, partial model in order to observe how the kinds of passwords PassGAN generates change over the course of training.

The sampling process will be explained better in section 4.2. As a reference, in all our tests we used the latest snap-shot/model available and we generated either 1.000.000 password candidates or 14.000.000 password candidates. Note that the number of models available changes depending on the amount of data and the length of training: using less data for training will result in fewer iterations and thus less snap-shots available, and vice versa.

Shifting our discussion to the hyper-parameters used with PassGAN, they are values we can change to influence how the training is done. All of the hyper-parameters present in Hitaj et al. [8] were kept the same in this paper. We will briefly touch upon some of them, explaining what they do and mentioning how they influenced our testing of PassGAN.

There are three hyper-parameters that are relevant to this discussion: the first is *sequence length*, a number that defines the maximum length of password candidates generated by PassGAN; this value is set to 10.

Sequence length is important because it will influence what kind of passwords we use for testing, for the explanation of how sequence length comes into play the reader may look at section 4.3.

Another important hyper-parameter is the number of total iterations, which defines for how long PassGAN should train: this number is set to 200000, but PassGAN never reaches the last iteration during our tests: Hitaj. et al. arrived at this value after several experiments as the ideal length of training for their case, however in our paper we have a considerably smaller training dataset of around 530,000 passwords (80% of the Libero set), so we have noticed that our training stops around iteration number 190000 or 195000 when PassGAN runs out of input data.

In hindsight we should have tried to tweak the iteration count to optimize for our particular amount of data: we don't really know if this had a negative impact on the resulting model, the extra iterations might have lead to over fitting, but we would not be able to tell as we did not have another model trained with a lower number of iterations as a term of comparison.
This might be seen as one of the downsides of using a generative system like a GAN: testing requires a more elaborate setup and its harder to spot problems when compared to simpler systems with built-in testing methods.

The third relevant hyper-parameter is k, one that we have already discussed at the end of the previous section. As a reminder, k is a value that determines how many iterations of D happen for every iteration of G. In effect, it controls how quick the feedback loop between the two systems is and how fast PassGAN learns and adapts. We have not modified k in our testing, but we mention it again here because we think it is key in understanding how PassGAN works.

3 Issues with the authenticity of the Libero dataset

In the next section we will cover our experiments with PassGAN and the Libero dataset. Before that however we need to address some fundamental issues with the dataset: firstly, our copy of the Libero database only contained plain texts passwords, and secondly we were unable to verify the exact provenance of this data. A direct link to the copy of the Libero leak used in this paper can be found in our reference list [15].

As mentioned in section 2.1, the Libero set is a JSON formatted file containing various pieces of information for each of the roughly 700.000 users within: common fields for each user include email address, user-name and internal User ID, but some users in the document also have a real name attached and other kinds of personal information. As for the passwords, each user record contains both a plain text password and an MD5 hash. However something is wrong with the hashes: upon closer inspection we find that the MD5 hash is the same for each user, and that MD5 hash encodes the word "boomerang".

We have no idea as to why the uploader or the original cracker would choose to do something like that, and no clear evidence towards any particular reason. The following is purely our speculation, but as an example the attacker might decide to throw away the original hash and only keep the plain text of the password in order to keep the information for each user organized and more easily exploitable: since they exfiltrated not just passwords but also personal information like names and addresses for some of the users, one might imagine that it would be convenient to organize all the information for each user in a single JSON object and perhaps leave a placeholder hash in place of the original password hash.

This alteration to what we might normally expect to find - a file containing both plain texts and hashes of those plain text passwords - made us question the authenticity of the data in our possession: while we were unable to find definitive proof, the number of accounts present in our file is roughly consistent with the number reported in [12]; further more, the hash of the archive matches that of a VirusTotal report, that shows the archive first appeared on VirusTotal in December 2016 (a couple of months after news broke about the Libero Mail security breach)[20]. Italian news articles reporting on the incident came out around September 7th 2016, when Libero Mail sent out an email to their customers informing them of the breach [10, 4, 11].

Looking at the database file itself, its *mtime* (the time-stamp of when the file was last modified) is dated to September 25th, placing its origin somewhere closer to the date of the incident.

Unfortunately neither the news articles nor Libero Mail themselves seem to give a clear indication of when exactly the incident occurred, so its hard to judge the exact timeline of events; upon reading the email that Libero Mail sent to customers on September 7th (available in [11] in Italian), the language used suggests that Libero may have been aware of the breach for a considerable amount of time before breaking the news to customers.

The elements above leads us to believe that the data we have is likely to be from the original Libero leak, though ultimately its very hard to know for certain: these accounts and passwords have been re-used and included in a variety of other sources ever since the incident, and if the original leak came from the Tor network like the Virus Total report tags seem to suggest, it might be exceedingly hard to track down the original archive and/or its creator.

The lack of hashed passwords in our copy of the Libero leak did not interfere with training of the PassGAN system, as the input data to PassGAN consisted of plain text passwords, but it did pose a challenge for testing: in order to test PassGAN we needed to crack the Libero leak passwords using HashCat, and thus we needed to turn the plain text passwords back into hashed passwords. In order to do that we hashed the plain text passwords with MD5 and used the resulting file for testing.
Our choice of MD5 was somewhat arbitrary, as there is no clear indication of how the passwords might have been stored in the Libero Mail database: we chose MD5 because the filename of the database file hinted that that might have been the hashing algorithm used originally, and also because the weakness of MD5 would make the cracking process much faster compared to using SHA-1 or SHA-256.

