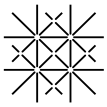


University of Basel

Óðinn: A Framework for Large-Scale Wordlist Analysis and Structure-Based Password Guessing

Master's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Databases and Information Systems Group
https://dbis.dmi.unibas.ch

Examiner: Prof. Dr. Heiko Schuldt
Supervisor: Silvan Heller, MSc.

Sein Coray

September 3rd, 2019

Abstract

In the last years, many websites were breached, compromising personal information of billions of users, often including their passwords. These collected credentials provide insights about used passwords. Analysis tools may provide information about the structure and common patterns of passwords, helping to understand the typical process followed by a human when choosing a password. Current state-of-the-art tools only allow the statistical analysis of the password length or characters used. While there exist approaches to further explore structures of passwords, they usually were not made to work with large-scale lists of passwords and are computationally too expensive.

This thesis introduces Óðinn: a tool exploring additional possibilities of analysis aiming at understanding human structures of passwords. We present an approach to split them into their essential components, and classifying them according to their semantic meaning. Furthermore, we show that these analysis results can be visualized and used to conclude about the quality of a password list, for example, when there are entries which most likely are not real passwords. Additionally, the analysis results can be used to guess new password candidates using observed combinations and patterns. We evaluate these new guessing methods against other state-of-the-art tools, and we find that our approaches create better candidates when benchmarking against difficult-to-guess passwords.

Table of Contents

1 Introduction

Nowadays, it is well known that a strong password should be chosen and ideally, a password manager should be used in order to secure an (online) account properly, enforcing different randomly chosen passwords for each service. Still, many people are using outdated techniques to create their passwords, including personal information, names, years, and even worse, reuse their passwords for multiple services. There are many reasons for this, e.g. laziness or too complicated password managers.

Wang et al. [22] recently looked at the passwords of an extensive collection of breaches from the last few years. They show that many passwords are still constructed with simple structures and heavily reused. Based on this dataset and the research of Wang et al., Dashlane listed top passwords used based on topics in their blog[1] which shows the most used weak techniques for password creation. This user behavior still allows guessing a password, based on knowledge about the person who constructed the password, and creating lists with the most common passwords based on collected data. Not to use strong passwords is a risk for users, as this allows hackers to steal data and potentially steal money or harm people in other ways. Beside hackers, weak passwords can also allow the recovery of a password in a legal scenario (e.g. as presented in the next section).

Typically passwords are put through an irreversible process (hashing) prior to saving them. Therefore they can not be reversed from this state (still, the process often wrongly is called decrypting/reversing/dehashing[2]). The typical approach of recovering a password is made by trying many combinations which are put through the same irreversible process as well and then compared to the searched output. This process is called brute-forcing. If the output of a tested combination matches the searched one, the password is *cracked* as it is now known which candidate produced it.

Today's modern hardware allows password reconstruction efficiently. Single Graphics Processing Units (GPUs) up to clusters of many GPUs can be used to run different kinds of attacks to try to find the original combination. Depending on the hashing method used, complex passwords can be recovered even in seconds.

[1] https://blog.dashlane.com/virginia-tech-passwords-study/
[2] https://www.techsolvency.com/passwords/dehashing-reversing-decrypting/

1.1 Motivation

Every person owning a computer and actively using Internet services needs passwords, according to Dashlane[3] on average a user has 90 online accounts. Depending on how a user manages their passwords, it is more or less challenging to recover them (e.g. if they used a password manager with randomly generated passwords). It can be an issue in case someone dies unexpectedly. If the user did not note some vital information about individual passwords/accesses anywhere, it could be difficult to access data that might be needed.

As a scenario, we assume a close relative died unexpectedly and did not leave any information about passwords used. The personal computer is using *Full Disk Encryption* which makes it impossible to take out the hard disk and directly extract data. In order to get access to the data on the hard disk and also to get access to potential online passwords stored in the browser storage, the goal is to recover the password used for the hard disk encryption.

Cracking an encrypted disk is typically a slow process. Therefore we need to optimize our guesses to have the most probable ones first. As the password needs to be entered manually every time the system is booted, we assume that a non-random password was used (otherwise it would be hard to remember and written down), potentially containing some information connected to the relative who died. We want to analyze existing password wordlists and filter out the most probable ones and then be able to construct typical passwords containing personal information which might be used in the password.

Figure 1.1 shows the high-level abstraction part of the newly created tool which should help to guess the person's password efficiently. Based on analyzed password structures and personal information, we will be able to create the best password guesses in decreasing order of probability in order to decrypt the hard disk. These analyzed structures should be determined using different aspects of passwords in order to cover a broad range of possible password construction methods and patterns. The acquisition of personal information is not in scope of this thesis. Instead of using personal information, it is also possible to use frequently used words which are popular in general.

1.2 Contribution

The contribution of this thesis is fourfold: First, we present new approaches of structural analysis of large-scale wordlists and implement these. Second, we show possible conclusions which can be drawn from analysis results. Third, we evaluate different approaches to use the analysis results to guess passwords. Doing this, we also present a comparison between other state-of-the-art guessing tools. Fourth, we integrate the candidate guessing into a large-scale password recovery tool.

