PCFG Developers Guide

Release 4.3

Matt Weir

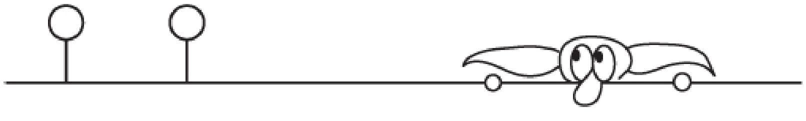
Nov 26, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION:



PCFG = Probabilistic Context Free Grammar

PCFG = Pretty Cool Fuzzy Guesser

In short: A collection of tools to perform research into how humans generate passwords. These can be used to crack password hashes, but also create synthetic passwords (honeywords) and to help develop better password strength algorithms.

Version: 4.3

This project uses machine learning to identify password creation habits of users. A PCFG model is generated by training on a list of disclosed plaintext/cracked passwords. In the context of this project, the model is referred to as a ruleset and contains many different parts of the passwords identified during training along with their associated probabilities. The generation of this ruleset can loosely be consider a machine learning stemming algorithm, and the resulting partial passwords in the ruleset can be useful for other cracking tools such as PRINCE, and/or can be directly incorporated into more traditional dictionary-based attacks. This project also includes a PCFG guess generator that makes use of this ruleset to generate password guesses in probability order. This is much more powerful than standard dictionary attacks and in testing has proven to be able to crack passwords on average with significantly less guesses than other publicly available methods. The downside is that generating guesses in probability order is slow, meaning it is creating on average 50-100k guesses a second, where GPU based algorithms can create millions to billions (and up), of guesses a second against fast hashing algorithms. Therefore, the PCFG guesser is best used against large numbers of salted hashes or other slow hashing algorithms where the performance cost of the algorithm is made up for with the accuracy of the guesses.

This Developer's Guide was written for three different audiences:

1. Power users who want to take full advantage of the PCFG configuration options, or want to manually modify rulesets
2. Developers who want to add or extend PCFG features
3. Developers who want to create their own password cracking tools

To this end, this Developer's Guide will contain a mix of feature descriptions, underlying code documentation, and password cracking theory. Part of the drive to document this code is I'm not doing much development with this tool. It's been three years since the last major update, and I want to make this easier for other people (you) to pick this up and modify it to fit your own needs.

If you have any suggestions or changes please don’t hesitate to reach our or to submit a merge request.

CHAPTER

TWO

BUILDING DOCUMENTATION INSTRUCTIONS:

2.1 How to Build/Update the Developer's Guide

Sphinx is used to create the Developer's Guide. To install Sphinx (the following assumes that python3 matches your Python 3 environment):

python3 -m pip install sphinx

For the following commands you should run them from the docsdirectory.

To create an HTML viewable version of the Developer's Guide run:

sphinx-build -b html docssource docsbuild

Then you can view the Developer's Guide at docsbuildindex.html

To create a PDF of the Developer's Guide you will need a LaTeX distribution, the latexmk package, and likely PERL as well. Instructions for how to do that are below. Once you have compleated that you can generate the PDF of the guide by running:

make latexpdf

2.1.1 Building a PDF Dev Guide on Windows:

First install sphinx as described above. Next you will need to install a LaTeX editor.

One option is to install https://miktex.org/. There is a bunch of spammy links on the website. My apologies. Click on the download link. Do NOT click on the "START NOW" link. Once you have it installed, open the Miktex program, click packages button and then select latexmk to install that package. Around this point you might realize that building the html documentation might have been easier. Let me apologize on behalf of the academic community. The next step is you'll need to install Perl. https://www.perl.org/get.html. I use strawberryperl simply because the install process was less annoying. A quick link to that is https://strawberryperl.com/.

Ok, if you had any command prompt windows open, you'll now need to close and reopen them for the links to latexmk to work. Now you should hopefully be able to build a PDF of the document.

Next, in a terminal or command prompt navigate to the ./docs directory and run: *make latexpdf*. The build may require installing additional LaTeX packages. I had to install what felt like 50 of them for the basic “hello world” PDF example. I swear I am not knowingly installing any malware or backdoors on your computer with this. If all of this worked you should now be able to view the PDF in docs\build\latex\pcfgdevelopersguide.pdf

If you receive an error saying that an image can not be found while building the PDF, my experience is that MiKTex has gotten confused. In the MiKTex GUI/console, click on Tasks and select “Refresh file name database”. If this doesn’t fix the problem, update all the packages in MiKTex and then refresh the file name database again.

2.1.2 Building a PDF Dev Guide on Linux:

First install sphinx as described above. If you get an error along the lines of */bin/sh: 1: sphinx-build: not found* you did not install sphinx to the Python deployment it is trying to use. I’m sorry if that’s happening, and while I can refer you to the sphinx documentation: https://www.sphinx-doc.org/en/master/usage/installation.html, your problem may be deeper than that with how Linux distros handle different Python environments. Basically if you were chuckling at the craziness with Windows, well this is the Linux equivalent.

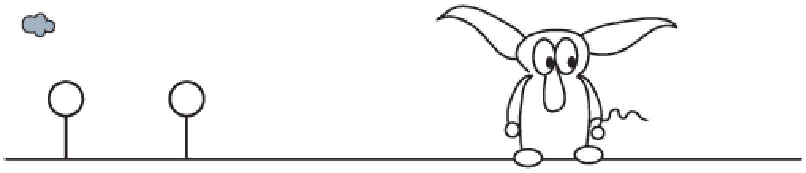
Next you will need to install a LaTeX editor. In this case, the easiest one to install is *latexpdf* which hopefully should be easier to do on most Linux distros than on Windows. For example, on Ubuntu 20.1 you can simply type *sudo apt-get install -y latexmk*. Note, this installs what seems like a million fonts and templates. Once again, you have my word I’m not trying to hack you.

Next you need to install even more LaTeX extras to get cmap.sty. Why do you need cmap.sty? I’m not really sure, but it doesn’t work without it. To get it, on Ubuntu you can run *sudo apt install -y texlive-latex-extra*

CHAPTER

THREE

REQUIREMENTS + INSTALLATION



- Python3 is the only hard requirement for these tools
- It is **highly recommended** that you install the chardet python3 library for training. While not required, it performs character encoding autodetection of the training passwords. While this is often installed with pip3 this is not a core part of Python3. To install it:
- Download the source from https://pypi.python.org/pypi/chardet
- Or install it using `pip3 install chardet`

CHAPTER

FOUR

QUICK START GUIDE

4.1 Training

The default ruleset included in this repo was created by training on a 1 million password subset of the RockYou dataset. Better performance can be achieved by training on the full 32 million password set for RockYou, but that was excluded to keep the download size small. You can use the default ruleset to start generating passwords without having to train on a new list, but it is recommended to train on a target set of passwords that may be closer to what you are trying to target. If you do create your own ruleset, here is a quick guide:

1. Identify a set of plaintext passwords to train on.

- This passwords set should include duplicate passwords. That way the trainer can identify common passwords like 123456 are common.
- The passwords should be in plaintext with hashes and associated info like usernames removed from them. Don't try to use raw .pot files as your training set as the hashes will be considered part of the password by the training program.
- The passwords should be encoded the same way you want to generate password guesses as. So, if you want to create UTF-8 password guesses, the training set should also be encoded as UTF-8. Long term, the ability to modify this when generating guesses is on the development plan, but that feature is currently not supported.
- The training password list should be between 100k and 50 million. Testing is still being done on how the size of the training password list affects guess generation, and there has been good success even with password lists as small as 10k, but an ideal size is likely around 1 million, with diminishing returns after that.
- For the purposes of this tutorial the input password list will be referred to as INPUT_PASSWORD_LIST

2. Choose a name for your generated ruleset. For the purposes of this tutorial it will be NEW_RULESET

3. Run the trainer on the input password list

- `python3 trainer.py -t INPUT_PASSWORD_LIST -r NEW_RULESET`

4. Common optional flags:

- **-coverage**: How much you trust the training set to match the target passwords. A higher coverage means to use less intelligent brute force generation using Markov modeling, (currently using the OMEN algorithm). If you set coverage to 1, no brute force will be performed. If you set coverage to 0, it will only generate guesses using Markov attacks. This value is a float, with the default being 0.6 which means it expects a 60% chance the target password's base words can be found in the training set. Example: `python3 trainer.py -t INPUT_PASSWORD_LIST -r NEW_RULESET -c 0.6`
- **–save_sensitive**: If this is specified, sensitive data such as e-mail addresses and full websites which are discovered during training will be saved in the ruleset. While the PCFG guess generator does not currently make use of this data, it is very valuable during a real password cracking attack. This by default is off to make this tool easier to use in an academic setting. Note, even when this is off, there will almost certainly still be PII data saved inside a ruleset, so protect generated rulesets appropriately. Example: `python3 trainer.py -t INPUT_PASSWORD_LIST -r NEW_RULESET --save_sensitive`
- **–comments**: Adds a comment to your ruleset config file. This is useful so you know why and how you generated your ruleset when looking back at it later. Include the comment you want to add in quotes.

4.2 Guess Generation

This generates guesses to stdout using a previously training PCFG ruleset. These guesses can then be piped into any program that you want to make use of them. If no ruleset is specified, the default ruleset **DEFAULT** will be used. For the purposes of this guide it will assume the ruleset being used is **NEW_RULESET**.

1. Note: the guess generation program is case sensitive when specifying the ruleset name.

- A session name is not required, (it will by default create a session called **default_run**), but it is helpful to make restarting a paused/stopped session easier. These examples will use the session name **SESSION_NAME**. Note, there is no built in sanity check if you run multiple sessions with the same name at the same time, but it is recommend to avoid that.

2. To start a new guessing session run:

- `python3 pcfg_guesser.py -r NEW_RULESET` -s SESSION_NAME

3. To restart a previous guessing session run:

- `python3 pcfg_guesser.py -s SESSION_NAME --load`

4.3 Password Strength Scoring

There are many cases where you may want to estimate the probability of a password being generated by a previously trained ruleset. For example, this could be part of a password strength metric, or used for other research purposes. A sample program has been included to perform this. - INPUT_LIST represents the list of passwords to score. These passwords should be plaintext, and separated by newlines, with one password per line.

1. To run a scoring session: `python3 password_scorer -r NEW_RULESET -i INPUT_LIST`
2. By default it will output the results to stdout, with each password scored per line

- The first value is the raw password
- The second value will represent if the input value was scored a 'password', 'website', 'e-mail address', or 'other'. This determination of password or other is dependent on the limits you set for both OMEN guess limit, as well as probability associated with the PCFG.

- The third value is the probability of the password according to the Ruleset. If it is assigned a value of 0.0, that means that the password will not be generated by the ruleset, though it may be generated by a Markov based attack
- The fourth value is the OMEN level that will generate the password. A value of -1 means the password will not be generated by OMEN.

4.4 Prince-Ling Wordlist Generator

Name: PRINCE Language Indexed N-Grams (Prince-Ling)

Overview: Constructs customized wordlists based on an already trained PCFG ruleset/grammar for use in PRINCE style combinator attacks. The idea behind this was since the PCFG trainer is already breaking up a training set up passwords into individual parsings, that information could be leveraged to make targeted wordlists for other attacks.

Basic Mechanics: Under the hood, the Prince-Ling tool is basically a mini-PCFG guess generator. It strips out the Markov guess generation, and replaces the base structures used in normal PCFG attacks with a significantly reduced base-structure tailored for generating PRINCE wordlists. This allows generating dictionary words in probability order with an eye to how useful those words are expected to be in a PRINCE attack.

Using Prince-Ling

1. Train a PCFG ruleset using trainer.py. Note you need to create the ruleset using version 4.1 or later of the PCFG toolset, as earlier versions did not learn all the datastructures that Prince-Ling utilizes.
2. Run Prince-Ling `python3 prince-ling.py -r RULESET_NAME -s SIZE_OF_WORDLIST_TO_CREATE -o OUTPUT_FILENAME`

- **–rule**: Name of the PCFG ruleset to create the PRINCE wordlist from
- **–size**: Number of words to create for the PRINCE wordlist. Note, if not specified, Prince-Ling will generate all possible words which can be quite large depending on if case_mangling is enabled. (Case mangling increases the keyspace enormously)
- **–output**: Output filename to write entrees to. Note, if not specified, Prince-Ling will output words to stdout, which may cause problems depending on what shell you are using when printing non-ASCII characters.
- **–all_lower**: Only generate lowercase words for the PRINCE dictionary. This is useful when attacking case-insensitive hashes, or if you plan on applying targeted case mangling a different way.

4.5 Example Cracking Passwords Using John the Ripper

```
python3 pcfg_guesser -r NEW_RULESET -s SESSION_NAME | ./john --stdin --format=bcrypt
PASSWORDS_TO_CRACK.txt
```

CHAPTER

FIVE

ADVANCED EXAMPLE (BUILDING THE RUSSIAN TRAINING RULESET)

5.1 Overview:

Part of the feature enhancements of Version 4.3 of the PCFG Toolset is enhanced non-ASCII character set support. To demonstrate this, I figured it was time to release a new ruleset trained on non-English passwords to to complement the "Default" ruleset which was trained on a subset of the RockYou passwords. Part of the challenge of training a PCFG ruleset is finding a set of passwords that pairs well with the training process. It helps if the ruleset is created from a set of plaintext passwords. This is because you can't train on passwords you haven't cracked. Duplicate passwords also results in a more effective PCFG Ruleset. Otherwise how will you know that '123456' is more common than '348251'? Also, public password dumps have a lot of … for lack of a better term … junk in them. So finding a publicly available password dump that is relatively clean can be challenging. Finally you want to have a training set that resembles passwords that you are trying to target. One password list that I have that meets many of those requirements is from VK.com which is the Russian equivalent of Facebook. The list I have was supposedly from a 2012 hack, (there's been other reported hacks, including one recently), and it made its way through many different hands before I was able to obtain a copy. To read more about the hack, here is a reference: https://thehackernews.com/2016/06/vk-com-data-breach.html. What's useful is that this list represents a good set of password for Russian speakers, the passwords are in plaintext, it includes duplicate passwords, and the copy I received seems to be relatively well formatted.

5.2 A quick note on ethics:

The password set I obtained did not include usernames or e-mail addresses. It only included the passwords. This was a bonus for me. I don't want to associate individual passwords with real people. While the PCFG grammar does a decent job of breaking down individual passwords, it does include private data in the generated ruleset that users included in their passwords. That's a big reason why I selected such an old set. All the passwords in the VK password list are at least 10 years old and have been extensively traded in underground circles. Therefore I believe the research use of such a ruleset outweighs the privacy impact of making portions of this list partially available in the PCFG code repository. I have also not validated that these passwords are real, and from the actual 2012 vk.com hack. In fact, there's no way for me to truly validate this list. Once again, I view this as a bonus. While there are indicators that this list is in fact legit, (which I'll talk about in the training portion of this chapter), this list could be a fabrication.

5.3 Pre-processing the List:

Since I'm planning on releasing this ruleset as part of the PCFG code repository, I want to keep the generated grammar relatively small. To this end, I selected a one million password subset of the original vk.com list to use in training. Previous tests have shown that while larger training sets can help improve the effectiveness of PCFG grammars, the value of larger training sets diminishes drastically after the first million passwords. In fact, even training on only 100 thousand password can be quite effective, but one million is a nice pleasing sounding number. An added advantage of this approach is I can then the remaining passwords to run tests against.