The JSON file contained no explicit indication of of whether salts had beeen used originally, so we decided to not try to salt the password either. However for all the reasons discussed above, its possible that Libero Mail used a more secure storage method with different hashing algorithms and salts, we simply do not have enough information to express a judgement either way. As a final note, the fact that the user accounts are stored in JSON objects does not necessarily indicate that that was the original data structure used: the Libero Mail database might have used a more traditional database for all we know, and the attacker might have simply organized the data he obtained as a JSON file; the inclusion of UIDs however might also be an indication that JSON was indeed the format used in the original database.

We are aware that this lack of information and authenticity hurts the applicability of this paper. Because of these compromises we cannot claim that our results are 100% applicable in the real world, and our argument would have been more robust if we had a dataset that more closely matched the original way in which passwords were stored.

In order to prepare the data from the Libero leak for both training of PassGAN

and subsequent testing, we used a combination of UNIX shell commands, displayed below.
The red arrows in the code listing do not belong to the actual code, but are instead visual markings to indicate line wrapping in order to make the sometimes long commands fit on the page.

```
1  #Extract passwords from the databse file
2  cat libero.it-lebero-poinx21.pxusers-plaintext-md5-2016-09-json
   ↪ -900k-users-extremely-private.txt|grep clearPassword |cut -d
   ↪  ":" -f 2 | awk '{gsub␣("\"","");gsub(",","");print␣$1}' >
   ↪ passwords.txt
3
4  #Split the entire password dataset into 80%/20%
5  head -n 534171 passwords.txt|shuf > train.txt
6  tail -n 133542 passwords.txt|shuf > test.txt
7
8  #Extract passwords that are 10 characters or less (needed for
   ↪ testing)
9  grep -x '.\{1,10\}' test.txt > test_10c.txt
10
11 #Generate MD5 hashes of the testing set for Hascat
12 for i in $(cat test_10c.txt); do echo -n "$i"| md5sum | tr -d "␣-"
   ↪  >> test_10c.hash; done
13
14 #Generalized parameters for PassGAN
15 python2 train.py --output-dir output --training-data $DATASET
16
17 python2 sample.py --input-dir output --checkpoint output/
   ↪ checkpoints/checkpoint_95000.ckpt --output $WORDLIST -n
   ↪ 14344392
18
19 #Generalized parameters for Hashcat
20 hashcat -a 0 -m 0 --potfile-disable test_10c.hash $WORDLIST -r
   ↪ $RULESET -o out.txt
```

Note that some parameters have been omitted from the last three commands: some parameters for HashCat and PassGAN like the dataset fed to PassGAN for training change depending on the test being run, so we have replaced them here with generic all-uppercase variable names such as `$DATASET`.

4 Testing and evaluation

In this next section, we are going to evaluate the performance of the PassGAN system on the Libero set, as well as give a brief overview of our methodology and the experimental setup.

4.1 Experimental setup

Both training of the GAN and testing were carried out on a Linux machine running Debian stretch, equipped with a GTX 1070 graphics card, an i5-2500 CPU and 8 GB of system memory. With this card, training of PassGAN took roughly twelve hours. PassGAN was run on Python 2.7, running Tensorflow 1.12.0.

For testing we used the latest stable version of HashCat available at the time (version 5.1.0).

Figure 4 below illustrates our overall process, which is divided in two Phases:

- A Training Phase, in which we train PassGAN on a given dataset and sample password candidates from the trained model.
- A Testing Phase, in which we use HashCat to try and crack a sub-set of the passwords in the Libero dataset.