[3] https://blog.dashlane.com/infographic-online-overload-its-worse-than-you-thought/

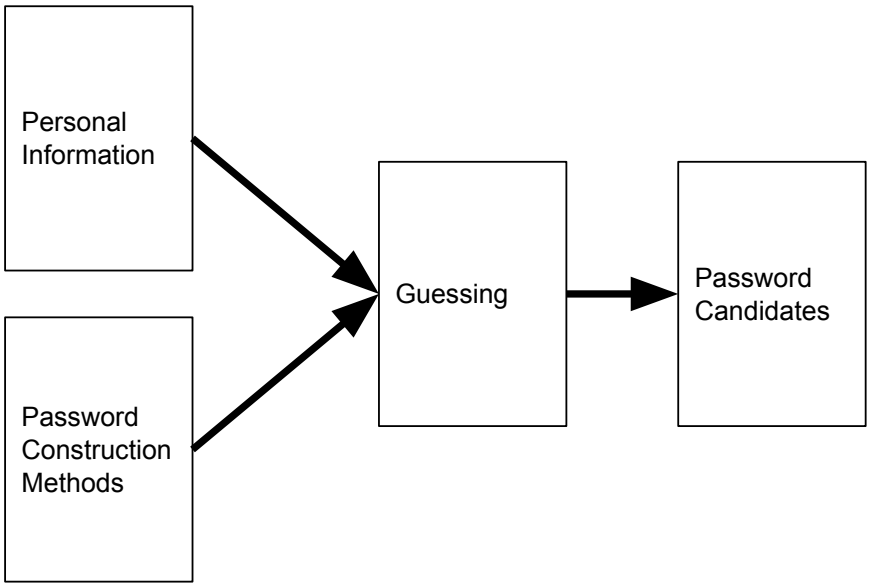


Figure 1.1: High-level abstraction of guessing a password of the person.

1.3 Outline

The application developed and presented in this thesis is named Óðinn (from the Icelandic spelling) as from the Scandinavian Mythology. There, the god Odin is said to be wise and always in search of wisdom. The document is structured in the following way: In Chapter 2 we provide a short introduction in the basics of password recovery. Chapter 3 covers the current state-of-the-art and related work in the field of password analysis and guessing. In Chapter 4 we present the architectural structure of Óðinn and in Chapter 5 we deepen some implementation-specific topics. We present and discuss our evaluation results in Chapter 6. Chapter 7 concludes the achieved outcome and shows possible future work. In Appendix A we list usage examples for Óðinn and additional plots are presented in Appendix B.

2
Foundations

This chapter provides a simple introduction into password cracking, the typically used expressions and related topics to hashes. It provides the basics needed to be able to follow this thesis further for a non-experienced reader in this field. A list of terminology for specifically used terms can be found in Section 2.4.

2.1 Password Recovery

User passwords should not be stored in plain text on a website or computer as this might reveal it to other persons such as administrators, developers, and hackers in case they get access to the database where they are stored. In order to protect the password, a so-called *hashing* is applied. Hashing is done by applying a specific one-way function to the string. This algorithm typically produces a fixed-length output which is the so-called *hash* of the password. A well-known and simple hash algorithm is *MD5*. It produces a 128-bit hash, usually represented in a hexadecimal string:

$$md5(password) \rightarrow \texttt{5f4dcc3b5aa765d61d8327deb882cf99}$$

Hash functions usually have the following properties:

- The resulting hash changes completely, even on only small changes in the input (e.g. appending a character).
- They are collision-resistant. It means that though theoretically possible (as the input set is larger than the output set), in practice it is not feasible to find two inputs for a hash function that produce the same hash.
- They always produce the same output for the same input.

The third property makes it possible to use hashes for password verification for users on authentication. The input from the user is hashed with the same algorithm and the output then is compared to the stored one. If they match, the input was identical, and the user therefore entered the correct password.

There exists a large number of hash functions, not all of them should be used for password hashing, either because they have another use case or they are broken due to cryptographic weaknesses (including MD5). Additional security can be added by the use of so-called *salts* to make it harder to attack lists containing multiple hashes. Each password has an associated salt which is for example pre- or appended to the password before hashing, leading to different hashes for different users, even if they have an identical password. For example, appending a random eight-character salt and hash it with MD5:

$$salt = \texttt{JHge6hgd}$$
$$md5(password + salt) \rightarrow \texttt{f6bb8786fa275ee22e7b164c8829a3f4}$$

The salt is then stored in plain together with the hash (or included in the hash representation). It is then used again to verify the password. An example of a hash which includes the salt inside the hash is bcrypt:

$$salt = \texttt{aaaaaaaaaaaaaaaaaaaa}$$
$$bcrypt(password + salt) \rightarrow \texttt{\$2y\$12\$aaaaaaaaaaaaaaaaaaaaaOC...rab3VFz7lFq}$$

Thanks to the one-way property, it is not feasible in practice to retrieve the password from the hash directly. The only way to find an input that produces a given hash is by hashing input candidates and test if they produce the targeted hash. This process can be called *password recovery*, *hashcracking* or *password cracking*. Typically, this process is done in a highly parallel system like GPUs. The hash algorithm which was used to hash the password affects the speed of the guesses which can be calculated per second (e.g. bcrypt is much slower than MD5 because it uses a high number of iterations of hashing).

So-called *attacks* which can be run against a hash, might be of the following variants:

Brute-Force Attack Testing all possible combinations of a specific charset, e.g. length 6 with all lowercase letters (aaaaaa - zzzzzz). This attack also is called Mask Attack[4].

Dictionary Attack Testing each line of the dictionary/wordlist. It can be a list of common words from a language or contain known other passwords.