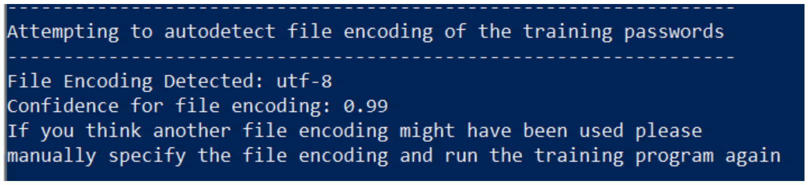
One tool that I would highly recommend to clean up password lists before training on is: https://github.com/NetherlandsForensicInstitute/demeuk. If you want to crack passwords like Europol agents, Demeuk is the tool for you. It is very effective at fixing encoding issues, cleaning up weird junk that tends to show up in password lists, as well as tailoring the training set to the password policies you are targeting. I'm going to talk about Demeuk more later, but as some background, I did not use it for the Russian ruleset included in this repo. I went back and forth on this, but in this case the testing I did showed that the code cleanup options in the core-PCFG code were sufficient for processing the training list without using Demeuk.

5.4 Training on Vk Passwords:

To create the Russian ruleset, I ran the PCFG trainer and only specified the training list and the ruleset name. I left the rest of the options at their default settings.

```
python3 trainer.py -t vk_1m.txt -r Russian
```

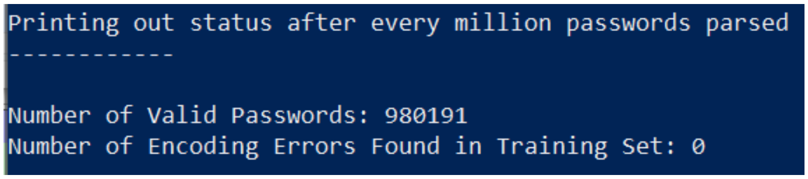
Doing this told me a couple of things. The first thing was that the encoding of this dataset was likely UTF-8, which is really helpful since that makes it more widely applicable for other cracking sessions. Here is an output from my training session that shows that.



```
------------------------------------------------------------------
Attempting to autodetect file encoding of the training passwords
------------------------------------------------------------------
File Encoding Detected: utf-8
Confidence for file encoding: 0.99
If you think another file encoding might have been used please
manually specify the file encoding and run the training program again
```

I've seen a number of other Russian password sets encoded as ISO 8859-5, or MacOS Cyrillic. This means converting your training set to the type of encoding you are targeting is very important. I'll cover that in more detail later. For now though, it's helpful to highlight that the Russian PCFG ruleset included in this code repo is made to target UTF-8 encoded passwords.

You can verify the auto-detected encoding by looking at the number of encoding errors encountered when parsing the entire password set. You can see that in the following screenshot that no encoding errors were encountered during the training process.

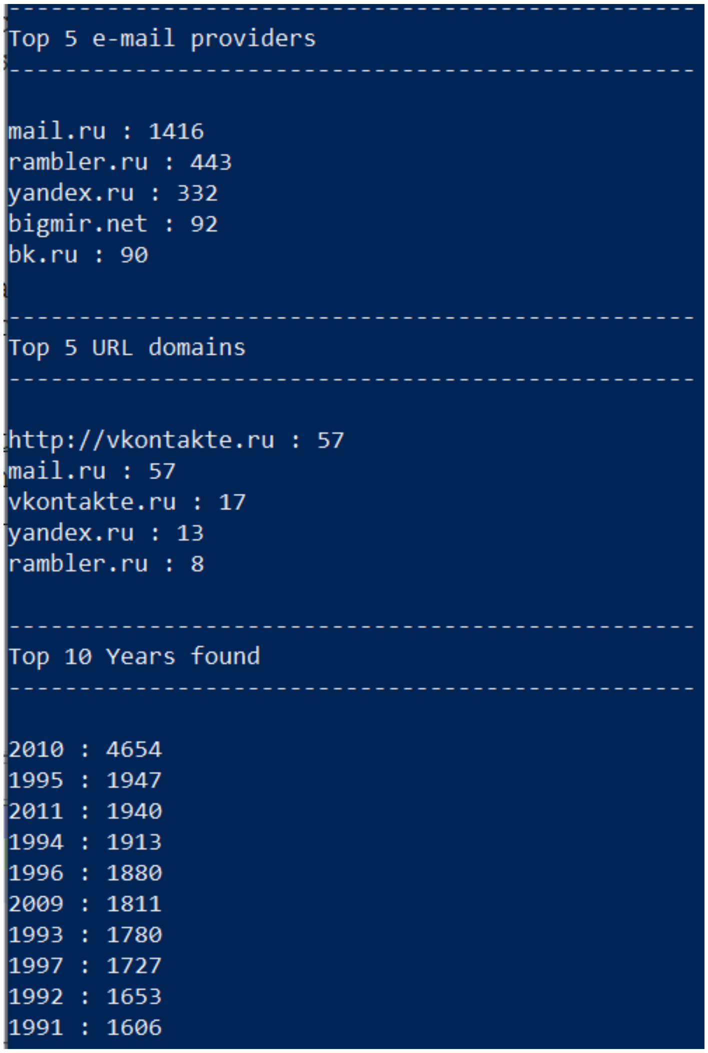


```
Printing out status after every million passwords parsed
------------

Number of Valid Passwords: 980191
Number of Encoding Errors Found in Training Set: 0
```

You might also notice that it only detected around 980 thousand valid passwords even through I passed it a training list of one million passwords. There's a number of reasons a line/password in the training list can be rejected, but the most common reason is the trainer will reject blank lines. This also means that a PCFG based attack will never guess a "blank" password. I've gone back and forth on if this is desired behavior or not, but given how messy training lists tend to be, rejecting blank lines generally seems to be the right call. This also means you might want to run a quick check to validate that a password exists and was not blank before running a PCFG based cracking attack. Getting back to the VK list, I opened it up the training file troubleshoot this, and yes, it turned out that roughly 2% of the lines were blank.

The final point I want to highlight about the training process is the statistics the trainer produces at the end. The following picture shows the statistics generated from training on the Vk list.



```
-------------------------------------------------
Top 5 e-mail providers
-------------------------------------------------

mail.ru : 1416
rambler.ru : 443
yandex.ru : 332
bigmir.net : 92
bk.ru : 90

-------------------------------------------------
Top 5 URL domains
-------------------------------------------------

http://vkontakte.ru : 57
mail.ru : 57
vkontakte.ru : 17
yandex.ru : 13
rambler.ru : 8

-------------------------------------------------
Top 10 Years found
-------------------------------------------------

2010 : 4654
1995 : 1947
2011 : 1940
1994 : 1913
1996 : 1880
2009 : 1811
1993 : 1780
1997 : 1727
1992 : 1653
1991 : 1606
```

Looking at the e-mail list, it certainly appears this training list was created from Russian speakers. Side note, bigmir.net is a Kiev based Ukrainian site, which is a good reminder that just because a list is made up of Russian speakers, it doesn't mean they are all Russian users.

Looking at the top five URLs, you might notice a number of the top e-mail providers there as well. This is a common occurrence I've seen across multiple different password dumps. Part of this is a limitation of the trainer. If the trainer parses username@mail.ru, the trainer can tell it is an e-mail account. If the trainer parses mail.ru though, that string without an '@' symbol looks like a URL. Probably the best way to handle this would be to identify e-mail providers during the first pass the trainer runs on the input list, but I haven't gotten around to implementing that yet. This is all a long way to say you can usually skip past e-mail providers in the URL domains. Looking at the remaining URLs, we can see that vkontakte.ru is highlighted, which gives further credence that this password list came from Vk. This approach is very useful for identifying where random password lists originated from, which is something I end up dealing with quite a bit.

When it comes to years, you'll notice that 2010 is the most common year, despite the fact that this list supposedly was stolen around 2012. This actually matches up with behavior seen in other password disclosures as well. You need to remember that unless a mandatory password change policy was in place, most of the passwords will have been created a year or two before the breach. Therefore you should expect a similar distribution. Opening up the Digits/1.txt file in the generated ruleset, I could see that 2012 was actually the 36th most common date found, with 2013 being the 76th most common date. This seems to validate the story that the password list was stolen in early 2012. The other point I'd like to raise is to highlight the prevalence of dates in the early 1990's. Since people tend to use their birthdays in their passwords, this implies that at the time of the theft, the user population skewed to be older teenagers to people in their early 20's.

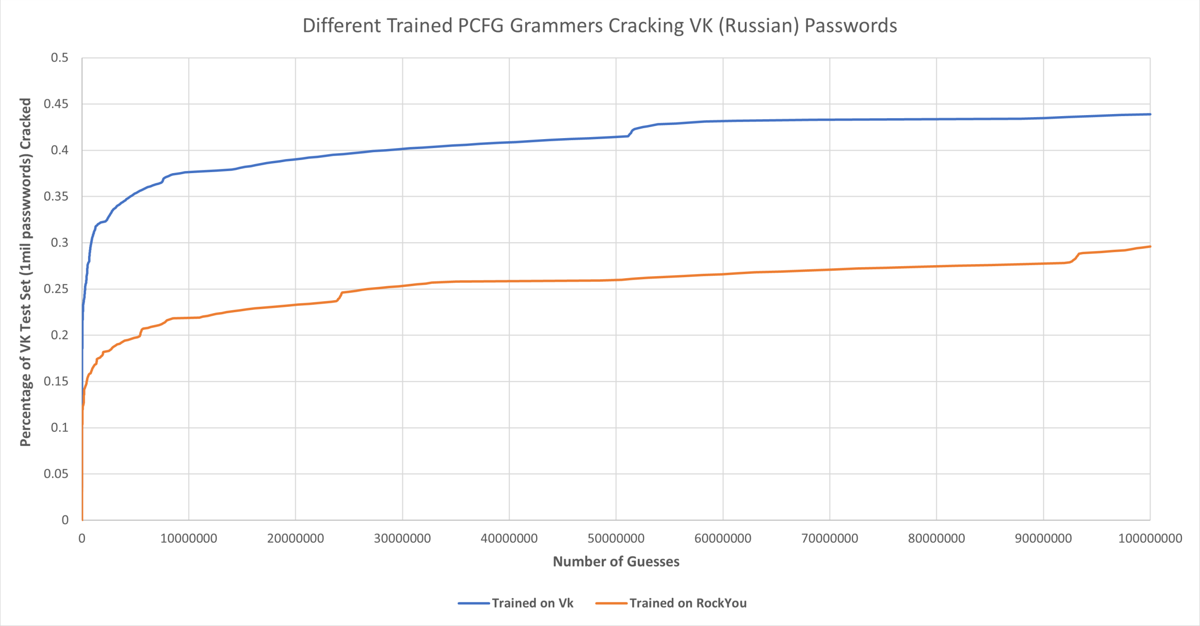
5.5 Testing the Russian Ruleset

The next step was to test the effectiveness of the Russian ruleset. As mentioned earlier, I had divided up the VK list into 1 million password sized sub-lists, which allows running an initial test against a set of very similar, but different, users compared to the ones found in the training list. While it is certainly possible to use a traditional password cracking program to test the ruleset, I like to use the checkpass program which is part of: https://github.com/lakiw/Password_Research_Tools, since it allows me to graph the results. I then ran two cracking sessions using both the Russian ruleset, and the "Default" ruleset generated from RockYou. The following are the commands I used to test a cracking session of 100 million guesses:

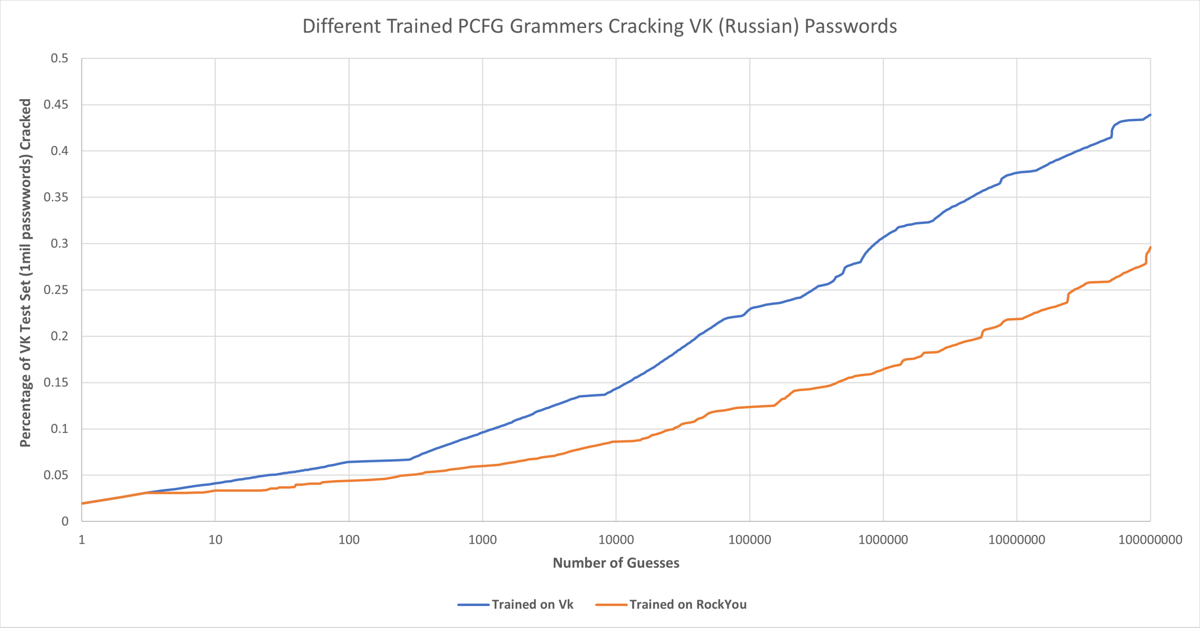
```
python3 pcfg_guesser.py -r Russian | ../Password_Research_Tools/checkpass.py -t ../../
↪research/password_lists/vk_1m-test-2.txt -m 100000000

python3 pcfg_guesser.py -r Default | ../Password_Research_Tools/checkpass.py -t ../../
↪research/password_lists/vk_1m-test-2.txt -m 100000000
```

And below are the results:

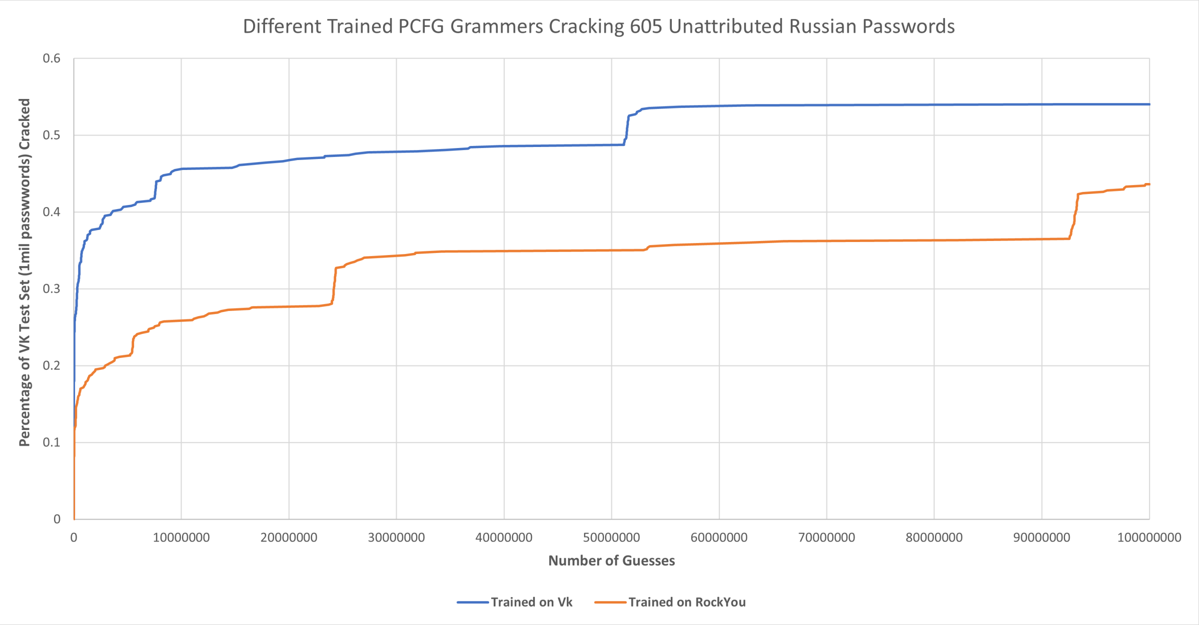


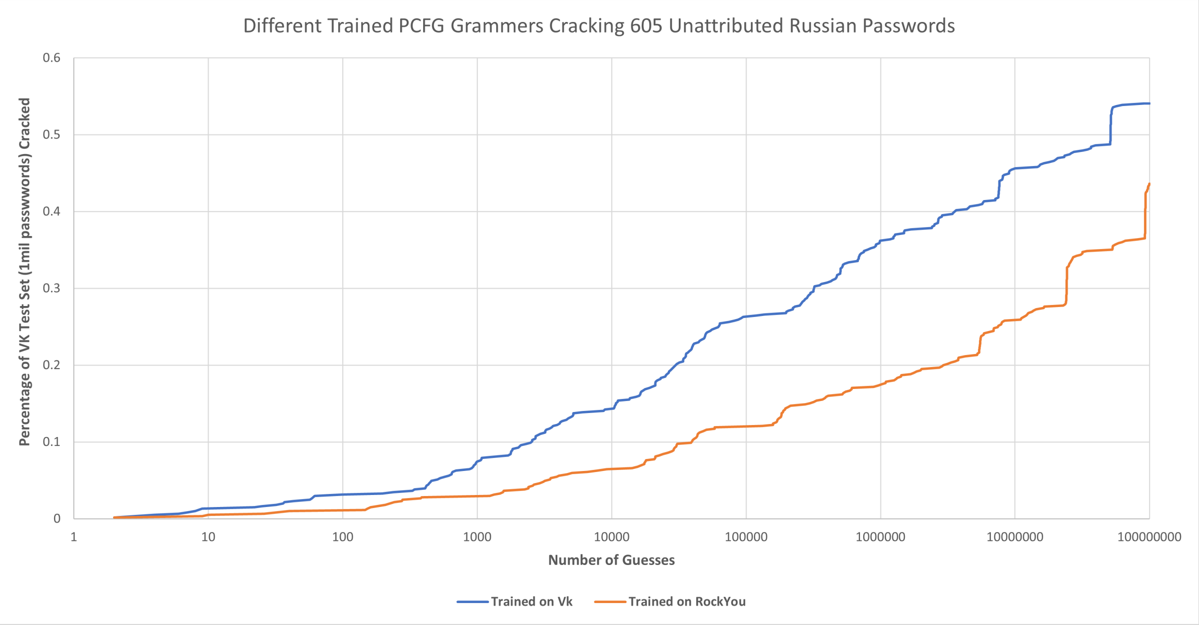
As you can see, the Russian ruleset significantly outperformed the Default RockYou ruleset when targeting passwords created by Russian speakers. Here is the same graph on a logarithmic scale to better look at how it performed earlier in the cracking session.



It's still debatable if password distributions follow Zipf's law (https://en.wikipedia.org/wiki/Zipf%27s_law). Zipf's law is useful though as a general rule of thumb when analyzing the success, (and possible optimization), of password guessing strategies. For example, if Zipf's law held true, then an indicator that your password guesser is optimized correctly is if the line in a logarithmic graph is relatively straight, (and the steeper the incline the better). Looking at these results, you can see where the Russian ruleset for the PCFG guesser starts to perform better after 1000(ish) guesses. This implies that the Russian ruleset still isn't optimized for guesses early on. In addition, the overall results even using the Russian ruleset only cracked 44% of the test set. This is significantly less than I'd expected, as I was hopeful based on the target demographic and password policy that it would crack around 60%. Even considering 2% of the passwords in the test set were "blank" and thus won't be cracked, this is a mystery to me and something I plan on digging into more later. But considering it's been two years since my last release of the PCFG toolset I didn't want this analysis to hold publishing a new version.

To validate these findings, the next step was to test on a different set of Russian passwords. For this test, I ran an attack against 605 unattributed Russian passwords I had obtained using the same settings as I had done against the VK test set. The results are below:





This time, the results were better, but still not where I'd like them to be. It did validate the earlier test, as the Russian ruleset continued to outperform the RockYou ruleset. As with targeting the VK set though, you can see on the logarithmic graph that the PCFG didn't start behaving as expected until around 1000(ish) guesses again.

CHAPTER

SIX

ACADEMIC PAPERS:

Original 2009 IEEE Security and Privacy paper on PCFGs for password cracking: https://ieeexplore.ieee.org/abstract/document/5207658

My 2010 dissertation which describes several advanced features such as the updated “next” algorithm: https://diginole.lib.fsu.edu/islandora/object/fsu%3A175769

Bill Glodek’s Thesis which detailed an earlier version of this tool. “Using a Specialized Grammar to Generate Probable Passwords” Authors: Bill Glodek: https://diginole.lib.fsu.edu/islandora/object/fsu:182401

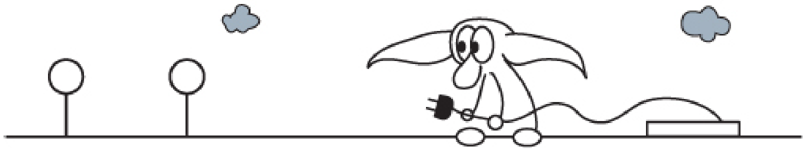
Comparative Analysis of Three Language Spheres: Are Linguistic and Cultural Differences Reflected in Password Selection Habits? Authors: Keika Mori, Takuya Watanabe, Yunao Zhou, Ayako Akiyama Hasegawa, Mitsuaki Akiyama and Tatsuya Mori: https://nsl.cs.waseda.ac.jp/wp-content/uploads/2019/07/EuroUSEC19-mori.pdf

OMEN: Faster Password Guessing Using an Ordered Markov Enumerator. Authors: Markus Duemut, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, Abdelberi Chaabane: https://hal.archives-ouvertes.fr/hal-01112124/file/omen.pdf

CHAPTER

SEVEN

DESIGN PRINCIPLES:



I felt it may be helpful to lay out some design principles for this particular PCFG implementation. None of these are hard rules, but they will influence any merge requests I accept. I also hope they explain certain architectural choices I've made when developing this code. Note, these policies only apply to this particular PCFG implementation. For example in the C compiled version of the PCFG is by its very nature going to be harder to install and run than the Python version in this repository.

1. The code should be easy to install and run

- A good number of the people who use this code-base are researchers and students new to security research. I'm convinced one of the reasons why this work has been able to have the impact it has is that researchers can download it and use it within minutes. Let's face it, security research is hard and most tools out there have a steep learning curve. Making something easy for people is often worth a lot more than a new feature provided by a hard to install package.
- As a corollary to this point, I am extremely resistant to include **numpy** as a required library. The versioning issues numpy has, along with the platform specific compiler needed makes it much more difficult to create portable and easy to install tools.
- I'm also resistant to adding any other additional required Python packages to this code-base. If there is a good reason and it's not numpy I'm open to hearing about adding new features that require additional libraries but will need a pretty good justification to accept a merge request. I will also try to ensure that the codebase can run without that package/feature. An example of how an optional library is currently treated in the codebase is the **chardet** library. chardet is pretty common and installed in most Python environments, but if people don't have it they can still train rulesets. They just have to manually detect and specify the encoding used in the training set themselves.
- This toolset should work on both Windows and most Linux variants. So for example, if you need to reference a file path, make sure you use *os.path.join()* vs. hardcoding in an OS specific backslash or forward-slash.

2. Whenever possible, the program should learn from the training set vs. having regexs or strings pre-defined

- This is a rule which I've broken a lot. For example the "Context Sensitive" X replacements are all pre-defined. Also things like e-mail and website TLDs are hard-coded in. Hence the weaselly word "whenever possible"
- An example of this approach in practice is how multi-word detection is handled. There is no predefined list of words to identify what is a multi-word and what is not. It learns all the base words during training.

- Learning from the training set is the desired approach because it makes the tools more language agnostic, and more importantly, pre-defined lists are really hard to maintain and often have frequent gaps. Also there are hidden benefits. For example, people often put words such as "Fall" or "June" at the end of passwords where there is a mandatory password change policy in place. The PCFG currently doesn't have any logic tailored to this specific mangling technique, but it gets it partially for "free" with the way it currently learns multi-words from the training set.

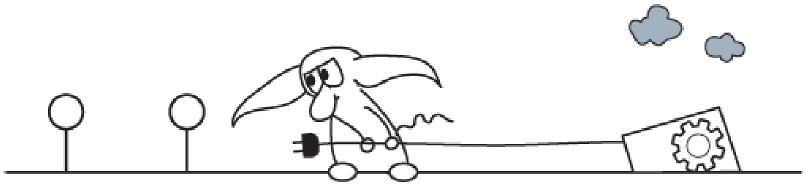
3. When possible, there should be a clear delimitation between the training phase and the running phase of tools in this repo

- This is another way of saying when training, save the output and then process that output with other tools when you want to use it.
- The reasoning behind this is that I've found it extremely useful to have the training set available to manually tweak or to make available for other tools and uses.

CHAPTER

EIGHT

PCFG “NEXT” FUNCTION DESIGN:



This section of the Developer’s Guide is geared for researchers and developers and focuses on the “Next” algorithm which is the core algorithm behind the PCFG guesser. In a nutshell, the “Next” algorithm determines what the next guess to generate should be for a given PCFG. The current Next algorithm in the PCFG is tailored to generating guesses in probability order. This means it’ll start by generating the most probable guess, followed by the second most probable guess, and so on. That is great for a password cracking attack, but it has some significant performance downsides. Which is another way of saying it is slow and requires increasing memory the longer it is run. Now not all Next algorithms have to follow this path, which is why this is a good area for improvements. For example a limit based Next algorithm that generates guesses in “mostly” probability order might have much better performance characteristics.

8.1 Definitions:

1. **PCFG**: Probabilistic Context Free Grammar
2. **Probabilistic**: Every transition has a probability associated with it
3. **Context-Free**: Individual transitions do not impact each other

- Normally this is desired since in passwords the base word is often independent of the other mangling rules.
  - E.g. ‘Monkey’ + ‘123’
- Sometimes you want to add context though. For example, create a password guess exactly 7 characters long. There are ways to design a PCFG to model these requirements and add some “context” if desired..

4. **Grammar**: This is a model. Much like how a sentence can be constructed, so can a password. The grammar is the model that is being used.
5. **Transition**: Replacing one value in a PCFG generated string with another while following the rules of the grammar

- An example of that may be taking the PCFG value “Password” + D2, and rewriting it as “Password25”. In this case the D2 transitions to the number “25”.

6. **Terminal**: A final guess/string generated by a PCFG. Once a terminal is created it is ready to be sent to whatever is using the program.

- An example terminal might be 'Password123' or 'Super!Secret182'

7. **Pre-Terminal**: A PCFG value where there still remain transitions to take

- Going back to the previous example "Password" + D2 would be classified as a pre-terminal. Likewise A8D2 would also be an earlier pre-terminal, with it represented as an 8 letter alpha word, followed by two digits.

8. **S**: This is the Starting pre-terminal. All transitions in the PCFG start from S.
9. **Parse Tree**: A record of all the transitions required to move from **S** to a given terminal.

8.2 Assumptions:

1. The transitions in the grammar are non-ambiguous

- By this, no transitions share the same replacement

  -E.g: Not allowed: Trans1 -> ReplacementA; Trans2 -> ReplacementA

- Disclaimer: In practice, most password based grammars are ambiguous due to features like representing keyboard walks. For the purposes of discussing the "Next" algorithm though, we're going to ignore those.

2. The grammar does not contain recursion

- There are no loops
  - E.g: Not allowed: Trans1 -> Trans2 -> Trans1;
- Disclaimer: Support for recursion was actually included in the PCFGv3 implementation. It adds a lot of complexity, and recursion was never made use of in the password grammar, so for this guide we're going to ignore it.

3. When multiple transitions occur before a final terminal (excluding the first transition from 'S'), the following transition type occurs universally to the previous transition

- This is a huge restriction, and made to simplify the implementation of the "Next" function
- Think case-mangling. S -> ['cat', dog', 'rat']. Now you want to apply case mangling to those alpha strings. It is easier to represent that case mangling as a mask vs. having to have a separate transition for each individual word.
  - E.g. S -> ['cat', 'dog', 'rat'] x ['LLL', 'ULL', 'UUU']
  - This would normally be represented in a more traditional PCFG as 'cat' -> ['cat', 'Cat', 'CAT']

8.3 General Documentation Conventions:

1. A PCFG transition will be represented by a capital letter

- Example: 'A'
- Note, this is different from how base structures are handled in the current grammar where the transitions are usually represented a capital letter followed by a number indicating how long the transition's word/digit/string is. I'm simplifying that out for this discussion about "Next" functions so this number doesn't get confused for the probability order number which is represented as a subscript.

2. Transitions are followed by by a subscript of the current replacement ordered by probability

- Example: 'A_1' for the most probable, 'A_2' for the second most probable, and so on.

3. Based on Assumption #3, transitions are applied universally so they can be written expanded out

- Example: For case mangling 'A_1B_2' could represent the most common word with the second most common case mangling mask applied to it.

8.4 Desired Goals of the "Next" Algorithm:

1. Only generate a parse tree once

- This is another way of saying avoid making duplicate guesses

2. Generate parse trees in probability order

- Start by generating the most probable terminal/guess first, then the second most probable one, and so on.

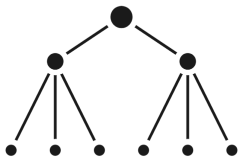
3. Minimize memory and running time requirements

Note: Achieving all three goals at once remains an open problem.

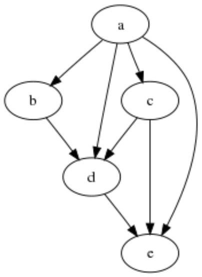
8.5 Current Approaches for Designing a "Next" Algorithm:

At a high level, all the current approaches for designing a Next function for a PCFG based password cracker rely upon turning the PCFG into a tree search problem.

- Once you do this, you can start to apply traditional search techniques such as Depth First Search (DFS), Breadth First Search (BFS), etc.