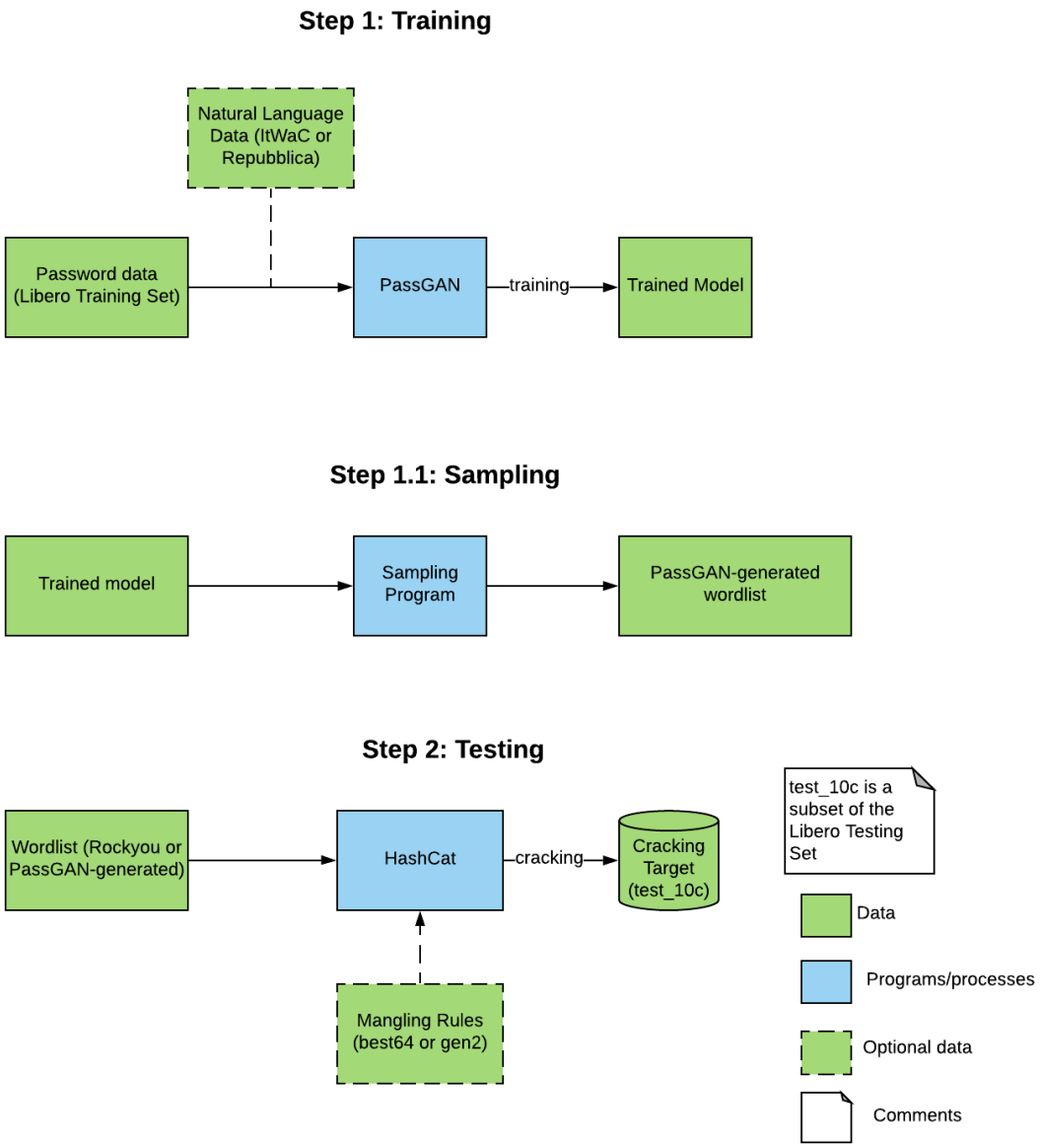


Figure 4: A diagram illustrating our general workflow for evaluating PassGAN, including Training and Testing.

4.2 Training the PassGAN system

As we explained back in section 2.2.3, in order to train PassGAN we took the plain text passwords from the libero leak (667,714 passwords) and split them into two groups. 80% of the passwords went into our Training Set, while the remaining 20%

went in the Testing set: the Training set is used to train PassGAN as can be seen in Step 1 of Figure 4, while the passwords in the Testing set are hashed with MD5 and serve as a cracking target for HashCat as seen in Step 2 of figure 4. We will go into further detail with the Testing set in section 4.3, as the brief explanation provided here is not entirely accurate.

PassGAN can additionally be trained on Natural Language data, shown as a dotted box in figure 4; elements shown within a dotted box are considered optional, they can be included or not. Natural Language data will be further discussed in section 4.4.

Once PassGAN is done training with a given dataset, it produces a trained model that can be used to generate candidate passwords via sampling (Step 1.1 in figure 4).
We covered the concept of models back in section 2.2.3, but as a reminder a model is a snap-shot of the internal state of a network, containing all the weights and parameters necessary to process new input (generate password candidates in the case of PassGAN).

We can use the trained model produced at the end of training to sample password candidates generating from PassGAN: The result of the sampling process will be a list of candidate passwords that make up the PassGAN wordlist used with HashCat when testing. We can choose to generate more or less candidate passwords when sampling: for instance the tests listed in table 1 use a wordlist of 1 million password candidates, while tables 2 and 4 use a wordlist of roughly 14 million password candidates.

Below is a small excerpt of password candidates generated by PassGAN, taken from the 1 million wordlist used in table 1:

```
22052906    mariamuna   iaup63     poldetta53
001178      04051987    topolapa    aenara
rotsanera   princone    fadyreda21  giugki
dimonepa72  12orlomon   deb57828    belnabola
marcanalla  ACADCNAN   leegenniki  vicaletto
```

What we found particularly interesting was that while the samples showed the typical patterns exhibited by user-generated passwords (numbers appended or pre-pended to words, so-called leet speak etc..), a majority of them also seemed to mimic the sound of Italian words and phrases: most of these words were meaningless, but nonetheless we speculated that the GAN might be trying to "learn" Italian as a side-effect of generating samples close to the training data distribution. As an example the string `vicaletto` resemble the italian word "vicoletto" meaning "a narrow street", while the string `princone` sounds vaguely like the italian word for pickaxe, "piccone".

4.3 Testing the PassGAN system

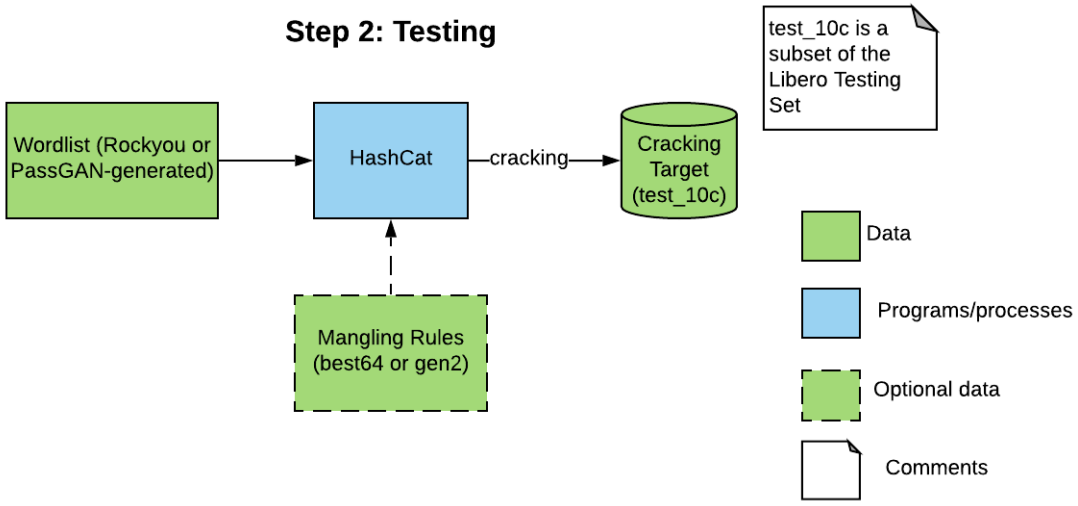


Figure 5: A zoomed-in version on the previous diagram, showing only the Testing process

The testing process is depicted in figure 5: In it we use HashCat to crack a sub-set of the Libero leak passwords (the cracking target), using a wordlist and optionally a set of mangling rules as input.