Dictonary + Rule Attack As an extension to a dictionary attack, so-called mangling *rules* can be applied to each word from the dictionary which changes some characters, append something, etc. (e.g. a common rule is to append numbers, a typical pattern in passwords)[5].

There exist further variations (hybrids) as combinations of the above mentioned attacks or similar ones[6].

[4] https://hashcat.net/wiki/doku.php?id=mask_attack
[5] https://hashcat.net/wiki/doku.php?id=rule_based_attack
[6] https://hashcat.net/wiki/doku.php?id=hybrid_attack and https://hashcat.net/wiki/doku.php?id=combinator_attack

2.2 Password Analysis

In most cases, there is only a little or no knowledge present about the password which needs to be recovered. At that point, the question is, what kind of attacks should be executed. The typical approach is to try more probable password candidates first and then more complex attacks or candidates with a lower success rate. Brute-Force attacks are often very costly and therefore rarely used. There exist wordlists in different sizes that contain the most common passwords[7] which are suitable to test if a common password was used. This approach gets especially important when only a few guesses can be tried in a reasonable time, for example when a bcrypt hash is attacked instead of a MD5 hash (speed difference of >1GH/s on MD5 vs. 1kH/s on bcrypt).

The straight-forward approach of creating such wordlists is by collecting data about used passwords (mostly recovered from leaks) and order them by their overall frequency. Depending on the use case, there might be more sophisticated approaches to create efficient wordlists and produce guesses more likely to crack the searched password. For this, wordlists can be analyzed, also to find out what kind of words a large wordlist contains. The current state-of-the-art of wordlist analysis is described in Section 3.1. In Section 4.1, we describe how the password analysis component is integrated into the project.

2.3 Hashtopolis

Depending on the executed password recovery task, the time needed to complete is too long to be completed with one GPU. In order to have more capabilities, more GPUs can be added to be used with Hashcat. At some point, a (hardware) limitation of the maximum number of GPUs is reached where it cannot be scaled further without using multiple machines. When using Hashcat on more than one computer, manual handling of the tasks becomes tedious.

At this point, Hashtopolis[8] comes into play. It allows the distribution of Hashcat tasks to a large number of nodes by splitting the provided task into chunks. This way, long-running tasks can be completed faster than just running them on a single node. Hashtopolis can be used in local cluster environments as well as over the internet.

The strengths of Hashtopolis are that it can run on a broad spectrum of systems as long as they support Hashcat itself and that it already supports a very flexible way to create tasks and run attacks. Still, there is room for improvement to extend its functionality and support to more stability.

2.4 Terminology

Wordlist In the password recovery field, every list of words or strings can be used as a wordlist. In this thesis, we only focus on the case of wordlists containing real-world passwords. Therefore, we use the term *wordlist* denoting a list of passwords. A

[7] e.g. https://github.com/berzerk0/Probable-Wordlists
[8] https://github.com/s3inlc/hashtopolis

commonly used wordlist is Rockyou[9], which was dumped from the website unencrypted and therefore also contains long and difficult passwords.

Hashcat Hashcat[10] is the most used password recovery program. It is able either to use GPUs or CPUs to run different types of attacks against more than 200 different hashing algorithms.

Hashes/second (H/s) When recovering password hashes, the cracking speed is typically determined by how many hash calculations (and comparisons) can be done per second. This measurement gives the hashes/second ratio in which the capabilities of a system are measured. For larger numbers, the notations kH/s (10^3), MH/s (10^6), GH/s (10^9), etc. are used.

Full Disk Encryption (FDE) In order to prevent offline read access to a hard disk and its stored content, full disk encryption is used either at software or hardware level to encrypt all of the system and user files. Typically, a password is required during the boot procedure to unlock and decrypt the disk to start the system. Commonly used applications are Truecrypt[11](even if officially deprecated) or its successor Veracrypt[12].

Leak The term leak describes a hashlist/wordlist/SQL dump which was extracted from an online service and made available online. When conducting research, typically the sensitive data gets stripped out (e.g. emails, usernames, and other personal data), and only the hashes/passwords are used, either as a source for passwords (on plaintext dumps) or as a benchmark to check the efficiency of a password guesser. A well-known leak which is often used in research due to its size and worldwide coverage is Linkedin, which was breached in 2012 and contained ∼110 million user accounts.

Mask A mask is used to denote a certain structure of passwords regarding the used characters. There are two different notations of masks used in this thesis. The first one consists of using the classes *string*, *digit*, *special* to denote one or more chars of this group. For example, *password123* is included in the mask *stringdigit*. The second mask variant is more detailed, as it provides a class for each position and also knows more different classes (e.g. *?l* for lowercase characters, *?u* for uppercase characters, *?d* for digits and more). For example, *password123* is included in the mask *?l?l?l?l?l?l?l?d?d?d*. This notation also is used in Hashcat to set the covered keyspace in a brute-force attack.

Rules So-called mangling rules are a way to generate further candidates from a wordlist. The rule syntax used by Hashcat[13] and John the Ripper[14] provides possibilities of modifying the given input (e.g. appending a character). These single actions can be combined to be applied after each other (e.g. *$1$2$3* appends *123* to the input).