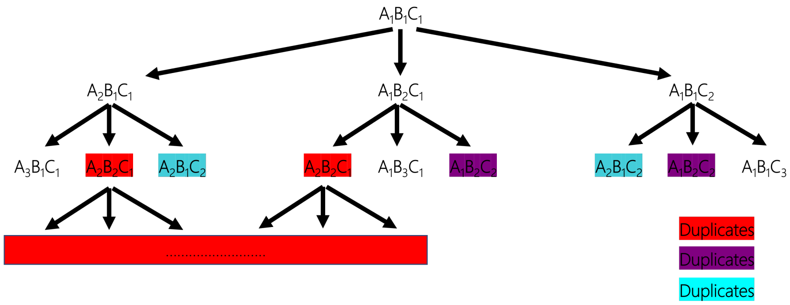
- This may seem like a fairly straightforward representation but it probably took me over a year of my PhD research to think of this optimization.
- Now technically a PCFG is not a Tree. By default it is better represented as a Directed Acyclic Graph (DAG). If you squint these may look a lot like a tree but way more complicated.



- "More complicated" is usually not a good thing when designing algorithms. Therefore steps need to be taken to simplify the PCFG DAG into a Tree.
- If I had to sum up my PhD defense in the most boring way possible, it was that I took a Probability DAG, created an algorithm to convert it into a Probability Tree, and then used a priority queue to perform a BFS walk of the tree which generated all the terminals in probability order. I then called this a password cracker.
- What this means is that there really are two steps in the "Next" algorithm. 1) Converting a DAG into a Tree, 2) Performing BFS of the tree. Both steps has lots of room to be optimized. For example, if you don't care about generating guesses in true probability order, there are a number of Depth First Search (DFS) optimizations you can use instead of BFS.

8.5.1 Challenges Traversing a PCFG as a Tree:

Lets consider a PCFG grammar as a Tree with the root node being $A_1B_1C_1$. I'm ignoring the **S** pre-terminal for the sake of convenience, but you can mentally put it above the root node if you want to. If we continue to build this grammar out as a tree with each lower probability transition being a leaf, the following will usually occur:



As you can see, many leafs are effectively duplicates of each other. This is why you usually see PCFGs represented as a DAG instead of a Tree. This also causes problems for our "Next" function since duplicates are something we want to avoid if possible. Therefore most of the discussion you'll see about converting a DAG to a Tree will focus on how to "trim" branches of the tree above to eliminate duplicate guesses.

8.5.2 PCFG v0 Next Algorithm:

The very first approach we took when investigating if PCFGs could be used to represent human generated passwords was to generate all possible passwords up to a given probability, save them to disk, sort them in probability order, and then print them. This effort was lead by Bill Glodek and was the basis for his Thesis. The actual algorithm is described in section 3.4.2 of his Thesis, and the code is in the Appendix of his Thesis, but let me try and simplify the description for this guide:

- Since the guesses are sorted after the fact and do not need to be generated in order, branches can be trimmed off the DAG so each variable can be incremented from left to right. So for example, guesses can be generated as:
  - $A_1B_1C_1$
  - $A_2B_1C_1$
  - $A_3B_1C_1$
  - $A_1B_2C_1$
  - $A_2B_2C_1$
  - $A_3B_2C_1$

  - $A_1B_3C_1$
  - …

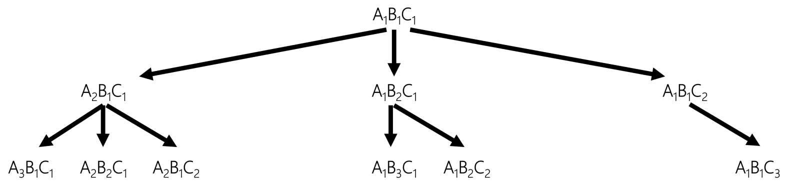
- Disclaimer: I'm not entirely sure the above is exactly the way the code looped through the generation process as I'd need to look at the code in more detail to verify the algorithm. But the general idea should hold up.
- While we hadn't defined it as such at the time, the above idea uses the idea of a "pivot" point to prune branches and eliminate duplicate guess generation. This was expanded out in the Version 1 "Next" algorithm for the PCFG.

The upside of this approach is that it is simple. Also, once the initial wordlist is generated/sorted it can be used by any other password cracking tool. More importantly, the cost of generating this wordlist is a one time cost! Therefore you can think of it as a time/memory trade-off as you are only doing the computations once and using a ton of disk space to cache the results for future cracking attacks.

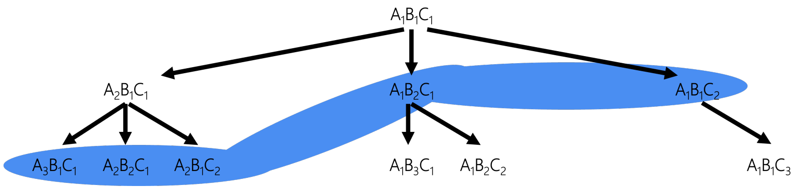
The challenge is the memory requirements are onerous as you need to save then sort every single guess before you even start a password cracking attack. If you don't mind terabytes of wordlists, and spending hours/days generating the initial wordlist, this approach is still totally viable.

8.5.3 PCFG Next Algorithm v1:

The PCFG "Next" algorithm version 1 is the approach described in the original PCFG S&P paper. It rests on the idea of assigning a pivot value to each node to prune branches and convert a DAG to a Tree. The pivot value specifies what children/leafs can be created by a particular node. A child can only be created if the transition position is equal to or greater than the pivot value. So for example, a root node of $A_1B_1C_1$ would be assigned a pivot value of 0, which means it can create children for all of its transitions. Its child $A_2B_1C_1$ would likewise inherit the pivot value of 0 and could create all of its own children as well. On the other hand, the child $A_1B_2C_1$ would be assigned a pivot value of 1, since it was created by incriminating the second transition. Therefore it could only create children for its B and C transition. Therefore it creates trees such as the one below:



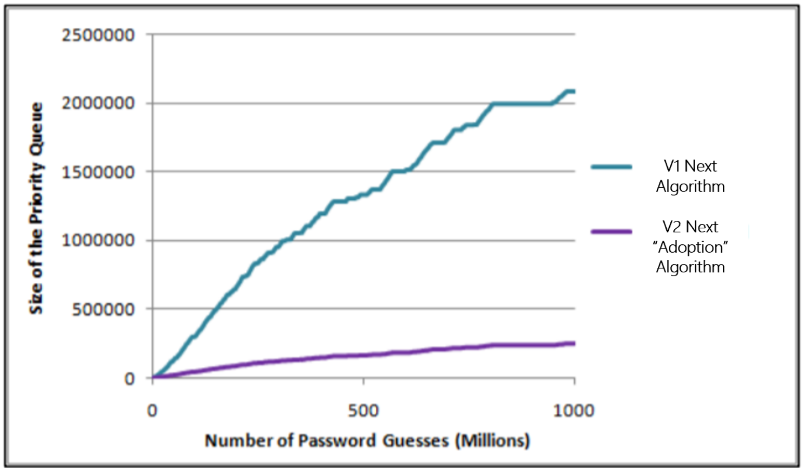
Once the PCFG has been converted to a Tree, the next step is to use it to generate guesses in probability order. The easiest way to do that with a priority queue. The priority queue is initialized with the root node in it. You then pop the most probable node off the queue, generate guesses from it, and then push all the children from that node back into the queue. This continues until you've cracked the password, or you run out of nodes to traverse. Therefore the priority queue will hold all of the nodes of the current breadth first search of the Tree. This can be seen in the following diagram.



The reason why this is the Version 1 of the "Next" algorithm and not just the "Next" algorithm can be summed up by the term: "memory usage". To put it simply, a lot of very low probability nodes get pushed into the priority queue which causes it to expand very fast because the tree is so heavily weighted to the leftmost nodes. Looking back to the previous picture of the Tree, you can see a node on the left will have significantly more children than the right side. This means that when one of those nodes gets popped and its children get pushed onto the priority queue this may include a lot of very low probability children that will hang out in the priority queue for a significant amount of time. Think of those nodes as a trust fund children that are able to show up simply because their parents had a high probability, and they just hangs out in the priority queue not contributing anything and taking up resources.

8.5.4 PCFG Next Algorithm v2 (Adoption Algorithm):

The second version of the "Next" algorithm is the Adoption algorithm. I originally called the "Deadbeat Dad" algorithm, but I'm regretting the negative connotations so I'm renaming it. The key idea behind this approach is that a node can have multiple different parents (that old DAG vs. Tree issue), so a child should only be pushed into the priority queue if its least most probable parent was just popped from the queue. This relies on the assumption that a child's probability can't be greater than its parents. In the current PCFG code, this approach is implemented by generating all the children for any node popped from the queue, and then walking back up the Tree/DAG and generating all the parents for those children. If a child has a lower probability parent than the parent that was just popped, it is left for that lower probability parent to "adopt" that child later. If the recently popped parent is the lowest probability parent for a child node though, that child is then pushed into the priority queue. Ties in parent probability are resolved by the arbitrary but deterministic approach of selecting the rightmost node to be the parent. Admittedly this approach requires a lot more computation than the Version 1 "Next" algorithm. The memory savings and correspondingly faster priority queue pushes make the Version 2 approach better algorithm for most use-cases though. This can be seen in the following graph.



8.6 Suggested Research Areas and Open Questions:

The current Version 2 “Adoption” algorithm is computationally expensive and even though its memory usage is better than the Version 1 “Next” algorithm, it still grows quite large for longer cracking sessions. Therefore there is still a lot of room for improvement when it comes to weaponizing a PCFG for use in password cracking attacks.

8.6.1 Improving the Breadth First Search (Priority Queue):

- There’s been a lot of research into better BFS algorithms in the last ten years since the v2 “next” algorithm was written
- There’s some promising approaches beyond using a PQ
- Lots of improvements are possible, but memory management is still a running concern for alternate BFS implementations that I’ve looked at
- Long story short, this is a problem that I keep expecting to have been solved in other contexts, either in general tree BFS, DAG BFS, or PCFG BFS
- Memory management is a killer. Most “AI” solvers I’ve looked at tend to use a combination of Depth First Search (DFS) and BFS to compensate
- If you come up with a better solution, I will be **very** interested in it

8.6.2 Other Approaches:

- Dropping the “Probability Order” Requirement
  - If you no longer care about generating guesses in true probability order, there are multitude of **much** faster “Next” algorithms you can use.
  - Basically you are no longer using BFS. Other approaches can be using DFS which has very limited memory and computational requirements.
  - This can be seen in John the Ripper’s –Markov mode
- Dropping “Probability” Altogether
  - Rather than calculate the true probability of each PCFG terminal, you can quantize each transition and use addition + limits, much like OMEN and JtR’s –Markov algorithm.
  - Now you are doing addition of INTs, vs. multiplication of FLOATs, which speeds things up.
  - In fact, PCFGs can use the underlying algorithm in OMEN with very little modification. Instead of length, choose a base structure as a starting point. Since the CF in PCFG stands for “context free” this allows for even more optimizations compared to other Markov based approaches

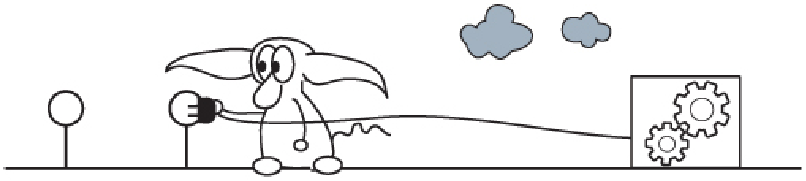
8.6.3 Summary:

- There’s a lot of ways to generate guesses with Context Free Grammars
- The speed/memory trade-offs occur when specifying requirements for generating probable guesses first
- Most other current guessing algorithms can be adapted to use Context Free Grammars if desired
- Feel free to think outside the box. Don’t let current implementations that utilize pqueues and large memory requirements limit your thinking!

CHAPTER

NINE

MODIFYING/EXTENDING THE PCFG TRAINER



9.1 Overview:

The PCFG trainer is the core of the PCFG toolset. The trainer determines what the grammar looks like, and how subsequent tools and attacks model human generated password creation strategies. Areas of innovation include:

- New mangling rules can be added to the trainer. For example, l33t sp33k substitution could be implemented.
- Probability smoothing could be applied to reduce the amount of transitions. For example, if a transition A is found 10002 times in the training set, and transition B is found 10003 times in the training set, the trainer could smooth their probabilities so they occur with the same frequency in the generated grammar. This can significantly speed up cracking attacks using the grammar
- Coverage can also be determined by the trainer. Aka how often do you try transitions that were not seen in the training set. For example, if the number '18632' is not in the training set, you may still want your grammar to generate it at a low probability.
- Finally existing mangling rule detection can be improved. For example the keyboard walk detection currently has a lot of false positives, while at the same time it only detects walks without gaps. Therefore it would detect '1qaz' as a keyboard walk, but it would not detect '1z2x3c'

9.2 Unit Testing:

There currently are a number of unittests built for the PCFG trainer. This is a good way to ensure that when code is modified that the rest of the functionality works as expected. As a request on my end, if you submit a merge request that modifies the PCFG trainer code, that you run the unittests before hand, and also add or modify unittests as needed. The number of bugs I've caught with these unit tests is significant enough that I'm super glad I took the time to write them.

To run the unittests:

1. From the top level pcfg_cracker directory run: *python3 -m unittest*

- Note: Change "python3" to whatever name you use for your python 3 environment

2. Some of the tests are "noisy" as I haven't figured out how to suppress their outputs to stdout, but the last two lines you should see 70+ unit tests ran successfully with an "OK" in the last line. If you encounter any errors that implies some of the training functionality has become broken.
3. If you want to add or modify any unit tests you can find them in the: pcfg_crackerlib_trainerunit_tests directory

- All current Unit Tests are written with the Python unittest module.

9.3 Understanding the Current Training Process:

9.3.1 Looping Through The Input List

When thinking about where to start when describing the current PCFG training process, the first thing that pops into my head is talking about how it loops through the training set multiple times. Each pass through the training set allows the trainer to learn more which then help inform future passes and training techniques. Below is a high level description of what it does in each pass:

- (**Pre-training Phase**):
  - If the –encoding flag is not passed to the trainer, the first thing it does is read through a subset of the training set and attempt to auto-detect the encoding of the training file by using the chardet python package. It then interprets all subsequent passes through the training set using this character encoding
- **First Pass**: This pass is meant to learn all the base features of the training set that more traditional "mangling" detection rules can utilize. Therefore this run happens prior to any PCFG grammar transitions being created/learned/defined.
  - The trainer identifies individual words to use in multi-word detection
  - The trainer identifies the alphabet to use for constructing Markov chains
  - Duplicate password detection is performed. Duplicates a good, so if no duplicates are found in the training set it can print an warning to the user of this tool. In the future, this may also be used in some of the probability smoothing functionality.
- **Second Pass**: This pass is where the core grammar for the PCFG is created from the training set
  - All of the base structures and non-Markov transitions are identified and assigned a probability in this pass. For example it learns the base structure masks, such as "A3D2", and probabilities such as "A3" -> "cat" at 20%.
  - The trainer also learns NGrams for Markov based transitions. It is currently using the OMEN algorithm for this.
- **Third Pass**: In this pass it re-runs through the training list to perform any post-processing that may be necessary
  - The trainer learns what Markov (OMEN) probabilities the training passwords would be created at. This is an OMEN specific task to assign probabilities to each of the different OMEN levels
- (**Final Summary**): Once all the passes through the training set are performed the following happens
  - The PCFG grammar is finalized and written to disk
  - "Interesting" statistics are printed to the screen. Ultimately I'd like to expand these so that the trainer could be used as an alternative to PACK.