The cracking target is a sub-set of the Libero Testing set, - the set of 20% of the libero leak passwords that we set aside from testing as mentioned in section 2.2.3 and at the beginning of the previous section.

Our cracking target is not the whole Testing set, but a sub-set of it composed of passwords with a length of 10 characters or less: this was done because Hitaj. et al. [8] also limit their testing with HashCat in this same way.
We decided to do it too in order to more closely follow their methodology. We speculate this limitation exists to optimize the passwords in the cracking target to PassGAN. As we mentioned back in section 2.2.3, PassGAN sets the *Sequence Length* hyper-parameter to 10 during training: this parameter sets the maximum length of password candidates generated by PassGAN during sampling, so testing against a sub-set of 10-character passwords is a way to tailor the cracking target to PassGAn's output slightly.

We do not fully understand why Hitaj. et al. made this choice, but nonetheless we adopted it ourselves in an effort to more closely mimic their evaluation process.

We believe this sub-set of ten character passwords still represents the whole Testing set fairly, as not many passwords were removed: the vast majority of passwords in the Testing set are 10 characters or less, 111,994 or 83.86% to be precise. From here on, we will refer to this sub-set of 10 character passwords as *test_10c* (the actual filename) or more simply as "the cracking target"

Touching upon the wordlist and rules used, they are also depicted in figure 4: for the former we use either the *RockYou* wordlist or a wordlist generated by PassGAN, while for the latter we use either the *best64* or *generated2* rule sets. Both the RockYou wordlist and the two rule sets mentioned above were chosen because they were used in Hitaj et al. [8], but also because they are widely used and understood to be among the more efficient wordlists available to an attacker. We believe that they demonstrate rather well what rule-based password crackers can do without further language-specific optimizations such as those that PassGAN provides.

The RockYou wordlist contains more than 14 million passwords, all of which come from various data breaches that have been compiled into one resource; it contains passwords of varying length and complexity, and as such should provide a good performance baseline for rule-based crackers. The best64 ruleset contains around 70 mangling rules, while the genrated2 ruleset contains more than 65,000.

As a final note, HashCat's default behaviour is to cache passwords found during an attack in what the HashCat manual calls a *potfile*, so that if an attacker chooses to approach the target using a different wordlist or a different set of rules, no time is wasted cracking passwords that were already found previously; because using different rule sets can greatly affect the efficiency of cracking, this form of caching allows an attacker to crack more passwords by combining different approaches. In order to test the performance of each wordlist and rule combination separately, we disabled caching of cracked passwords between cracking attempts: this was done with the `-potfile-disable` option in HashCat, and it allowed us to evaluate the performance of each different combination of wordlists and rules singularly. However disabling the potfile also leads to a lower overall number of passwords found. Tables 1, and 2 show tests done with caching disabled, while the figures obtainable with caching will be mentioned in section 4.3.1.

The tables below will illustrate the results of our tests attacking the cracking target with different combination of wordlists and rules. To clarify, we decided early on to use mangling rules also when testing password samples from PassGAN, since they greatly increase the number of passwords found compared to just using a wordlist. We mention this because Hitaj. et al. seem to not have used rules in their evaluation of PassGAN, but instead have chosen to generate a huge number of password candidates and use PassGAN alone to match passwords in the cracking target. We have also briefly tried this approach, our experience with it will be detailed in section 4.3.2.

Wordlist/Ruleset	**-**	**Best64**	**Generated2**
Rockyou	19,215 (23.78%)	30,170 (37.33%)	59,134 (73.18%)
PassGAN	4,637 (5.74%)	11,053 (13.68%)	34,896 (43.18%)

Table 1: A comparison of the number of passwords found in the cracking target by RockYou and PassGAN, using a 1 million wordlist for PassGAN

Table 1 shows that PassGAN performs rather poorly in comparison with the RockYou wordlist: we attributed this shortcoming to two main factors: firstly the size of the wordlist generated by PassGAN, which is 1 million entries as opposed to the 14+ million entries of RockYou, and secondly the quality of the PassGAN wordlist. Both of these factors play into each other, because while mangling rules provide great a advantages in terms of password found, ultimately they just modify the entries in the wordlist. Thus we might say that the whole system is limited by the capabilities of the wordlist that is used.

In order to test this hypothesis we used our existing PassGAN model to generate a bigger wordlist with as many entries as as RockYou, and performed a second test:

Wordlist/Ruleset	**-**	**Best64**	**Generated2**
Rockyou	19,215 (23.78%)	30,170 (37.33%)	59,134 (73.18%)
PassGAN	10,735 (13.28%)	20,638 (25.54%)	48,751 (60.33%)

Table 2: A comparison of the number of passwords found in the cracking target by RockYou and PassGAN, using a 14 million wordlist for PassGAN

As table 2 shows, PassGAN found roughly 15% more passwords when using a bigger wordlist: this leads us to believe that wordlist size and quality does constitute a performance bottle-neck. It should be also noted that when compared with the original, 1 million wordlist, we found a significant 40% overlap in the passwords that were generated. Addressing the problem of quality of the wordlist, we speculated that one of the reasons for the gap in efficiency between PassGAN and RockYou might be the fact that the strings generated by PassGAN don't follow grammatical rules, and this might have hindered the efficacy of the base wordlist. This hypothesis however was later proven wrong by our natural language experiments, detailed in section 4.4. In reality this gap in performance is probably due to the fact that PassGAN needs a far bigger wordlist to be able to match the performance of RockYou: in section 4.3.2 we discuss this further, and estimate that it would take a wordlist of around 300 million password candidates to bridge the gap.