[9] Originally retrieved by the breach of Rockyou.com in 2009 it is nowadays commonly used as wordlist and is also present in the security Linux distribution Kali. Download: https://wiki.skullsecurity.org/Passwords#Leaked_passwords
[10] https://hashcat.net
[11] http://truecrypt.sourceforge.net
[12] https://www.veracrypt.fr
[13] https://hashcat.net/wiki/doku.php?id=rule_based_attack
[14] https://www.openwall.com/john/doc/RULES.shtml

Rule files then contain one such rule per line. Each of the rules then is applied to each of the lines of the wordlist which will produce `#of lines in wordlist` $*$ `#of rules in rule file` candidates in total.

Specifically for Hashtopolis there are further terms which we use.

Agent An agent is a node running a client binary which connects to the Hashtopolis API of the server to retrieve and run tasks.

Task A task sets a certain attack (e.g. with certain wordlists, rules) to be executed on a hashlist (defined below). This task is then distributed, which means it will be split and sent to all available agents.

Chunk To distribute each task, they are split into smaller pieces that can be completed in a pre-defined time by the agents. These resulting parts are called chunks and are defined by a skip and a length value.

Hashlist All tasks target a specific hashlist with a specific hash algorithm. A hashlist can consist of a single hash or several, but are produced by the same hash function. Hashtopolis cares about storing all hashes and their corresponding plaintexts (if recovered).

3
Related Work

This chapter describes the research done on password analysis and password guessers. We show the generally available tools on wordlist analysis and then cover the more specific parts of splitting passwords and analyze their semantic meaning. Additionally, we show an overview of the most common candidate generators called preprocessors.

3.1 Wordlist Analysis

A popular tool used for password analysis is contained in PACK [3] (Password Analysis and Cracking Kit), which statistically analyzes a list of passwords for charsets, lengths, and complexity. PACK allows the analysis of the most common simple passwords patterns used in the input and is often used to crack the remaining passwords of a hashlist. Nevertheless, it does not help to get information about how these passwords were constructed (randomly vs. words).

Florencio and Herley [11] have shown that users heavily re-use passwords on different websites and that a large number of the passwords used are of poor quality. The same was shown by Cazier and Medlin [8] by studying passwords used on e-commerce websites. They claim that people choose simple passwords because they cannot remember complex ones; otherwise, the same applies to password sharing among other sites. This issue might be addressed with password managers, but as mentioned in the introduction, still many people do not see the need of using a password manager.

A tool called password analysis and research system (PARS) was presented by Ji et al. [15] in order to support password research by providing a uniform platform integrating multiple cracking algorithms, password strength meters, and academic papers. PARS allows comparing password guessers against each other to test their efficiency on typical leaks to have a uniform comparison.

The focus of the research community lies in analyzing the most common passwords used in leaks to be able to efficiently recover all simple and often-occurring passwords from an uncracked list, but this might need a different approach if a single hash is used as the target. It still might be worth to try some standard wordlists and approaches, but if this is unsuccessful, more targeted candidates might be more efficient.

3.2 Word Splitting

In order to increase the security of passwords, some providers require a certain length to be accepted. As people often tend to use words in their passwords, they use multiple words instead of just one. These expressions might even be parts of a sentence (e.g. *ilovecats*). Bonneau and Shutova [7] have shown that users prefer to select phrases that are chosen non-randomly. They tend to select patterns that are common in natural language. The distribution of the bigrams of words is not random in most cases. Ur et al. [18] researched how similar words used in passwords are to natural language, and they have shown that passwords in English were more likely to contain nouns and adjectives compared to verbs or adverbs.

Therefore, the probability of word combinations is not equally distributed but bound to the word type. In order to research this, the password first needs to be segmented into its fragments (e.g. *ilovecats* → *i-love-cats*). As part of their password analysis, Veras et al. [21] used a combined approach to split passwords. They used a variety of English corpora with their appearing frequency and also a collection of part-of-speech tagged n-grams with their frequency of use. This data allowed to find the segmentation, which is the most likely one, as some passwords might be segmented into multiple variants.

3.3 Semantic Analysis

To see how people construct passwords, it is necessary to also know out of which words they create them. Veras et al. [21] have shown that with using semantic analysis, they were able to crack 67% more of the Linkedin and 32% more of the MySpace hashes than the state-of-the-art of probabilistic context-free grammar crackers by Weir et al. [24]. Therefore, the number of candidates can be reduced by targeting specific semantic categories. Veras et al. [21] only focused on the English language. They show how words which are not in their corpora could be added, but using trained structures on another language might be less effective as the probabilities for certain part-of-speech elements and words would be different.

Still, part-of-speech (PoS) tagging is an important aspect when considering passwords created out of (partial) sentences. Rao et al. [16] claim that by using PoS tagging, the search space can be reduced by more than 50% due to the grammatical structures. This approach can especially be useful for long passwords, as there it is more common, that multiple words were used (beside random ones). By using a grammar-aware cracking algorithm, they cracked 10% more passwords than the state-of-the-art (John the Ripper [2] at that time) when using the same amount of guesses.

Chou et al. [9] analyzed frequently used password patterns and their associated probabilities to develop a model to have a 1.4 to 2.5 times higher success rate than John the Ripper. They also claim that as long as a password is not hard to remember it stays vulnerable to smart dictionary attacks. Ur et al. [19] also have shown in their user study, that people consciously made weak passwords and also still have the feeling that adding a single special character (e.g. *!*) makes a password more secure. People also assume that birth-dates or names are secure as long as they are not on Facebook and therefore, could be used in

passwords. Ur et al. also claim that people mostly consider targeted attacks as their threat model but not as part of cracking a full leak, where the full list is attacked in general.