9.3.2 Learning Mangling Rules

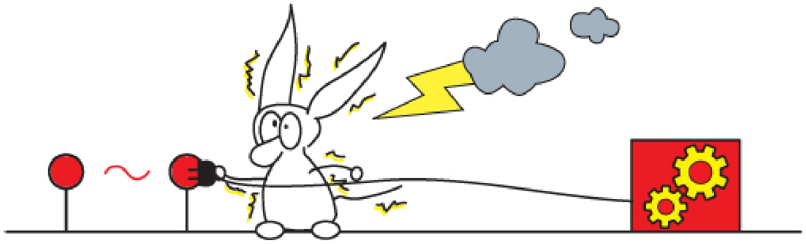
The “pcfg_parser” is the top level object/class that takes as input each individual word from the training set and then identifies the mangling rules associated with it. During this stage, each input password is parsed independently from other training passwords, which is why all the pre-processing needs to happen before this.

- The current PCFG grammar’s root nodes are centered around the idea of “base structures” These are high level definitions of what type of transition to take for each chunk of the password. Initially a “section_list” that represents this base structure is initialized to be undefined for the length of the training password. What that means is that every character in the training password is initialized as “undefined” as the trainer does not know how it was generated.
- The pcfg_parser performs mangling detection in a pre-defined order, so earlier mangling detection techniques have priority over later techniques. This is important because different mangling detection techniques can identify the same strings in the training password differently.
  - For example, the keyboard walk detection might identify the string ‘1qaz’ as a keyboard walk
  - By default the Alpha and Digit detection would classify ‘1qaz’ as a D1A3 (One digit followed by a three letter word)
  - Therefore its important that the keyboard walk detection occurs before the Alpha and Digit detection and takes precedence over them.
- As a string/sub-section of the training input is classified, it updates the “section_list” with that classification. Therefore subsequent mangling detection techniques are only run against the undefined sub-sections.
  - One key point is that even though the current mangling detection techniques skip portion of the “section_list” that are defined, that doesn’t mean that **your** mangling techniques need to skip or ignore those. The data is available so if you can make use of it please don’t hesitate to do so.
- At the end of the processing, “Other” string detection is run as a clean-up/default option. Any undefined “section_list” portions are classified as “Other”.

CHAPTER

TEN

MODIFYING/EXTENDING THE PCFG GUESSER



10.1 Overview:

The PCFG Guesser is a tool meant to support password cracking attacks. At a high level, it reads in a previously trained PCFG grammar, and then outputs password guesses based on that grammar to stdout for use in an actual password cracking tool. Therefore as the name implies this tool only generates "guesses". It doesn't actually crack any passwords on its own.

The PCFG Guesser is currently aggressively single threaded. The only part where threading is currently used is to check for the user pressing a key so that it can print out the status/help messages. This focus on not parallelizing the guesser may seem odd considering that this tool is CPU bound and the slowness of it hurts the viability of using PCFGs in a password cracking session. I wish I had a deep technical reason that shows great insight to respond to this, but I don't. The reason why it's single threaded is every time I attempted to add parallization, it actually slowed the program down. If you can improve the performance and succeed where I failed, I'm all about that.

10.2 Generation of Pre-Terminals:

- One thing that on retrospect may cause a bit of confusion, is that the `pqueue` referenced in `cracking_session.py` is NOT a traditional Priority Queue, but is instead a `PcfgQueue`. Now, the PcfgQueue does contain a priority queue itself, and operates by popping the most probable pre-terminal structure off of it. But it also contains the current "Next" algorithm implementation that converts the PCFG DAG into a Tree structure. To learn more about this please refer to the "PCFG Next Function Design" section of this Developer's Guide. Suffice to say, a lot of stuff is happening in the background when you call `pqueue.pop()`.
- Now the next question you may be asking yourself is "What exactly is a pre-terminal". Technically that refers to a PCFG parse tree where all of the transitions have not been completed as terminals. When it comes to the PCFG Guesser though, this takes on a special meaning. In this context, it means that all of the remaining transitions lead to leaf nodes of the same probability. Therefore all terminals/guesses generated by this pre-terminal will have the exact same probability. This is really important since guesses can now be generated from them without having to do any more probability calculations. Think of these pre-terminals as very fine-grained wordlist rules

in your more traditional password cracking programs, with the caveat that these pre-terminals not only define the traditional mangling rules to use, but also the “wordlist” to operate upon.

- Once a pre-terminal is defined, the PCFG guesser can then generate password guesses from it and then output those guesses to stdout. This looks much like you would find in a more traditional password cracker. This is also a horribly slow step thanks to Python’s abysmal I/O latency and overhead. Seriously, every time I profile my code I’m horrified by how poor Python is at outputting guesses in a timely fashion. If I didn’t love writing Python code, and if Python wasn’t so easy to run, I would never recommend Python scripts to be used to generate the input for password crackers.

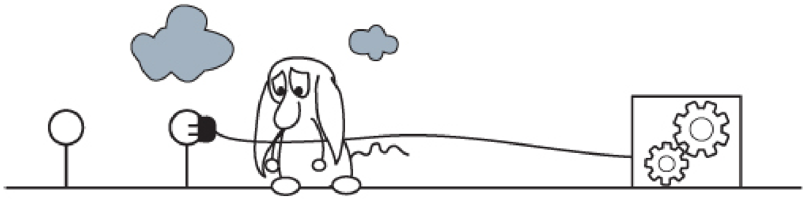
10.3 Future Areas of Research:

- Python is slow, C is much faster. You can find a C version of the PCFG guesser here: https://github.com/lakiw/compiled-pcfg
  - Note: My C coding skills leave a lot to be desired, and quite honestly I don’t like coding in C. Therefore the compiled PCFG does not support all of the features of the Python version, and it is significantly more unstable.
  - That being said, it’s wicked faster than the Python version!
  - Improving the C version of the guesser and potentially integrating it directly into other password cracking programs remains an item eternally on my to-do list. If you have any interest in doing this you have my full blessing.
- Currently the PCFG guesser outputs guesses in probability order. I’d really like to add support for generating guesses based on a probability limit instead. You can see more discussion about this in the “PCFG Next Function Design”, but at a high level, being able to generate guesses via a limit would potentially allow parallelizing the guess generation process, and might even be something that could take advantage of GPU processing.
- Another neat feature would be to generate word mangling rules for other password cracking programs vs. generating guesses. This would remove all the performance issues with using PCFGs in a password cracking session. The biggest downside I can see is that the current PCFG does so well due to its ability to optimize the use of “words” in its wordlist. That would be difficult to translate over to more traditional password cracking rulesets. Aka the word “password” is really common so PCFG generated guesses contain it much more often than the word “zebra”. It’s hard to shoehorn that type of optimization into other password cracking rulesets though.

CHAPTER

ELEVEN

HONEYWORDS:



11.1 Honeywords Overview

Definition: Honeywords are synthetic, (computer generated), passwords that attempt to look like passwords created by humans

Why would you want honeywords?

Good question! I'll admit the main reason I've worked on honeywords is they are essentially "free" research since a PCFG grammar that can generate a realistic looking synthetic password list also helps cracking passwords with a PCFG. To put it another way, a perfect honeyword generator would also be a very effective password cracker. Also, it gave me a chance to collaborate with Ron Rivest, and you just can't say no to that. The reason why I'm including honeywords in this Developer's Guide though is there are some use-cases where honeywords may help out other researchers:

- Honeywords can be used as part of an IDS to detect stolen credentials. Basically if you see a honeyword being used you can create an alert. For this use-case the honeyword generation really depends on how your IDS is configured and what it is listening for. If you are looking for lateral movement and have an entirely fake account set up, you will want the honeyword to be truly unique to avoid false positives. Which means using a PCFG may not be the best approach because it generates passwords that real users on your network might also select. There have been several papers though that suggest storing honeywords alongside the real credentials of users. In that case, if you control the hashing scheme, I **strongly** suggest generating honeywords via intentional hash collisions vs. trying to generate realistic looking fake passwords via a PCFG. The easiest way to create honeywords via collisions is to split up the password hash between the server and the IDS verifier. Therefore the attacker will generate collisions regardless of their guessing strategy, and the verifier can differentiate between a legitimate login, an attacker generated collision, and a user simply typing their password in wrong with a very low false positive rate. The simple fact is that using a PCFG adds a lot of complexity to this use case, and during testing that I ran in collaboration with RSA Labs we highlighted that a knowledgeable attacker could often differentiate between the real password and a PCFG generated honeyword. In short, generating fake passwords that look real is a hard problem to solve!
- PCFG generated honeywords can be used to programatically generate fake passwords for honeypot systems and networks. **This is where honeywords shine**. Unless you want to create 1k fake active directory passwords by hand being able to script up and automatically generate realistic passwords is nice. Note, one area of open

research is to generate multiple passwords for the same user that share characteristics. This seems to be an area where the context-free nature of PCFGs could be leveraged for this. Aka words could be swapped out but the same mangling techniques could be used, or vise-versa the same words could be used but dates could change, or other features such as "summer22" could be added to the end of them. Basically, if you are interested in adversarial engagement and deception, honeywords can make your life easier.

- PCFGs can be used to quickly creating passwords for password cracking competitions. This is a tricky one since PCFGs **will leak sensitive user data**. I can't stress that enough, so please DO NOT use a PCFG trained on a non-public password list to create passwords for a capture the flag type event. That warning aside, I've participated in a lot of password cracking competitions, and one big artificiality most of them have is that the passwords you are cracking do not resemble real-life passwords. Therefore much of the skill in these competitions is identifying how the organizers created the passwords and developing rules on the fly to target the competition specific generation strategies. A honeyword list generated by PCFGs on the other hand would more closely resemble real life passwords. Also, it's really easy to generate PCFG based honeywords which makes the life of a CTF organizer easier, which I have to say with my experience helping to run the BioHacking Village CTF at Defcon, is a big plus. There's always more fun challenges to develop and only so many "volunteer hours" before the competition.

11.2 Generating Honeywords with a PCFG

First thing first, honeyword generation is currently not supported by the Version 4.x branch of the PCFG toolset. Instead you'll need to download the legacy Version 3 PCFG toolset available here: https://github.com/lakiw/legacy-pcfg

If you are wondering why honeywords aren't part of the 4.x branch, as some background I had collaborated with RSA Labs in 2014. This resulted in an unpublished paper about honeywords that fell through when Dell bought EMC (which owned RSA), and lets just say lawyers got involved. That's still a sensitive subject but enough years have passed that one day I might add that functionality back in.

Using the program in the "honeywords" directory, the following command will generate 100 honeywords from a saved grammar:

```
python3 honeyword_gen.py -r Default -n 100
```

Note: You will also need to generate a training set using the legacy pcfg_trainer.py. This will not work with grammars generated with the Version 4.x trainers.

Generated Honeywords:

```
671149941
mariposa
24680076manguera
m1a06
abbieh
 birthday
ruben1
camille
heather
eduardo30
rhadzar123456
BABY27
1012424
ashtynn
755555
calzon16
fkggcewi
```

(continues on next page)

(continued from previous page)

```
2681996
tiger
britney
2104lisbeth
jenicitalindalas
arnold18
04882120701
0847438080
parts28
puypuy
m091989
777death
ladida
lokita36
password
lol123
RAFAEL
123456789
143733
borris85
free15
latinthugz
rockme
carito
weaver95
072590
Bubsie
junahbell91
waylie91
117198
wellington
123456
shomocka2
chester01
sodoff2008
Crystal
sexydillon
1234567890
hepertyor
smineo
arli1994
9726369292
faggioni
jan7784hf
berrios66
dughug
malachi
fuckyou1
romania777
katvin
tlichoplayer
suksan
```

(continues on next page)

(continued from previous page)

```
7catset
raynak23
trebmal
biteme
pelota
thailand
davejelek
rufus.honda
123456
bubblegum
photos
bunnymuu
1edwardluvranpha
123456
Jamaal
4737613
gita11
k9788201
171006
carters
annie4114
chocolate
d1life
LOVEHINA76
cornell2
eliane8
tigers11
211204
amotebebe
angelito
yahomail
```

11.3 Developer's Guidence for Honeywords:

The good news when generating honeywords is you don't really need to worry about a "Next" algorithm that the PCFG cracker uses. Instead you can use a weighted traversal of the PCFG DAG to generate honeywords according to the probability of the terminals. This means you'll often generate many weak/common honewords such as '123456' or 'password123', but you will still also generate very unlikely passwords, just at a lower frequency.

This approach currently goes as follows:

1. Start at the root node of the PCFG. In the current implementation this is the "base structure".
2. Generate a random float between 0 and 1.
3. Select the base structure that appears in that probability range the random float specified. There are different approaches you can take for identify what the range is. For example:

- You could assign a start value/stop value for each transition based on it's probability. For example, if you had 4 terminals, with A:40%, B:30%, D:20%, E:10%, you could assign the ranges: A[0.0-0.4] B[0.4-0.7], C[0.7-0.89]. D[0.9-1.0]. Yes there are overlaps in the edges, but you can give preference to one of them however you desire.

4. Generate another random float between 0 and 1 and repeat the process for the next transition. Continue until you have generate a terminal. This is your honeyword

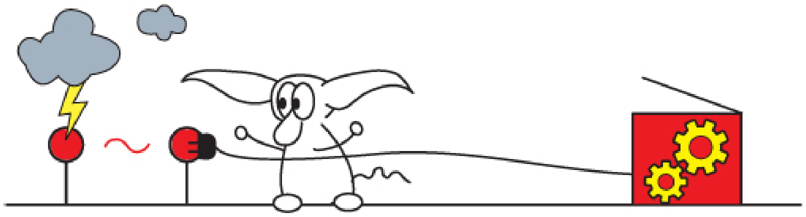
Things get a bit more complicated when generating honeyword Markov strings, (such as for the OMEN transition in the current PCFG). One easy approach is to simply skip OMEN generated terminals. If you want to keep a Markov based brute force generation available for your honeywords, that is an open problem that I haven't really sat down and solved.

Another open area of research is generating multiple honeywords for the same user. One approach might be to generate the first honeyword as normal, but keep track of the parse tree that generated it. When generating subsequent honeywords, you can then start the process by generating a random variable to pick the follow-up generation process. The previous honeyword could be re-used at a certain probability, a completely different honeyword could be generated at a certain probability, or the previous honeyword could be "mangled" at a certain probability. The "mangling" in this case would be selecting one or more different decisions in the previous honeyword's parse tree. For example a different base world could be selected, or a different digit string or capitalization approach could be taken.

CHAPTER

TWELVE

TRAINER CODE DOCUMENTATION:



12.1 trainer.py

Name: PCFG Trainer

> Training program that creates Probabilistic Context Free Grammars (PCFGs) from plaintext passwords
>
> Can also be used to generate statistical data and dictionaries for other cracking methods such as MASK attacks and OMEN

Contact Info: cweir@vt.edu

`trainer.main()`

> Main function, starts everything off
>
> Responsible for calling the command line parser, detecting the encoding of the training set, creating the initial folders and then kicking off the training via run_trainer()
>
> **Inputs:** None

Returns: None

`trainer.parse_command_line(program_info)`

Responsible for parsing the command line.