4.3.1 Running HashCat with caching

Now that we had an idea of the performance of each method, we wanted to see what was the maximum number of passwords we could find by combining the two approaches (RockYou and PassGAN): In order to do this we took the best performing ruleset (generated2), and we performed two rounds of cracking with caching enabled. First we run RockYou with the generated2 ruleset, that yielded the same results shown in Table 2 for that particular combination; next we tried PassGAN plus generated2, which yielded a final result of 63,847 passwords found (79.01% of the cracking target). Running the two approaches in combination seemed to definitely be more affective that using PassGAN alone.

Furthermore when we look at the two lists of cracked passwords, we can notice around 5,000 unique passwords that were matched by PassGAN but not Rockyou, showing that PassGAN might have some potential as a tool to close in on more difficult passwords that have a specific linguistic provenance. Looking at some of these unique passwords, they mostly consist of proper names and more obscure italian words.

4.3.2 Testing the limits of PassGAN wordlist size

As we said previously Hitaj. et al. [8] don't seem to have used rules when testing PassGAN: instead they seem to have taken a different approach, taking advantage of the fact that PassGAN can generate arbitrary numbers of password candidates from a trained model; these password candidates may not match the target as well as the passwords in an established wordlist like RockYou, but PassGAN can generate vast quantities of them: thus Hitaj. et al chose a "quantity over quality" approach.

We were interested in seeing how such huge wordlists would perform without the help of rules, so we run a couple of tests: we took our existing PassGAN model, our standard cracking target `test_10c`, and we generated three new wordlist for PassGAN: one with 50 million entries, one with 100 million entries And finally one with roughly 196 million entries. The reason why we chose these three steps is simple: as we can see in tables 1 and 2, going from 1 million entries to 14 million entries yielded more than double the number of passwords when not using rules, so we wanted to see if testing a wordlist with 14^2 million passwords would yield roughly the same increase compared to the results shown in table 2. The 50 million and 100 million wordlists are intermediate steps that can give us clues on how the number of passwords cracked scales with wordlist size.

Wordlist/Ruleset	**-**	**Best64**	**Generated2**
PassGAN (50 million passwords)	13.898 (17.20%)	-	-
PassGAN (100 million passwords)	15.669 (19.39%)	-	-
PassGAN (196 million passwords)	17.523 (21.68%)	-	-

Table 3: A comparison of the number of passwords found by just PassGAN using different wordlist sizes and no mangling rules

While the increment in passwords found between each of the three wordlists in table 3 is small, overall comparing the 196 million wordlist with the first column of table 2 (the 14 million wordlist) shows a substantial jump in performance of about 60% more passwords found. Whats more, 21.68% is relatively close to the number of passwords found by RockYou: if we assume that the growth rate between 50 million and 196 million password candidates holds, we should be able to match the number of passwords found by RockYou with about 300 million password candidates.

While we consider these results fascinating, they can be mis-leading in the context of the PassGAN paper: in order to outperform RockYou with the genenerated2 rules, Hitaj. et al had to generate around 10^9 password candidates, which if far more than what we have shown here.

We understand why they chose this approach, they saw mangling rules as a hand-coded representation of password features and PassGAN as a tool that autonomously learns those features. PassGAN is seen as a potential replacement for rule-based crackers in [8], so it makes sense that Hitaj. et all did not use mangling rules with PassGAN.

We would argue however than when approaching a set of passwords in another language, this mentality starts to become less relevant: in such a context we feel that the two tools complement each other, they have different roles that do not overlap as much; we see the use of rules with PassGAN as a useful tool that lessens the negative aspects of PassGAN (namely the amount of time and processing power required to train and sample such huge amounts of password candidates), while still reaping the benefits of machine learning systems such as the adaptability to the language of the dataset.

In conclusion, we think the best approach might be similar to what we have done in this section: generate enough samples to match the performance of a commonly used wordlist such as RockYou, and then use PassGAN with rules to match a sizable majority of the passwords while retaining language-specific features.

The point made by Hitaj. et al. is still valid, in principle mangling rules are limiting, and PassGAN can potentially express unique patterns that do not exist in mangling rules, however we are not sure whether this theoretical benefit is worth the additional costs in terms of processing power and storage space in practice. The authors of the

original paper argue that storage space and processing powers are cheap and of no practical concern, we'd be more inclined to favour a faster and more practical approach using mangling rules.

4.4 Training PassGAN with Natural Language corpora

Our next step was to include natural language data in the input data and re-train PassGAN, in the hope that this new input data would help the system generate more grammatically correct strings and thus improve the number of passwords matched.

Firstly though we should define what our goal was for using Natural Language with PassGAN, and what we mean with grammatical correctness.