Another way how people are creating passwords was shown by Veras et al. [20]. They discovered that 5% of the lines in the Rockyou wordlist are pure dates. By looking at the distribution of these dates, they were able to show that specific dates are contained more frequently, e.g. first days of months, recent years, and holiday days. Such patterns mostly are language independent (besides the date notation) and could be used on hashes that are suspected to be from a non-English source.

Sen [17] showed that 30% of the Ashley Madison passwords and 36% of the Myspace wordlist contained meaningful English words (with the vast majority being nouns) and were recognized using part-of-speech tagging. They used the analysis of PACK [3] to show that a significant number of passwords in the analyzed lists consisted of sequential alpha characters which most likely are words. Additionally, they used extracted words from Twitter (and removing the stop-words which do not provide any additional personal word) to bring down the guessing space further if a specific person's password is searched.

3.4 Preprocessors

Most existing password recovery solutions like Hashcat [1] and John the Ripper [2] either use candidates defined by a mask of character sets or from a given dictionary (and applying mangling rules if needed). However, there does not (yet)[15] exist any solution where candidates can be generated directly inside the recovery software with a sophisticated approach like PCFG-generators, in general, these can be called *Preprocessors*. Bonneau [6] showed that switching from a global optimal to population-specific guessing does not give more than a factor 2 in efficiency for recovering. Therefore, it makes sense to learn the best guessing technique applied to the overall password distribution.

Due to this limitation of the password recovery solutions themselves, such candidates need to be generated externally by the preprocessor. A simple approach is to generate the candidates and save them in a file and then afterward use them in the attack. This method might work for smaller amounts of candidates, but if the number of guesses grows, saving them in a file becomes impossible or very costly regarding time and storage space. Therefore, the candidates ideally get piped directly from the preprocessor into the password recovery program. This approach works well on a single machine or when done manually, but as soon as there is a higher-level distribution, the application needs to support this piping. Multiple distribution applications exist, but none of them currently support that preprocessors can be integrated for a password recovery task.

The most common preprocessors are:

PRINCE [5] (PRobability INfinite Chained Elements) Princeprocessor takes an input wordlist and generates chains of combined words from this input with these chains having 1 to N words which are concatenated. It allows receiving higher-grade combinations than just two words. A deeper explanation how PRINCE works was presented

[15] https://hashcat.net/forum/thread-7936.html

at PasswordsCon 2014[16] and also described in more detail by ReusableSec[17].

OMEN [10] The *Ordered Markov ENumerator* can generate password guesses based on the occurrence probabilities of the used input wordlist. As the name suggests, it uses Markov chains and splits the input passwords into n-grams to extract their probability.

PCFG Cracker [4][14] The Probabilistic Context-Free Grammar Password Research Project was created by Weir et al. [24]. They claim that they were able to guess 28-129% more passwords than with the default John the Ripper [2] attack with the same number of candidates by using probabilistic context-free grammar sets trained on known passwords. This tool uses a list of passwords as the training set and splits them into their components of alpha characters, digits, and special characters. These combinations are ordered by probability and can be used to generate the most probable password constructions first.

Hitaj et al. [13] [12] compared a deep learning approach for password guessing to some of the above mentioned state-of-the-art tools with the developed tool PassGAN. The approach consisted of a Generative Adversarial Network (GAN) trained and tested on the Rockyou and Linkedin passwords. They claim to have outperformed state-of-the-art tools in most of the cases they tested, and they showed that PassGAN could compete with other password generation tools. However, there are two restrictions to be kept in mind. First, PassGAN needs to do a significant number of guesses at the start until the quality improves. Therefore, it is more suited for guessing against lists of hashes according to the learned distribution, and it might not be ideal against single targets. Second, Hitaj et al. stripped away all passwords longer than ten characters from the training and testing set. It first needs to be confirmed that PassGAN also works for longer passwords, otherwise the benefit on faster hash algorithms is relatively small, as passwords up to length ten can even be brute-forced in reasonable time[18].

There exist additional open-source preprocessors as well as some closed source ones. Of the ones mentioned above, currently, only PRINCE has a built-in mechanism to distribute the generation of the candidates. This restriction might only be an implementation-specific limitation on other preprocessors as so far there was no need to have it implemented.

[16] https://hashcat.net/events/p14-trondheim/prince-attack.pdf
[17] https://reusablesec.blogspot.com/2014/12/tool-deep-dive-prince.html
[18] With today's possible speeds (https://www.onlinehashcrack.com/tools-benchmark-hashcat-gtx-1080-ti-1070-ti.php), we exhaust the keyspace of length ten consisting of the charset a-z, A-Z, 0-9 in 17 days on ten RTX 2080 Ti GPUs. Length 9 with the same charset is already exhausted in 6.8 hours.

4

Architecture

In this chapter, we present the architecture used for Óðinn. Figure 4.1 shows an overall view of all components and how they are supposed to work together. The analysis is used to extract the desired information from a wordlist and provide it as structured data in the results file. *Results.json* is the central component of the whole structure as it holds the analysis results, which can be used for multiple further uses, e.g. to filter wordlists for specific patterns based on the analysis. The two primary purposes of the results are to serve the visualization component to produce a report about the analyzed wordlist and to use discovered password structures to generate new password candidates with the guesser. Depending on the use case, the guesser can use additional information about the target hash (e.g. words extracted from personal data) to provide more specific variations. Generated candidates can be used in Hashtopolis to distribute the workload to allow faster recovery and being able to conduct long-running tasks.