Note: This is a fairly standardized format that I use in many of my programs

Inputs: program_info: A dictionary that contains the default values of command line options. Results overwrite the default values and the dictionary is returned after this function is done.

Returns: True: If successfully

False: If a value error occurs

(Special: Program Exits): If Argparse is given the –help option

12.2 banner_info.py

Contains banner ascii art and displays for the training program

`lib_trainer.banner_info.print_banner()`

ASCII art for the banner

Putting it here to make it easier to change

12.3 base_structure.py

This file contains the functionality to create base structures

These are the first transistions in the PCFG. For example A5D2

`lib_trainer.base_structure.base_structure_creation(section_list)`

Parse out the base structure in a password

Returns: Returns two values, is_valid, base_structure

is_supported: If the base structure is currently supported by the default PCFG cracker. This is useful since things like e-mail addresses and websites will likely not have support until targeted cracking sessions support is added

base_structure: A string representing a base structures

12.4 calculate_probabilities.py

This file contains the logic to calculate probabilities for a given Counter

This also includes the logic to perform probability smoothing.

Note: Probability smoothing is currently not implimented

`lib_trainer.calculate_probabilities.apply_probability_smoothing(counter)`

Apply probability smoothing

This is a way to smooth probabilities between different items to help the actual password cracker making use of them.

For example, if ‘chair’ was seen 100000 times, and ‘table’ was seen 100001 times, it would be nice to treat them as the same probability to reduce the amount of work the pcfg cracker needs to perform

Currently this is a no-op / placeholder and does not actually do anything

Variables:

counter: A Python Counter object containing all the items to calculate probabilities for

`lib_trainer.calculate_probabilities.`**`calculate_probabilities`**`(`*counter*`)`

Calculated the probabiilty for items stored in a Python Counter

Variables:

counter: A Python Counter object containing all the items to calculate probabilities for

Returns:

prob_list: A Python List containing ('value','probability) pairs ordered from hightest probabilty to lowest probability

12.5 config_file.py

This file contains the functionality creating the config file for rulesets

Note, in the past I tried to make this generic so that when a new grammar rule was added to the trainer, no changes would have to be made to the config file generation code. This didn't work out as well as I wanted, so to save time and reduce complexity this config file code needs to be updated when the grammar generated by the trainer is modified

`lib_trainer.config_file.`**`add_alpha`**`(`*config*, *filenames*`)`

Creates the configuration for the Alpha Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`add_capitalization`**`(`*config*, *filenames*`)`

Creates the configuration for Capitalization Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`add_context_sensitive`**`(`*config*`)`

Creates the configuration for Conte(X)t Sensitive Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`add_dataset_details`**`(`*config*, *program_info*, *file_input*`)`

Adds the details from the training dataset to the config file

Variables: config: the config file being created

program_info: Info about the program and config info about the current run

file_input: Info from the input dataset after parsing `lib_trainer.config_file.`**`add_digits`**`(`*config*, *filenames*`)`

Creates the configuration for the Digit Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`add_keyboard`**`(`*config*, *filenames*`)`

Creates the configuration for Keyboard Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`add_other`**`(`*config*, *filenames*`)`

Creates the configuration for the Other Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`add_program_details`**`(`*config*, *program_info*`)`

Adds the program details to the config file

Variables:

config: the config file being created

program_info: Command line results, and info about the program such as version and author

`lib_trainer.config_file.`**`add_start`**`(`*config*`)`

Creates the configuration for the base structure

Note, if any new terminal/non-terminal replacements get added this will likely need to be updated.

For example, if you add a new replacement for "sports teams" being a "T"

Variables:

config: the config file being created

`lib_trainer.config_file.`**`add_years`**`(`*config*`)`

Creates the configuration for Year Replacements

Variables:

config: the config file being created

filenames: A list of filenames associated with this replacement

`lib_trainer.config_file.`**`create_config_file`**`(`*program_info*, *file_input*, *pcfg_parser*`)`

Creates the config file and returns it

Variables:

program_info: Info about the program and config info about the current run

file_input: Contains info about the passwords just parsed

pcfg_parser: The pcfg parser class that was trained on passwords This is used for extracting file names

Returns:

config: The Python ConfigParser configuration to save for this rulesets

`lib_trainer.config_file.`**`create_filename_list`**`(`*input_dictionary*`)`

Takes as input a Python Dictionary and returns a list of filenames from the keys

Variables:

input_dictionary: The Python dictionary to create filenames from

Returns:

filenames: A Python List of all the filenames as strings

`lib_trainer.config_file.`**`save_config_file`**`(`*directory_name*, *program_info*, *file_input*, *pcfg_parser*`)`

Creates the config file and then saves it to disk

Variables:

directory_name: The full name of the rules directory

program_info: Info about the program and config info about the current run

file_input: Contains info about the passwords just parsed

pcfg_parser: The pcfg parser class that was trained on passwords This is used for extracting file names

Returns:

True: If everything worked correctly

False: If any errors were encountered

12.6 pcfg_password_parser.py

This file contains the functionality to parse raw passwords for PCFGs

The PCFGPasswordParser class is designed to be instantiated once and then process one password at at time that is sent to it

`class` `lib_trainer.pcfg_password_parser.`**`PCFGPasswordParser`**`(`*multiword_detector*`)`

Responsible for parsing passwords to train a PCFG grammar

This is designed to be created once, and then to process one password at a time.

`parse``(`*password*`)`

Main function called to parse an individual password

Inputs: password: (String) The password to parse

Returns: True: If everything worked correctly

False: If there was a problem parsing the password

12.7 prince_metrics.py

This file contains logic for extracting data that will be useful for generating wordlists to use in an optimized PRINCE guessing session

Note: While PRINCE is not a PCFG based guess generator, the data generated by this trainer can be useful for supporting other attack modes

Note 2: This is mostly a placeholder as all it does now is update the count_prince Counter with each parsed section. The plan is eventually to add additional logic in here that may be useful for optimizing a PRINCE dictionary. For example, certain objects may be weighted to make them more or less likely.

`lib_trainer.prince_metrics.`**`prince_evaluation`**`(`*count_prince*, *section_list*`)`

Save any statistics that would be useful to generating PRINCE wordlists

Note: This should be called after all the individual sections of an input password has already been parsed

Variables: count_prince: (Counter) A Python Counter object that holds all the sections that will be saved for PRINCE metrics

section_list: (List) A list containing all of the base_structure sections for the parsed password

Returns: None: No return value

12.8 print_statistics.py

This file contains functionality to print “interesting results” to stdout after a training session is run.

These results are currently geared to items that will help identify the source of an unknown training dataset/leak

`lib_trainer.print_statistics.`**`print_statistics`**`(`*pcfg_parser*`)`

Prints interesting statistics from the training set

Currently just prints top e-main providers, top URL domains and top years

Variables: pcfg_parser: Of type PCFGPasswordParser. Contains the statistics to print out

Returns: Null

12.9 run_trainer.py

Responsible for running the top level training session

Moving this here to get it out of the main() function

`lib_trainer.run_trainer.`**`run_trainer`**`(`*program_info*, *base_directory*`)`

Runs through the input and performs three passes on the training set to genetate the resulting grammar

This function is also responsible for saving the grammar to disk

Inputs: program_info: A dictionary containing all of the command line option results

base_directory: The base directory to save the resulting grammar to

Returns: True: If the operations completed sucessfully

False: If any errors occured

12.10 save_pcfg_data.py

Saves data relating to the PCFG cracker to disk

This also calls functions to calculate the statistics and apply prob smoothing

`lib_trainer.save_pcfg_data.`**`calculate_and_save_counter`**`(`*filename*, *item_counter*, *encoding*`)`

Saves data for an individual Python Counter of pcfg data

Inputs: filename: (String) The full path and filename to save the data to disk

item_counter: (Counter) The counter containing all of the data for this particular item to save.

encoding: (String) The encoding to save the data as. Aka 'ascii'

Returns: True: If it completed successfully

False: If any errors occured

`lib_trainer.save_pcfg_data.`**`save_indexed_counters`**`(`*folder*, *counter_list*, *encoding*`)`

Cleans up a rules directory and saves length indexed Python counter objects

Inputs: folder: (String) The full path to the folder to save the data to

counter_list: (List[Counter]) A list of length indexed counters

encoding: (String) The encoding to save the data as. Aka 'ascii'

Returns: True: If it completed successfully

False: If any errors occured

`lib_trainer.save_pcfg_data.`**`save_pcfg_data`**`(`*base_directory*, *pcfg_parser*, *encoding*, *save_sensitive*`)`

Saves data stored in a pcfg_parser to disk

This is the top level function to call to save all of the pcfg data to disk

Inputs:

base_directory: The base directory to save the data to

pcfg_parser: Contains all of the statistics in Python counters

encoding: The encoding to save the data as

Returns: True: If it completed successfully

False: If any errors occured

12.11 trainer_file_input.py

Contains all the file reading logic for the python pcfg trainer

`class` `lib_trainer.trainer_file_input.`**`TrainerFileInput`**`(`*filename*, *encoding='utf-8'*`)`

Reads input passwords from file, one by one

Making this a class so it can return one password at a time from the training file

`read_password``()`

Returns one password from the training set. If there are no more passwords returns None

Inputs: None

Returns: clean_password: (String) The next password

None: IF there are no more passwords to parse

`lib_trainer.trainer_file_input.`**`check_valid`**`(`*input_password*`)`

Checks to see if the input password is valid for this training program

Invalid in this case means you don't want to train on them

Additionaly grammar checks may be run later to futher exclude passwords# This just features that will likely be universal rejections

Inputs: input_password: (String) The input password to parse

Returns: TRUE: If the password is valid

FALSE: If the password is invalid

`lib_trainer.trainer_file_input.`**`detect_file_encoding`**`(`*training_file*, *file_encoding*, *max_passwords=500000*`)`

Used for autodetecting file encoding of the training password set

Autodectection requires the python package chardet to be installed

pip install chardet

You can also get it from https://github.com/chardet/chardet

I'm keeping the declarations for the chardet package local to this file so people can run this tool without installing it if they don't want to use this feature

Inputs: training_file: (String) The path+name of the file to open

file_encoding: (List) A list to return the possible/recommended file encodings of the training file

max_passwords: (Int) The maximum number of passwords to parse to identify the encoding of the file. This is an optimization so this function doesn't have to parse the whole file.

Returns: True: The function executed sucesfully

False: An error occured, or the user did not have the chardet library and decided to not accept the default 'ascii' setting

`lib_trainer.trainer_file_input.`**`get_confirmation`**`(`*warningtext*`)`

Prints a warning message and asks for user confirmation

Inputs: warningtext: (String) The warning to display to the user

Returns: True: User selected Yes

False: User selected No

12.12 trainer_file_output.py

Contains all the file output logic for the python pcfg trainer

`lib_trainer.trainer_file_output.`**`create_rule_folders`**`(`*base_directory*`)`

Creates all the folders for saving a pcfg rule to disk

Will create all files in the folder specified by base directory

Returns: True: If Sucessful False: If an error occured

`lib_trainer.trainer_file_output.`**`make_directories_from_list`**`(`*directory_listing*`)`

Creates all the directories from a list of files

Returns: True: If Sucessful False: If an error occured

`lib_trainer.trainer_file_output.`**`make_sure_path_exists`**`(`*path*`)`

Create a directory if one does not already exist

Will re-raise any exceptions and send them back to the calling program

12.13 alpha_detection.py

This file contains the functionality to identify alpha strings, (letters)

This will call multiword detection, though multiword detection code is not included in this file, and is handled by multiword_detector.py using the MultiWordDetector class

`lib_trainer.detection_rules.alpha_detection.`**`alpha_detection`**`(`*section_list*, *multiword_detector*`)`

Finds alpha strings in the password

Returns: Returns two lists, alpha_list and mask_list

alpha_list: A list of alpha strings that were detected

mask_list: A list containing all of the capitalization masks that were detected

`lib_trainer.detection_rules.alpha_detection.`**`detect_alpha`**`(`*section*, *multiword_detector*`)`

Looks for alpha strings in a section

For example 123password@#$ will extract ‘password’

Variables:

section: The current section of the password to process

multiword_detector: The multiword detector to identify multiwords

Returns: There are three return values:

parsing, found_strings, masks

parsing: A list of the sections to return E.g. input password is ‘123password’ parsing should return: [(‘123’,None),(‘password’,’A’)]

found_strings: A list containing all the alpha strings found for a section

masks: A list containing all of the capitalization masks. For example, the alpha string ‘PaSSword’ would have a mask of ‘ULUULLLL’

12.14 context_sensitive_detection.py

This file contains the functionality to detect context sensitive replacements

Examples include strings that contain multiple character types that have context sensitive meaning when combined. For example ‘;p’, ‘#1’, ‘<3’

`lib_trainer.detection_rules.context_sensitive_detection.`**`context_sensitive_detection`**`(`*section_list*`)`

Finds context sensitive strings in the password

Returns: Returns one list, conext_sensitive_list

context_sensitive_list: A list of the CS strings that were detected `lib_trainer.detection_rules.context_sensitive_detection.`**`detect_context_sensitive`**`(`*section*`)`

Looks for context sensitive replacements in a section

For example password#1 would extract '#1'

Variables:

section: The current section of the password to process This function will break up the section into multiple parts if a context sensitive string is found

Returns: There are two return values: parsing, found

> parsing: A list of the sections to return E.g. input password is 'password#1' parsing should return: [('password',None),('#1','X')]
>
> found: The context sensitive string found parsing this section, or None

12.15 digit_detection.py

This file contains the functionality to identify digit strings

Note, some digit strings may be classified before this is called, for example the year detection code will classify some four digit numbers as years

`lib_trainer.detection_rules.digit_detection.`**`detect_digits`**`(`*section*`)`

Looks for digit strings in a section

For example 123password@#$ will extract '123'

Variables:

> section: The current section of the password to process

Returns: There are two return values: parsing, found_strings

> parsing: A list of the sections to return E.g. input password is '123password' parsing should return: [('123','D'),('password', None)]
>
> found_digit: The digit that was found

`lib_trainer.detection_rules.digit_detection.`**`digit_detection`**`(`*section_list*`)`

Finds digit strings in the password

Returns: Returns one list, digit_list

> digit_list: A list of digit strings that were detected

12.16 email_detection.py

This file contains the functionality to detect e-mail addresses

`lib_trainer.detection_rules.email_detection.`**`detect_email`**`(`*section*`)`

Looks for emails in a section

For example bob@hotmail.com123 will extract bob@hotmail.com

Variables:

> section: The current section of the password to process This function will break up the section into multiple parts if an e-mail is found

Returns: There are three return values: parsing, found, provider

> parsing: A list of the sections to return E.g. input password is 'bob@hotmail.com123' parsing should return: [('bob@hotmail.com','E'),('123',None)]
>
> found: The email found parsing this section, or None
>
> provider: The provider 'e.g. gmail.com' for the e-mail or None

`lib_trainer.detection_rules.email_detection.`**`email_detection`**`(`*section_list*`)`

Finds likely e-mails in the password

Returns: Returns two lists, email_list and provider_list

> email_list: A list of email addresses that were detected
>
> provider_list: A list of the providers (after the @). Yes that info is avialble in found_emails but might as well identify the provider/tld now since it is useful to display

12.17 keyboard_walk.py

This file contains the functionality to detect keyboard walks

The keyboard walk blacklist will also be contained here, as there are a lot of dicitonary words that look like keyboard walks

`lib_trainer.detection_rules.keyboard_walk.`**`detect_keyboard_walk`**`(`*password*, *min_keyboard_run=4*`)`

Looks for keyboard combinations in the training data for a section

For example 1qaz or xsw2

Variables:

> password: The current section of the password to process When first called this will be the whole password. This function calls itself recursively so will then parse smaller chunks of this password as it goes along
>
> min_keyboard_run: The minimum size of a keyboard run

Returns: There are two return values: section_list, found_list

> section_list: A list of the sections to return E.g. input password is 'test1qaztest' section_list should return: [('test',None),('1qaz','K4'),('test',None)]
>
> found_list: A list of every keyboard combo found when parsing password

`lib_trainer.detection_rules.keyboard_walk.`**`find_keyboard_row_column`**`(`*char*, *keyboards*`)`

Finds the keyboard row and column of a character

Inputs: char: (char) The character to find

> keyboards: (list) The list of keyboards to search

Returns: pos_list: (dictionary) A dictionary of all possible positions it could be for different keyboard types. Each keyboard finding is saved as the following dictionary .. code-block:: python

> **"The name of the keyboard used":{** 'row':"(int) The row of the key", 'pos':"(int) The position of key in the row"
>
> }

`lib_trainer.detection_rules.keyboard_walk.`**`interesting_keyboard`**`(`*combo*`)`

Filters keyboard walks to try and limit false positives

Currently only defining "interesting" keyboard combos as a combo that has multiple types of characters, aka alpha + digit

Also added some filters for common words that tend to look like keyboard combos

Note, this can cause some false negatives in certain cases where a true keyboard combo will happen after a false positive check but still be part of the original como. For example 'er5tgb'

Haven't seen this much in user behavior so it shouldn't have much impact but want to disclose that for future coders. May eventually want to add checks for that.