As we showed at the end of section 4.2, a sizeable number of the password candidates generated by PassGAN during sampling are words: in the context of the strings generated by PassGAN, we define a "word" as simply a string composed of uppercase and/or lowercase letters. It might be entirely composed of letters like the two examples we gave in section 4.2, or have numbers pre-pended or appended to it. As we mentioned in those examples, most of the words generated by PassGAN when trained on passwords exclusively are meaningless: they may resemble italian words, but they are ultimately incorrect. Our goal for training PassGAN with natural language data was to hopefully teach the system enough about the italian language to turn a decent part of those meaningless words into actual italian words. In Turn, we thought, that might have yielded better results in terms of number of password cracked and especially in terms of their quality. We hoped the inclusion of natural language data would allow us to match more language-specific passwords like those 5,000 unique passwords talked about in 4.3.1

For this purpose we have chosen two different corpora of Italian Natural Language samples:

- The Repubblica corpus: A corpus of words extracted from the italian newspaper "Repubblica", taken from articles published between 1985 and 2000 (roughly 380 million words).[19]
- The ItWaC corpus: A large 2 billion word corpus obtained by crawling internet sites under the `.it` domain. [18].

Both corpora were created and used in earlier work [1, 2], and made available through the NoSketch Engine online tool [14].
We decided to sample one dataset from each corpus with the same number of entries as the libero password set: we sampled 700,000 words from both the Repubblica corpus and the ItWac corpus, and those two files served as the basis for our Training.

This was done to ease the process of organizing data for training PassGAN, and also because we were interested in the impact that different ratios of password data to

natural language data might have on the resulting model. Previous research [13] has suggested that introducing Natural Language data into a system trained on passwords tends to generate a lot of noise during training, and our results turned out to be in line with that paper.

Both corpora were simply a list of italian words, ranked by their frequency in the corpus: the two files produced by the NoSkecth tool were a list of words in descending frequency, accompanied by their frequency count.

In order to prepare the natural language data for training, we went through a similar process to what we did for the libero leak passwords: we divided each corpus into 80%-20%, removed the frequency counts and shuffled the words in each corpus. We then used the two 80% sets to make our final training data.

We realized later that dividing the corpora into 80%-20% was not necessary, as we did not use the remaining 20% for testing: in hindsight, we could have avoided this step.

The shell commands we used to accomplish this are shown below.

```
#Split each NL corpus into 80%-20% (not strictly necessary)
head -n 534172 Repubblica_700k.txt|cut -f1 > Repubblica-80.txt
tail -n 133543 Repubblica_700k.txt|cut -f1 > Repubblica-20.txt

head -n 534172 ItWac_700k.txt|cut -f2 > ItWac_80.txt
tail -n 133543 ItWac_700k.txt|cut -f1 > ItWac_20.txt

#Create training data for PassGAN with password data + 50% of the
   ↪ Repubblica corpus
shuf -n 267086 Repubblica-80.txt|cat - ../libero\ it\ July\ 2016/
   ↪ train.txt|shuf > libero+Repubblica-50.txt

#Create training data for PassGAN with password data + 100% of the
   ↪  repubblica corpus
shuf Repubblica-80.txt|cat - ../libero\ it\ July\ 2016/train.txt|
   ↪ shuf > libero+Repubblica.txt

#Create training data for PassGAN with password data + 50% of the
   ↪ ItWaC corpus
shuf -n 267086 ItWac_80.txt|cat - ../libero\ it\ July\ 2016/train.
   ↪ txt|shuf > libero+ItWac-50.txt

#Create training data for PassGAN with password data + 100% of the
   ↪  ItwaC corpus
shuf ItWac_80.txt|cat - ../libero\ it\ July\ 2016/train.txt|shuf >
   ↪  libero+ItWac.txt
```

As we can see in the code snippet above, we created two training datasets for each corpus: one contained all the passwords in the Libero Training set and 50% of the Natural language data, while the other contained all of the Training set passwords and all of the Natural Language data: we did this because we wanted to see what impact different amounts of natural language data had on PassGAN's performance (if any).

This left us with a total of four datasets ready for training,

We then proceeded to train 4 separate instances of PassGAN with our four datasets, and then sampled each trained model for testing: the results were 4 separate wordlists, each of around 14 million entries, that we used in HasCat to attack the same cracking target that we used in the previous section (test_10c).

Touching upon the impact of NL data on the model performance, our hypothesis was that we might see a decrease in model performance when training with 100% of the natural language data, since each corpus contains as much data as the Libero Training set holds passwords; By combining the two into a single dataset for training,

we effectively use as much NL data as password data. We though this might lead to noise and "confuse" PassGAN, resulting in a model with less focus on generating passwords and more focused on generating NL samples. While that turned out to be technically true, overall table 4 shows that the introduction of NL data does not seem to have any noteworthy impact on cracking performance.

The results of our tests on all four wordlists are shown in table 4 below.

Wordlist/Ruleset	**-**	**Best64**	**Gen2**
Libero+Repubblica 50%	-	-	51,232 (63.40%)
Libero+Repubblica 100%	-	-	48,651 (60.20%)
Libero+ItwaC 50%	-	-	50,133 (62.04%)
Libero+ItWaC 100%	-	-	47,646 (58.96%)

Table 4: Number of passwords found using PassGAN, trained on passwords and Natural Language data

As we can see, the results of training with natural language data are very similar to just using PassGAN+gen2 as shown in table 2; there is a slight improvement of 2-3% when using the datasets containing 50% of language data, but we do not believe it is a significant change. This result seems to be in line with Melicher et al.[13], and it seems to show us that natural language data does not have much of an impact on the output of PassGAN. If we take a random sample from the Repubblica 50% set, the one that performed the best even if marginally so, we can see that there has not been a substantial improvement in the grammatical correctness of the words in the sample.:

```
c3b243dc    fetemanio   carcilla    RATS1203
valentto    itefis82    carmoinx    gip1904
sederonco   elestarsia  220689      QunkYYTx84
n!utelo     kedea       Colpudari   rich1770
aletsa      simoshero   gidni12     1uva1[78
```

The words that PassGAN generates are still meaningless and do not seem to have a higher degree of grammatical correctness, the only change we have been able to observe is in the fact that more password candidates seem to be composed of letters exclusively.