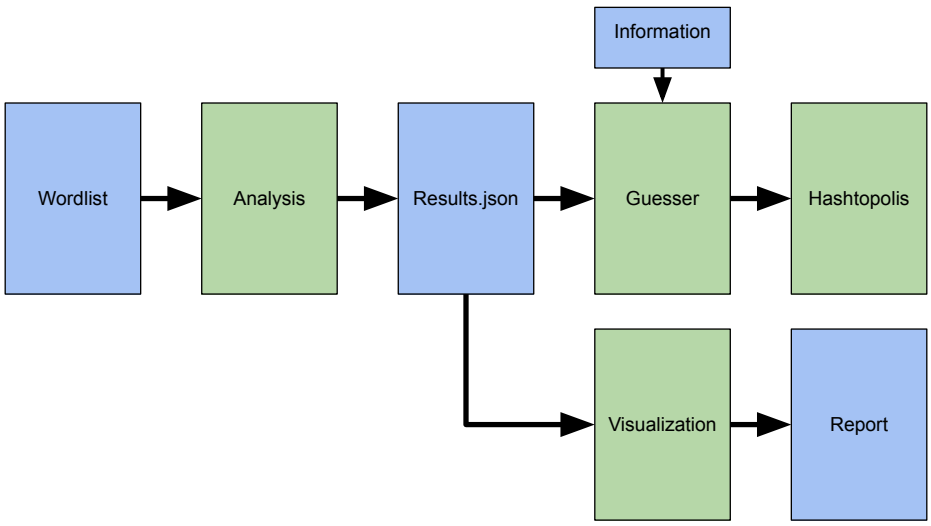


Figure 4.1: Overview of all components of Óðinn and how they work together. Data is highlighted in blue, processes in green.

4.1 Analysis

The main procedure is the analysis component, which takes a wordlist (or multiple ones) as input and analyzes them. To allow a flexible usage, each type of analysis is contained in a module that can be turned on/off for the analysis to have the desired results included. To be flexible regarding distributing the work of the analysis, we use an approach of splitting the input into small pieces (batches) and modular processing of the lines and aggregating over the data for the result. Figure 4.2 shows a high-level overview of the modules included in the analysis component.

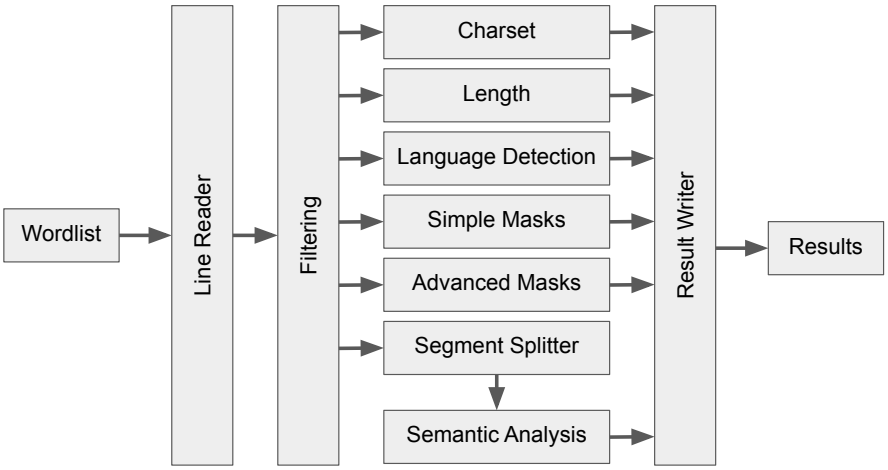


Figure 4.2: Modules included in the analysis component.

4.1.1 Filtering

It is prevalent that wordlists contain a certain amount of data which could be considered as *junk*, especially when they are large. This means, most likely these are lines which are either not real passwords or contain other data that made it into the wordlist due to other mistakes (e.g. lousy parsing when creating). Examples of such data are emails, HTML fragments, obviously generated content (e.g. extremely long and repeated), and hashes.

Depending on the use case, such bad entries should be removed before the analysis, as they could influence the result and are a waste of resources if they are not relevant anyway. As not always the same entries need to be filtered, there are modules for each type of possible junk entries. The following modules are the most important ones:

Email Detect if a line is a valid email address. Depending on the implementation, there can be multiple levels to set what should be considered as a valid email. This is explained in Section 5.1.

Length Allows setting a lower and upper bound which lines should be accepted.

Mask Filters all lines which match a certain mask (e.g. *?d?d?d?d* → all 4-digit lines). This is useful if it is known that a certain part is generated or specific patterns should not be considered.

Common Hashes It often occurs that either uncracked hashes or plaintexts which are still hashes appear in some wordlists. These for sure, are not any meaningful passwords in this representation and can be filtered. The filter can check for common lengths (e.g. 32, 40, etc.) and if all characters are hexadecimal to recognize hashes.

4.1.2 Modules

Each module can either produce output for each input line (Processor) or collected results from all input lines (Collector). The output of Processors needs to be put into Collectors before being forwarded to the *Result Writer*.

Charset (Collector) Collects the information about the character set combination used out of the four basic charsets *loweralpha*, *upperalpha*, *numeric* and *special* (e.g. *loweralphanum* denoting all lowercase characters plus digits).

Length (Collector) Collects the distribution of the length of the inputs.

Language Detection (Collector) Tries to detect if an input can be assigned to a specific language due to certain characters used and collects the distribution of these identifiable words.