`lib_trainer.detection_rules.keyboard_walk.`**`is_next_on_keyboard`**`(`*past*, *current*`)`

Finds if a new key is next to the previous key

Inputs: past: (dictionary) A dict of all the keyboard locations for the previous character

current: (dictionary) A dict of all the keyboard locations on the current keyboard

Returns: current_runs: (dictionary) A list of all the keyboards this run is happening on

12.18 multiword_detector.py

Tries to detect Multi-words in the training dataset

Aka CorrectBatteryStaple = Correct Battery Staple

There's multiple ways to do this. For example https://github.com/s3inlc/CorrectStaple

The current method attempts to learn multiwords from the training set directly, with a small list of assist multi words to aid in this.

Aka if 'cat' and 'dog' is seen multiple times in the training set but 'catdog' is only seen once, attempt to break it into a multi-word

`class` `lib_trainer.detection_rules.multiword_detector.`**`MultiWordDetector`**`(`*threshold=5*, *min_len=4*, *max_len=21*`)`

Attempts to identify and split up multiwords

Current works by creating a base word list from the training data of words that occur multiple times. Once this base set is genreated will then attempt to break up low occurence words into smaller base words

`parse``(`*alpha_string*`)`

Detects if the input is a multi-word and if so, returns the base words

I'm overloading the multiword detector usage to also be able to detect base words as well. This can be useful for things like l33t mangling.

Because of that it returns two variables. The first one is a True/False of it the multiword detector could parse the word, the second is the parsing of the multiword itself.

Variables: alpha_string: Note making this explicit that only alpha strings should be passed in. The goal of this function is *not* to parse out digits, special characters, etc Note: This can be a string OR a list of one character items.

Returns: Two variables, are returned, If_Parsed and [Parsing of word]

If_Parsed: True if the parsing found a multi-word or a base word

False if no parsing or base word was found [alpha_string]: if the alpha_string was not a multi-word [base_word,base_word,. . .]: List of base words making up the multi-word

train(*input_password*)

Trains on an input passwords

One "weird" note about the lookup table. To speed things up, I'm not checking mimimum length of strings before I insert them into the lookup table. Therefore there will almost certainly be one, two character strings in the lookup table that aren't valid base words. That's ok though since their "count" wont' be recorded so they will not be treated as valid base words, and since they are short and most will be parts of other words, they shouldn't take up a lot of space

12.19 other_detection.py

This file contains the functionality to identify "other" strings

Note: This is the last categorization, so anything not already will been classified as "other". In general this will be what is usually labeled as special characters. E.g. '!@#$%^&. . . .' but it is labeled other to support different languages and encodings.

`lib_trainer.detection_rules.other_detection.`**`other_detection`**(*section_list*)

Finds other strings in the password

Returns:

other_list: A list of other strings that were detected

12.20 tld_list.py

This file contains a listing of Top Level Domains (TLDs) to check

`lib_trainer.detection_rules.tld_list.`**`get_tld_list`**()

Returns a list of TLDs to check

12.21 website_detection.py

This file contains the functionality to detect websites

`lib_trainer.detection_rules.website_detection.`**`detect_website`**(*section*)

Looks for websites in a section

For example passwordwww.rockyou.com123 will extract www.rockyou.com

Variables:

section: The current section of the password to process

Returns: The following are the return values:

parsing: A list of the sections to return E.g. input password is 'rockyou.com123' parsing should return: [('rockyou.com','W'),('123',None)]

url: The full url found, (unformatted)

host: The base website found, lowercased. E.g. 'google.com'

prefix: The prefix at the front of the website. E.g. 'http://', 'www.', . . .

`lib_trainer.detection_rules.website_detection.`**`website_detection`**`(`*section_list*`)`

Finds likely websites in the password

Returns: Returns several lists: url_list, website_list, prefix_list

url_list: the raw URLs that are detected

host_list: The base website. For example google.com

prefix_list: What prefixes, (if any) are put in the website, 'www', 'http:'

12.22 year_detection.py

This file contains the functionality to detect years, such as '2021'

`lib_trainer.detection_rules.year_detection.`**`detect_year`**`(`*section*`)`

Looks for years in a section

For example password2019 would extract '2019'

Variables:

section: The current section of the password to process

Returns: There are two return values: parsing, found

parsing: A list of the sections to return E.g. input password is 'password2019' parsing should return: [('password',None),('2019',Y)]

found: The year found parsing this section, or None

`lib_trainer.detection_rules.year_detection.`**`year_detection`**`(`*section_list*`)`

Finds likely years in the password

Returns:

year_list: (List) A list of years that were detected

12.23 alphabet_generator.py

Learns an Alphabet from an input training set

Another way of saying it returns the N most common characters it sees for valid passwords in the training set

`class` `lib_trainer.omen.alphabet_generator.`**`AlphabetGenerator`**`(`*alphabet_size*, *ngram*`)`

Class that generates and alphabet of the most common letters seen

Making this a class so I can re-use trainer_file_io to read the passwords one at a time, and pass them into this

`get_alphabet``()`

Returns a string of the most common N characters

`process_password``(`*password*`)`

Parse one password

12.24 alphabet_lookup.py

Creates lookup tables for OMEN ngrams using the specified alphabet

Basically this handles the initial counts and telling if 'abcd' shows in a password

class `lib_trainer.omen.alphabet_lookup.`**`AlphabetLookup`**(*alphabet*, *ngram*, *min_length=1*, *max_length=21*)

Holds all the OMEN NGRAM lookup information and statistics

Responsible for performing lookups against it

`apply_smoothing`()

Applies probability smoothing to the grammar

Also calculates the OMEN "levels" for the different counters

`is_in_alphabet`(*cur_ngram*)

Checks if a ngram can be constructed with the alphabet

Returns: True: if ngram is composed of valid characters False: if any character in the ngram is not valid

`parse`(*password*)

Parses the input password and updates the global counts

12.25 evaluate_password.py

Collections of functions to evalute a password against a previously trained OMEN model

`lib_trainer.omen.evaluate_password.`**`calc_omen_keyspace`**(*omen_trainer*, *max_level=18*, *max_keyspace=10000000000*)

Finds the keyspace for OMEN levels

Variables: omen_trainer: A previously trained OMEN dataset Note: This function will modify the OMEN dataset to cache some of the keyspace calculations for the grammar

max_level: (INT) The maximum OMEN level to calculate keyspace Note: Default of 18 was figured off the RockYou dataset as less than 5% of passwords were of a higher calculated level. Note this is biased on training and evaluating off of same dataset so in real life my gut feeling would be something like 10% would be higher than 18. Still the keyspace grows so large that calculating it isn't worth it

max_keyspace: (INT) The maximum keyspace to search. This also is a cut-off for when to stop calculating the keyspace. The reasoning for this was that too large of a keyspace there was memory issues when running the training in a Windows cmd prompt. Also, it's unlikely that a super large keyspace OMEN level would actually be employed in a PCFG style cracking session.

Return Values: keyspace: An Python Counter object with the levels and associate keyspace

`lib_trainer.omen.evaluate_password.`**`find_omen_level`**(*omen_trainer*, *password*)

Finds the OMEN level for an input password

Input Variables: omen_trainer: A previously trained OMEN dataset password: The password to evaluate

Return Values:

level: An INT representing the OMEN level of the trained password Returns -1 if the password can't be parsed

12.26 omen_file_output.py

Contains OMEN specific file IO functions to save training data to disk

`lib_trainer.omen.omen_file_output.`**`save_omen_rules_to_disk`**(*omen_trainer*, *omen_keyspace*, *omen_levels_count*, *num_valid_passwords*, *base_directory*, *program_info*)

Main function called to save all OMEN data to disk

Returns: True: If everything worked ok False: If any problems occured

12.27 smoothing.py

Handles probability smoothing for a given grammar

Breaking this out into its own section to make changing this at a later point easier

`lib_trainer.omen.smoothing.`**`smooth_grammar`**(*grammar*, *ip_total*, *ep_total*)

Responsible for applying probability smoothing to the grammar

Will update the grammar and return it with the levels assicated with each transition

–The input grammar has the format (example for ngram=4)

```
{
    'aaa': {       //the starting charaters
        ip_count: 5,    //the number of times they have shown up in the ip␣
↪(begining)
        ep_count: 3,    //the number of times they have shown up in the ep (end)
        cp_count: 100,  //the number of times they have shown up total in passwords␣
↪(for cp)
        next_letter:{   //the next letter for cp
            a:5           //represents the cp 'aaaa' with the count of the times␣
↪that cp has been seen
            b:12,         //represents the cp 'aaab'
            ...,
        },
    },
    ...,
}
```

–The output grammar will have the following format (example for ngram=4)

```
{
    'aaa': {       //the starting charaters
        ip_count: 5,    //the number of times they have shown up in the ip␣
↪(begining)
        ip_level: 2,    //the smoothed level for the ip
        ep_count: 3,    //the number of times they have shown up in the ep (end)
        ep_level: 5,    //The smoothed level for the ep
        cp_count: 100,  //the number of times they have shown up total in passwords␣
↪(for cp)
        next_letter:{   //the next letter for cp
```

(continues on next page)

(continued from previous page)

```
            a:(1,5)       //represents the cp 'aaaa' with smoothed level 1, seen 5␣
↪times
            b:(0,12),     //represents the cp 'aaab' with smoothed level 0, seen 12␣
↪times
            ...,
        },
    },
    ...,
}
```

`lib_trainer.omen.smoothing.`**`smooth_length`**(*ln_lookup*, *ln_counter*, *max_level=10*)

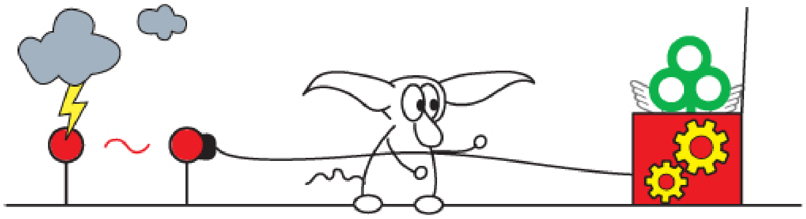
Responsible for smoothing the length info

Given a list of the lengths with their associated counts, return a list of the lengths with the (level, count)

CHAPTER

THIRTEEN

PCFG GUESSER CODE DOCUMENTATION:



13.1 pcfg_guesser.py

Name: PCFG Guesser

Probabilistic Context Free Grammar (PCFG) Password Guessing Program

Alt Title: Pretty Cool Fuzzy Guesser (PCFG)

Written by Matt Weir

Initial backend algorithm developed by Matt Weir, Sudhir Aggarwal, and Breno de Medeiros

Special thanks to Bill Glodek for collaboration on original proof of concept

Special thanks to the National Institute of Justice and the NW3C for support with the initial reasearch

Huge thanks to Florida State University's ECIT lab where the original version was developed

And the list goes on and on… And thank you whoever is reading this. Be good!

Contact Info: cweir@vt.edu

`pcfg_guesser.`**`create_save_config`**`(`*program_info*`)`

Creates the configparser object that will be used to save/load sessions

Inputs: program_info: A dictionary containing information about the current session

Returns: save_config: A configparser containing some of the values to save

`pcfg_guesser.`**`load_save`**`(`*save_filename*, *program_info*`)`

Loads a configparser object containing info about a saved guessing session

Inputs: base_directory: The directory to load the save file from

program_info: A dictionary containing information about the current session

Returns: save_config: A configparser containing some of the values to save

`pcfg_guesser.`**`main`**`()`

Main function, starts everything off

Inputs: None

Returns: None

`pcfg_guesser.`**`parse_command_line`**`(`*program_info*`)`

Responsible for parsing the command line.

Note: This is a fairly standardized format that I use in many of my programs

Inputs:

program_info: A dictionary that contains the default values of command line options. Results overwrite the default values and the dictionary is returned after this function is done.

Returns: True: If the command line was parsed successfully

False: If an error occured parsing the command line

(Program Exits): If the –help option is specified on the command line

13.2 banner_info.py

Contains banner ascii art and displays for the training program

Note: Need to print everything out to stderr since the output/guesses of this program will be sent to stdout

`lib_guesser.banner_info.`**`print_banner`**`()`

ASCII art for the banner

`lib_guesser.banner_info.`**`print_error`**`()`

ASCII art for displaying an error state before quitting

13.3 cracking_session.py

Name: PCFG_Cracker Cracking Management Code

Description: Manages a cracking session

Making this a class to help support different guess generation modes in the future

`class lib_guesser.cracking_session.`**`CrackingSession`**`(`*pcfg*, *save_config*, *save_filename*`)`

Used to manage a password cracking session

`run``(`*load_session=False*`)`

Starts the cracking session and starts generating guesses

`lib_guesser.cracking_session.`**`keypress`**`(`*report*, *pcfg*`)`

Used to check to see if a key was pressed to output program status *Hopefully* should work on multiple OSs

–Simply check user_input_char to see if it is not none

13.4 grammar_io.py

Responsible for loading up a PCFG grammar from disk

`lib_guesser.grammar_io.`**`load_grammar`**`(`*rule_name*, *base_directory*, *version*, *skip_brute*, *skip_case*, *base_structure_folder*`)`

Main function to load up a grammar from disk

Note, will pass exceptions and errors back up to the calling program

Inputs:

rule_name: the name of the ruleset to load

base_directory: The directory to load the ruleset from

version: The version of the guess generator, used to tell if ruleset is valid for this version

skip_brute: If brute force is not enabled

skip_case: If case managling should not be enabled

base_structure_folder: The folder where the base structure can be found

Returns:

grammar: A loaded PCFG Grammar, minus the (S)tart item and base structures. Mostly terminals, but some can be transforms like capitalization masks. Takes the form of a dictionary with the variable name, and a sub-dictionary of the form:

```
{
    'values':['11','51'],
    'prob':0.3
}
```

base_structures: A list of (base structures, prob) tuples in probability order. For example [(‘D2L2’,0.3), (‘D4L3’,0.2) . . .]

ruleset_info: A dictionary containing general information about the ruleset `lib_guesser.grammar_io.load_omen_keyspace(base_directory)`

Loads the OMEN keyspace information from file

This is useful for status outputs to let users know how many guesses are generated for each OMEN level

Will not catch exceptions. Passes file execeptions up to calling code

Inputs:

base_directory: The base directory to load the rules from

Returns:

omen_keyspace: A dictionary indexed by omen levels with the value being the keyspace. Initially empty Example: {'1':5000, '2':300012, '3':981138888}

13.5 pcfg_grammar.py

This file contains the functionality to parse raw passwords for PCFGs

The PCFGPasswordParser class is designed to be instantiated once and then process one password at at time that is sent to it

`class lib_guesser.pcfg_grammar.PcfgGrammar(rule_name, base_directory, version, save_file=None, skip_brute=False, skip_case=False, debug=False, base_structure_folder='Grammar')`

Responsible for holding all the information about the PCFG Grammar

`create_guesses(pt)`

Generates Guesses From a Parse Tree

This is mostly a wrapper to hide the recursive calls from the calling function. Will print guesses to stdout.