This does not discredit the use of Natural language as a whole, but points us towards the fact that natural language generation and password generation might be two different workloads that may not be accomplished well with the same machine learning system: it seems that introducing Natural Language data into PassGAN simply adds noise to the system and does not contribute significantly to the model's performance. The results in table 4 and the effectiveness of mangling rules as a

cracking tool also hint at the idea that user-generated passwords may be defined more by their patterns than by their language of origin.

5 Discussion

In this coming section we are going to discuss what we see as one of the potential shortcomings of this thesis, and give some ideas for avenues of future work.

5.1 Evaluating PassGAN's performance fully

While we were overall pretty happy with the results we obtained from our testing, we feel that there is a potential problem in our arguments that we need to address: its the fact that we did not try to crack the Libero passwords using the pre-trained PassGAN model that Hitaj. et al. [8] used in their paper.
We think this is a rather important omission, as doing that would have given us a base-line performance that we could compare our PassGAN model to. This comparison would have allowed us to determine what were the effects of using a model trained on largely english-language passwords to crack a dataset of italian passwords. We are still evaluating PassGAN in our thesis, but we feel that this comparison between two very different PassGAN models might have given us important insights into the performance of PassGAN with italian passwords.

Speaking of avenues of further work, there were some aspects of our thesis that we would have liked to expand upon or include:

- We would have liked to investigate what is the ideal ratio of natural language data to password data for training PassGAN.
- We could have performed more testing of huge wordlists, in the context of cracking italian passwords.
- As a follow-up to this thesis, we would be interested in an exploration of how PassGAN performs with Passphrases instead of Passwords.

5.2 Future work: natural language ratios

In section 4.4 we have covered our results in introducing natural language data into PassGAN for training, but we only tested two ratios of NL data to password data (50% and 100%): had we had more time we would have liked to experiment with more combinations and test their effects, maybe there is a combination that would actually boost PassGAN performance.

5.3 Future work: testing huge wordlists further

In section 4.3.2 we talked about trying to match passwords using the same method as Hitaj. et al. used (only using PassGAN wordlists and no rules).
We would have liked to explore this further, but sadly we started to work with huge wordlists very late in our thesis, and we did not have the time to fully integrate this

aspect in our testing. Specifically, we wanted to repeat the experiment presented in section 4.3.2 using our model trained on natural language data.

Because of this poor timing and the late realization that we were not using PassGAN in an optimal way, the results in our testing chapter end up being skewed in favour of rule-based password crackers.

As we started using bigger and bigger PassGAN wordlists, we also recognized that testing PassGAN with the same amount of password candidates that Hitaj. et al. used was beyond the scope of this thesis.

It could have been interesting to test how the thesis and results put forth by the original paper hold in the context of our thesis; As we mention in section 4, we believe that PassGAN should be used in conjunction with HasCat and other such tools instead of being considered a replacement for them.

The goal of this kind of work would be to try to reproduce the results from the original paper, but in the context of italian passwords: it would be an extension of this paper, in which we try the same approach as the original paper to see if there are changes in the results. In this thesis we were not interested in reproducing the results from [8], both because we did not think it was relevant (our approaches and datasets are different), and because it would have been extremely challenging to do so within the time we were given for this thesis: as an example, the original paper states that PassGAN was allotted a maximum of 5^{10} password candidates it could use to out-perform all the other methods: to just sample that many passwords at our current rate, it would have taken us more than 180 days of nothing but sampling.

5.4 Future work: passphrases

Our favourite way to build on this paper would no doubt be to do a follow-up study looking at passphrases, as a way to test the limits of PassGAN as a language-sensitive specialist cracker: one might make the case that by their nature, passphrases are more subject to grammatical and syntactical rules, and it would be interesting to repeat the experiments in this thesis using passphrases as a dataset.

As [13] states, rule-based password crackers tend to have a ceiling on the length and complexity of passwords they can efficiently crack: A similar study using passphrases might also test the limits of rule-based tools in such a way, that we might find PassGAN to be a more effective tool for this particular niche. The big problem we see with this research is the lack of data: passphrases are not usually widely employed by companies in their password policies, and there is a lower chance of leaked passphrases datasets being available to the public.

6 Conclusion

In this thesis we have looked at the impact that language has on PassGAN in various ways: To conclude our thesis, we would like to answer the questions posed in our problem formulation all the way back in section 1.

- *How does PassGAN perform when cracking Italian passwords?*

  Overall we can say that PassGAN performed worse in our experiments when compared to rule-based password crackers, but we did not play to PassGAN's strengths: in most of our tests we did not take advantage of the huge potential number of password candidates that PassGAN can output, but we also showed in section 4.3.2 that PassGAN can potentially match the performance of rule-based tools by just using bigger wordlists. This conclusion was first made in Hitaj. et al. but it holds true in our experience as well.

  Comparing the number of password candidates needed to achieve the same performance in RockYou and PassGAN, using RockYou with the generated2 ruleset gives us a maximum number of possible password candidates of around $9,1 \times 10^{10}$: if we take our rough estimate of 300 million password candidates to match the performance of the base RockYou wordlist, we will find that PassGAN needs around three orders of magnitude more password candidates to match the performance of RockYou+generated2.

- *Does difference in language have an impact on the passwords found by PassGAN and state-of-the-art password crackers?*

  We would say yes: even when PassGAN perform worse than traditional password crackers in terms of total number of passwords found, we argue that it is capable of finding unique and novel passwords that RockYou would not be able to match. An example of this was presented in section 4.3.1, where PassGAN found 5000 unique passwords that were not matched by RockYou. There we used a 14 million wordlist for PassGAN, the potential for unique passwords can be much higher if we start using bigger wordlists.