Simple Masks (Collector) Collects most common simple masks based on combinations of the basic charsets (e.g. *stringdigit* denoting a password which has one or more alphabetical characters followed by at least one digit).

Advanced Masks (Collector) Collects all character masks ordered by their frequency. A mask is given with a specific length and character set used for each position (e.g. *?l?l?l?l?d?d*).

Segment Splitter (Processor) Splits each word into its components, meaning it tries to determine out of which components the word was constructed (e.g. *hello1234* would be split into *hello* and *1234*).

Semantic Analysis (Collector) Takes split words as input and tries to classify the segments by their semantic meaning to collect a distribution of the constructed words from the input (e.g. how many words were constructed from a male name and a year).

4.1.3 Results Data Model

The information extracted with a wordlist analysis is saved in the results file. Figure 4.3 shows the structure of such a results file together with some example result data. Beside some general information, each module provides its specific results. If needed, some binary data can be referenced by a file path.

The results file is a central element as there are multiple use cases where this information can be used. Examples of such use cases are described in sections 4.2 and 4.4.

```
{
  "options": {
    "-wordlist": "rockyou100.txt",
    "-threads": 8,
    "-guesser": false,
    ....
  },
  "processedLines": 100,
  "timeFinished": 1547556228,
  "version": "0.1.1",
  "analysisTimes": {
    "lengthDistribution": 0.034,
    "charsetDistribution": 0.037,
    "advancedMasks": 0.033,
    "languageDetection": 0.033,
    "simpleSplitter-fragmentCounter": 0.078,
    "simpleSplitter-simpleSemantic": 2.407
  },
  "filters": {
   "lengthFilter/0.1.0": {
     "filtered": 4
   },
   "emailFilter/0.1.0": {
     "filtered": 13
   }
  },
  "modules": {
    "lengthDistribution/0.1.0": {
      "6": 49,
      "5": 4,
      "7": 19,
      "total": 100,
      "10": 3,
      ...
    },
    "advancedMasks/0.1.0": {
      "?d?d?d?d?d?d": 6,
      "?d?d?d?d?d": 1,
      "?d?d?d?d?d?d?d?d": 2,
      "?l?l?l?l?l?l?l?l": 16,
      "?d?d?d?d?d?d?d": 1,
      "?l?l?l?l?l?l?l": 16,
      "?d?d?d?d?d?d?d?d": 1,
      "total": 100,
      "?l?l?l?d?d?d": 1,
      "?l?l?l?l?l?l": 42,
      ...
    },
    ....
  }
}
```


Figure 4.3: Structure of the analysis result. Modules are always referenced with their name and their version number (e.g. *emailFilter/0.1.0*). In *options* the command line parameters which were used when calling Óðinn creating this results file are listed. *analysisTimes* shows the overall time in seconds which each module (or chained modules) needed. In *filters* all used filter modules are listed, providing the number of entries filtered by each of them. The main analysis results are shown in *modules* where the Collectors provide their data from the analysis.

4.2 Visualization and Reporting

Based on the analysis result output, the visualization component is making it possible to gather all this information output and put it into plots and readable information summed up in a report. This report can be set to only include specific details or all analyzed results, depending on the use case. As the visualization is separated from the analysis process, it is easily possible to re-run the visualization with another configuration without having to re-run the computationally intensive and time-consuming analysis process. Figure 4.4 shows the currently considered modules and how the component is structured.

4.2.1 Modules

Each module requires certain results from the previously shown analysis modules as a dependency. Additionally, every module has a LaTeX template that allows the plot/output to be integrated into the overall report, which can then be compiled to a PDF.

Length Takes the length distribution output from the analysis and plots it as a bar plot.

Simple Masks Shows the most common simple masks.

Heatmap Plots the fragment count against the length of the password to show structures of the passwords.

Fragments Shows the most common fragments extracted, separated by type (digits, word, etc.).

Languages Prints information about the most commonly found languages in the wordlist.

Semantic Words Shows common semantic structures of words appearing in the wordlist.

Advanced Masks Lists the Hashcat charset masks ordered by frequency count. It might also include information about which masks might be more worth to run and which not (depending on the keyspace).

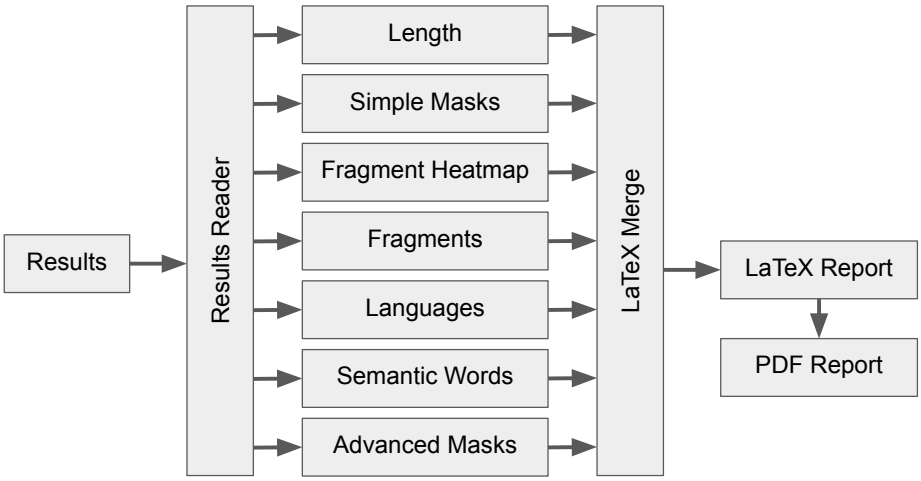


Figure 4.4: Structure and modules of the visualization component of Óðinn.