Inputs: pt: The parse tree, which is a list of tuples

Returns: num_guesses: The number of guesses generated

`find_children(pt_item)`

Finds the children for a given parse tree

Uses the Deadbeat Dad algorithm to determine if a child node should be taken care of by the current parent node

Inputs:

pt_item: A parse tree item. It is a dictionary with the following keys 'prob': The probability of the parse tree (float)

'pt': The parse tree, which is a list of tuples indexed into the grammar

'base_prob': The probability of the base structure

Returns: children_list: A list of all the children, formated as pt_item(s)

`get_status(pt, cur_guess='')`

Returns current status for a Parse Tree

Inputs: pt: The parse tree, which is a list of tuples

cur_guesses: The begining of a guess Use default (don't specify) if you are calling it directly

Returns: guess_status: Dictionary with the following keys depending on if is is an OMEN parse tree or not

```
-Omen PT: {
    'pt':[('M',1)],
    'level':"12",
    'keyspace':123953343,
    'guess_num':19533,
    }

-Non-Omen:{
    'pt':[('A3',10),('D2',1)]
    'first_guess': 'cat12'
    }
```

initalize_base_structures()

Initalizes and returns a set of parse trees from the base structures

This is used to initailize a cracking session by returning a set of the most probable parse trees for each base structure

Note, these will *NOT* be in true probability order. That will be up to whatever makes use of this list to sort them as desired

Inputs: None

Returns: pt_list: A list of the parse tree items. This is a dictionary with the following keys:

```
{
    'prob': The probability of the parse tree (float),
    'pt': The parse tree, which is a list of tuples indexed into the␣
↪grammar,
    'base_prob': The probability of the base structure,
}
```

is_parent_around(*pt_item*, *max_prob*)

Used as part of the Deadbeat Dad algorithm to identify if a child's parent node is currently in the pqueue or not. If so, this child does not need to be inserted into the pqueue.

Inputs: pt_item: The parse tree of the child

max_prob: The maximum probabilty of the parse tree. If the parent is still around it needs to be of a lower probability than max_prob.

Returns: True: There is a parent node still in the pqueue

False: There is no parent tree in the pqueue

omen_generate_guesses(*markov_cracker*)

Generates OMEN Guesses

Will print guesses out to stdout

Making this its own functions so that the load/restore and generate guesses from a normal session options can re-use this code

Inputs: markov_cracker: An OMEN MarkovCracker instance

Returns: num_guesses: The number of guesses generated for this OMEN session

print_guess(*guess*)

General code to print out a guess to stdout

If an error occurs will pass back OSError

Need to have error handling and want to centerlize all the calls to this so I don't accidently forget some printout somewhere else

Inputs: guess: The string to print out to stdout

Returns: None

`restore_omen`(*omen_guess_num*, *pt_item*)

Restores an OMEN guessing session and starts generating OMEN guesses.

Will continue the guessing session and eventually print guesses out to stdout by calling omen_generate_guesses()

Inputs: omen_guess_num: Where it is in the OMEN guess generation for the OMEN level

pt_item: The parse tree that specifies the OMEN level

Returns: Int: The number of guesses generated

`restore_prob_order`(*pt_item*, *max_prob*, *min_prob*, *save_function*, *left_index=0*)

Walks through the pt_item restoring children using save_function

Note: This works recursivly with the first call being passed the base_item which is the most probable pt parsing

Inputs: pt_item: A pt_item to parse

save_function: The function to call to save valid children

left_index: The index to find children at. The orig calling function should not use this

Returns: None

`save_to_file`(*filename*)

Sets the PCFG grammar to output guesses to a file vs. stdout

Note: If filename = None, then will continue to use the standard stdout option for guess generation, which is nice when parsing inputs from the command line

Additional Note: Not currently using this in the main pcfg cracker but leaving this in for other tool use.

Inputs: filename: The name of the file to save guesses to

Returns: None

`shutdown`()

Cleanup function when shutting down to ensure that any output files are properly closed

Inputs: None

Returns: None

`write_guess_to_file`(*guess*)

Saves the actual guess to file. Used to overload the normal print_guess to stdout function

Inputs: guess: The password guess to save to file

Returns: None

13.6 priority_queue.py

Name: PCFG_Guesser Priority Queue Handling Function

Description: Section of the code that is responsible of outputting all of the pre-terminal values of a PCFG in probability order. Because of that, this section also handles all of the memory management of a running password generation session

class `lib_guesser.priority_queue.`**`PcfgQueue`**(*pcfg*, *save_config=None*)

Main class for handling the classic PCFG next function using a PQueue

This is the "next" function to use if you want to generate guesses in true probability order

I may make changes to the underlying priority queue code in the future to better support removing low probability items from it when it grows too large. Therefore I felt it would be best to treat it as a class. Right now though it uses the standared python queue HeapQ as its backend

`insert_queue`(*queue_item*)

Inserts an item into the pqueue

Making this its own function in case I decide to change how the pqueue operates in the future

Inputs: queue_item: The value to save in the pqueue

Returns: None

`next`()

Pops the top value off the queue and inserts children back

Inputs: None

Returns: pt_item: A parse tree item that was popped off the queue

None: If no items are left to be popped from the queue

`restore_base_item`(*base_item*)

Restores all the items from the base_item to the pqueue

This is used to restore a previous guessing session

Inputs: base_item: A pt of the most probable pre-terminal for a base_item

Returns: None

`update_save_config`(*save_config*)

Updates the config file for saving/loading sessions, with current status

Inputs: save_config: A configparser object to save the current state

Returns: None

class `lib_guesser.priority_queue.`**`QueueItem`**(*pt_item*)

Wrapper for a parse tree item

This is so the Priority Queue can determine where to place a parse tree item Aka it holds all the "less than", "greater than", "equal to", logic for abs parse tree

Need to have a custom compare functions for use in the priority queue

I have to do this the reverse of what I'd normally expect since the priority queue will output stuff of lower values first. Aka if there are two items with probabilities of 0.7 and 0.4, the PQueue will by default output 0.4

13.7 status_report.py

Name: Status Report Description: Responsible for tracking statistcs and printing status reports

`class lib_guesser.status_report.StatusReport`

Used to keep track of a gussing session's status

`load`(*save_config*)

Loads data for the status output from a previously saved session

Inputs: save_config: A configparser object to load the current state

Returns: None

`print_help()`

Prints out help info for the status report to stderr

Inputs: None

Returns: None

`print_status`(*pcfg*)

Prints a status report to stderr

Inputs: pcfg: The PCFG grammar Object

Returns: None

`update_save_config`(*save_config*)

Updates the config file for saving/loading sessions for the status output, with current status

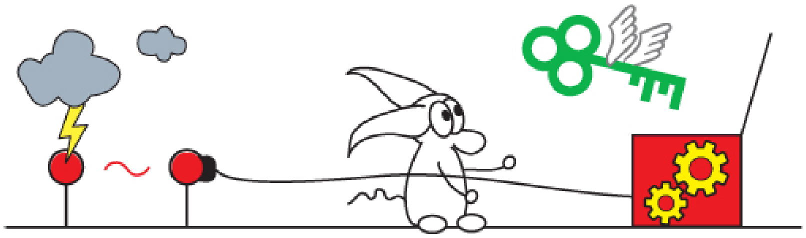
Inputs: save_config: A configparser object to save the current state

Returns: None

CHAPTER

FOURTEEN

PRINCE-LING CODE DOCUMENTATION:



14.1 prince_ling.py

Name: PRINCE-LING – PRINCE Language Indexed N-Grams

– Uses data from the PCFG trainer to make wordlists optimized for use in PRINCE stlye attacks

Contact Info: cweir@vt.edu

`prince_ling.main()`

Main function, starts everything off

Inputs: None

Returns: None

`prince_ling.parse_command_line(`*program_info*`)`

Responsible for parsing the command line.

Note: This is a fairly standardized format that I use in many of my programs

Inputs: program_info: A dictionary that contains the default values of command line options. Results overwrite the default values and the dictionary is returned after this function is done.

Returns: True: If the command line was parsed sucessfully

False: If a value error occurs

(Program Exit): If argparse is given the -h option

14.2 banner_info.py

Contains banner ascii art and displays for the training program

`lib_princeling.banner_info.`**`print_banner`**`()`

ASCII art for the Pring-Ling Banner

Inputs: None

Returns: None

14.3 wordlist_generation.py

Logic to create the PRINCE wordlist

Top level function is: create_prince_wordlist(pcfg, max_size)

`lib_princeling.wordlist_generation.`**`create_prince_wordlist`**`(`*pcfg*, *max_size*`)`

This is basically a stripped down version of the normal PCFG guesser

You give it a specialized PCFG grammar tailored for generating PRINCE style guesses, and it outputs the guesses generated for this.

Note: The PCFG grammar for PRINCE generally is typically only one transision, (or two if it has case mangling), so it should be pretty quick and not expand the size of the pqueue since each node will generally have only one child, (or two if case mangling).

Note: The grammar itself contains information about where the guesses generated will be saved (either stdout, or to a file)

Inputs: pcfg: The PCFG grammar

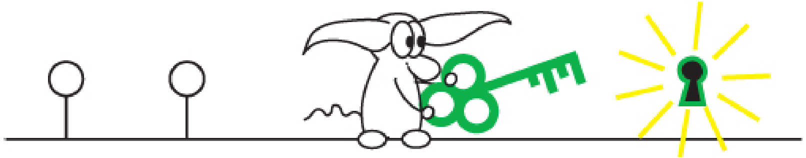
max_size: the maximum number of guesses/wordlist items to generate using this tool

Returns: None

CHAPTER

FIFTEEN

PASSWORD SCORER CODE DOCUMENTATION:



15.1 password_scorer.py

Name: PCFG Password Scorer –Will attempt to assign a probability score to an input values

> **–Uses previously trained grammars created by trainer.py to assign** probabilities to different transistions and terminals
>
> **–PLEASE DO NOT USE AS PART OF A PASSWORD CREATION POLICY OR BLACKLIST** WITHOUT MAJOR MODIFICATIONS. YOUR USERS WILL HATE YOU AND THERE'S A LOT OF FUNDAMENTAL WORK THAT STILL NEEDS TO BE DONE TO MAKE THIS USEFUL FOR THOSE USE CASES.

Contact Info: cweir@vt.edu

`password_scorer.main()`

> Main function, starts everything off
>
> **Inputs:** None

Returns: None

`password_scorer.parse_command_line(program_info)`

Parses the command line

Responsible for parsing the command line. If you have any command line options that you want to add, they go here. All results are returned as a dictionary in 'program_info'

If successful, returns True, returns False if value error, program exits if argparse catches a problem.

Inputs: program_info: A python dictionary that contains information about the program, and variables which cotain default values for command line options

Returns: True: If the command line was parsed correctly

False: If an error occured parsing the command line

15.2 banner_info.py

Contains banner ascii art and displays for the training program

`lib_scorer.banner_info.print_banner()`

ASCII art for the password scorer banner

Inputs: None

Returns: None

15.3 file_output.py

Contains file operations for saving output of a PCFG Scoring Run

`class lib_scorer.file_output.FileOutput(filename=None, encoding='utf-8')`

Saves output to a file or to stdout.

Making this a class so I can open the file once, (if needed), for writing and then can call the write function repeatedly each time there is new data to output.

The data this saves is a Python tuple but it should work for a Python dictionary as well.

`write(data)`

Takes a Python tuple and writes it as tab seperated to output followed by a newline. Will save to a file or print to stdout as defined by when this FileOutput class was initialized

Note: This only catches exceptions for some stdout issues, so if for example, the file becomes full that exception will be passed back up to the calling function.

Inptus: data: The Python tuple or dictionary to output

Returns: None

15.4 grammar_io.py

This file contains the functionality to load rules/grammar from a saved file

`lib_scorer.grammar_io.load_grammar(grammar, rule_directory)`

Loads the grammar from a ruleset.

Note, not using the normal pcfg_guesser grammar loader since that doesn't load info that we will need for essentially 're-stemming' the input words to score

Inputs: grammar: The pcfg grammar object to save the results in

rule_directory: The file directory of the rule/grammar

Returns: True: If the grammar was loaded sucessfully

False: If an error occured

15.5 omen_scorer.py

Responsible for scoring input values according to the OMEN level they would be generated at.

`class lib_scorer.omen_scorer.OmenScorer(base_directory, encoding, max_omen_level)`

Responsible for all OMEN options in the scorer Making this a class to bundle all of the OMEN functionality

`parse(password)`

Pases the password and assigns an OMEN score to it

Inputs: password: A string to parse using OMEN

Returns: (int): The OMEN level required to generate the parsed string If the string can not be parsed, returns -1

15.6 pcfg_password_scorer.py

This file contains the functionality to parse raw passwords for PCFGs

The PCFGPasswordScorer class is designed to be instantiated once and then process one password at at time that is sent to it

Note: This class is based heavily on /lib_trainer/pcfg_password_parser

`class lib_scorer.pcfg_password_scorer.PCFGPasswordScorer(limit=0)`

Responsible for holding the Grammar and evaluating inputs against it

`create_multiword_detector()`

Initializes the multiword detector

Inputs: None

Returns: None

`create_omen_scorer(base_directory, max_omen_level)`

Initializes the OMEN level parser

Inputs: base_directory: The main rules folder where the grammar is saved

max_omen_level: The maximum OMEN level to use when evaluating abs potential password

Returns: None

`parse`(*password*)

Parses an input value and determines if it is a password or not

Inputs: password: A string that is being evaluated to see if it appears to be a password

Returns: (password, category, probability, omen_score)

password: The password that was passed into this via the inputs

category: The category the input value was assigned

category = [pewo] p = password e = e-mail w = website o = other

probability: The PCFG probability of the input value according to the pcfg grammar used

omen_score: The OMEN level at which the input value would be generated at. If there is no valid OMEN parsing, it will be set to '-1'