- *How does the inclusion of natural language data during training affect PassGAN's performance?*

  From our experience, the inclusion of natural language data does not seem to have a positive impact on the performance of PassGAN: in fact, it had a slightly negative impact. Overall PassGAN's Performance using NL data can broadly be considered on par with a PassGAN model trained with just passwords.

- *Ultimately, can PassGAN be a useful tool when approaching password data from a particular language area?*

  Yes. In our opinion PassGAN can be a useful tool to crack passwords from a different language: while rule-based password crackers may be considered better at first, the flexibility that PassGAN provides ultimately leads us to believe it is better.

Thanks to PassGAN an attacker has a variety of approaches at his disposal: they can either use PassGAN as a specialist tool to supplement rule-based crackers, or use it as a replacement for rule-based tools as Hitaj. et al. suggest [8].

In the first case PassGAN would be used to tackle passwords that may not be part of wordlists (e.g passwords containing proper names, slang, idioms or other elements specific to a language). We would encourage the use of mangling rules with PassGAN in this case, since PassGAN's role would be to generate language-specific passwords, while mangling rules are used to apply the patterns typical in user-generated passwords to those candidates. This particular approach takes the best of both worlds, while still relying mainly on rule-based tools.

On the other hand if the data is valuable enough, an attacker can invest the time and money required to use PassGAN to its full potential following the idea Hitaj. et al [8] outlined in their paper. It may take a lot longer and require more storage space, but PassGAN would eventually crack more passwords than any rule-based tool if the authors of the original paper are right.

That said we don't think that PassGAN is necessarily the best application of deep learning to this field: Hitaj. et al. demonstrated that it can be effective, but we speculate that there might be better approaches. For example if PassGAN generated mangling rules directly instead of password candidates, one might get the similar results with far lower costs in terms of processing time or storage space required. On the other hand this approach might be more difficult, as the output of the network is more abstract than just strings: rules have to be written in a specific context-free or regular language, and this might present additional challenges.

In conclusion, we believe Deep Learning systems are definitely capable of dealing with passwords from a non-english source: they provide a level of flexibly and adaptability that are not present in rule-based password crackers by default. Whats more, their ability to learn password and language features from available data provides a new and unique approach to password cracking that is not limited by rules or wordlists compiled by human actors.

References

[1] Marco Baroni et al. "Introducing the La Repubblica Corpus: A Large, Annotated, TEI (XML)-compliant Corpus of Newspaper Italian." In: *LREC*. 2004.

[2] Marco Baroni et al. "The WaCky wide web: a collection of very large linguistically processed web-crawled corpora". In: *Language resources and evaluation* 43.3 (2009), pp. 209–226.

[3] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 2014, pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.

[4] *Hackerato il servizio di Libero Mail: rubate le password*. https://www.tomshw.it/altro/hackerato-il-servizio-di-libero-mail-rubate-le-password/. Accessed: 15-05-2019.

[5] *hashcat: advanced password recovery*. https://hashcat.net/hashcat/. Accessed: 26-02-2019.

[6] *hashcat documentation*. https://hashcat.net/wiki/. Accessed: 02-03-2019.

[7] *hashcat documentation: Mask Attack*. https://hashcat.net/wiki/doku.php?id=mask_attack. Accessed: 02-03-2019.

[8] Briland Hitaj et al. "PassGAN: A Deep Learning Approach for Password Guessing". In: *CoRR* abs/1709.00440 (2017). arXiv: 1709.00440. URL: http://arxiv.org/abs/1709.00440.

[9] *John the Ripper password cracker*. https://www.openwall.com/john/. Accessed: 26-02-2019.

[10] *Le password della mail di Libero sono in pericolo*. https://www.wired.it/attualita/tech/2016/09/07/libero-password-mail/. Accessed: 15-05-2019.

[11] *Libero Mail sotto attacco hacker, violato il database: consigliato cambiare la password*. https://tech.fanpage.it/libero-mail-sotto-attacco-hacker-violato-il-database-consigliato-cambiare-la-password/. Accessed: 15-05-2019.

[12] *Libero.it password leak: An analysis in-depth*. https://www.scip.ch/en/?labs.20180913. Accessed: 11-03-2019.

[13] William Melicher et al. "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016, pp. 175–191. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/melicher.

[14] *NoSketch Engine*. https://corpora.dipintra.it/public/run.cgi/wordlist_form?corpname=ukwac1. Accessed: 02-05-2019.

[15] *Source for the Libero dataset*. https://cdn.databases.today/Libero.it%20900k.zip. Accessed: 15-05-2019.

[16] Robert H. Morris Sr. and Ken Thompson. "Password Security - A Case History". In: *Commun. ACM* 22.11 (1979), pp. 594–597. DOI: 10.1145/359168.359172. URL: https://doi.org/10.1145/359168.359172.

[17] National Institute of Standards and Technology. *NIST Special Publication 800-63B: Authentication and Lifecycle Management*. Tech. rep.

[18] *The ItWaC corpus*. http://wacky.sslmit.unibo.it/doku.php?id=corpora. Accessed: 30-04-2019.

[19] *The Repubblica corpus*. http://docs.sslmit.unibo.it/doku.php?id=corpora:repubblica. Accessed: 30-04-2019.

[20] *VirusTotal report: libero.it-900k.zip*. https://www.virustotal.com/#/file/de22457614b404fff8e7038d889b9ecdc9bbd6692dfdde4016c3170101ded785/detection. Accessed: 15-05-2019.