4.3 Guesser

Similarly, as the other parts of Óðinn, the guessing part is built in modules. Each module has specific requirements from the results data and uses a different approach to create candidates based on this data. Below, we describe each module and show what it can generate.

4.3.1 Fragment Prepend/Append

This module takes a wordlist as base input and appends/prepends one or more of the most common fragments to these entries. Figure 4.5 shows an example output from a list of fragments ordered by frequency and a base wordlist. Depending on what a user wants to run, either both operations or only one of prepend or append can be applied.

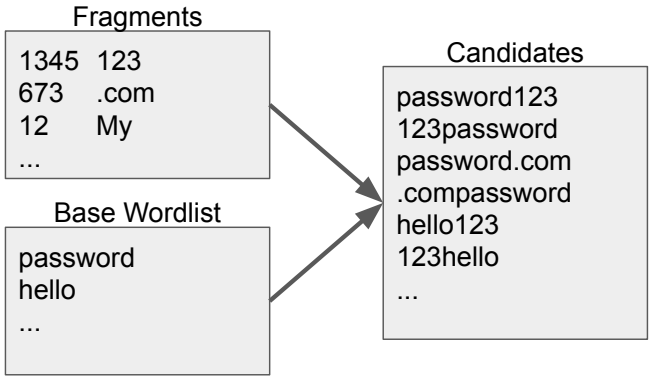


Figure 4.5: Example guessing of the FragmentPrependAppend module.

4.3.2 Fragment Mixer

Takes the most common fragments (determined from the analysis) and combines them together up to N times. Figure 4.6 shows an example of output from a list of given fragments.

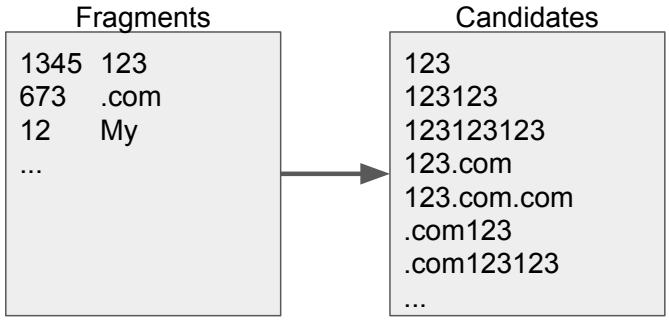


Figure 4.6: Example guessing of the FragmentMixer module with using a maximum of three combined fragments.

4.3.3 Semantic

This module takes the most common semantic combinations and creates guesses with words from the classified groups. The candidates can either be from defined class wordlists or other classification libraries. Figure 4.7 shows an example using two common semantic patterns *Female Names* and *Years* to generate combinations from the provided wordlists ordered by frequency or importance.

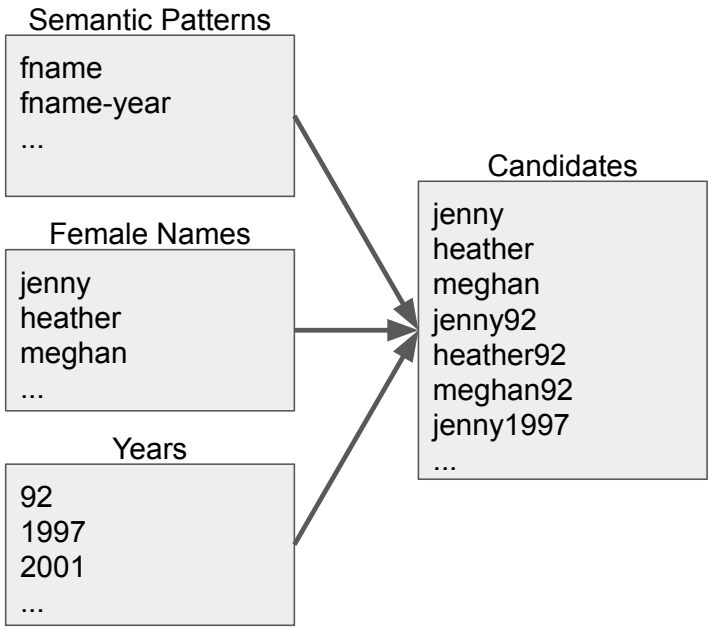


Figure 4.7: Example guessing of the semantic module where using two input wordlists.

4.3.4 Bi-Fragment Mixer

This module consists of two phases. It uses the model of Markov chains. From the analysis result, we have a frequency count for the fragment pairs occurring in the wordlist. We define N as the total number of pairs and each pair e consisting of the two elements e_l and e_r and a frequency count e_c.

$$e = \langle e_l, e_r, e_c \rangle$$
$$E = \{e_0, e_1, e_N\}$$

Pairs, which have l as their left component, are defined as:

$$E(l) = \{e \in E | e_l = l\}$$

When we now take the sum of the frequency counts for a given l defined as:

$$F(l) = \sum E(l)_c$$

we can define the probability for r following l given the tuple e containing this pair:

$$P(r|l) = \frac{e_c}{F(l)}$$

Based on these probabilities, we can build a lookup table. We use l as the key and provide a list of all possible r. This list is in decreased order by the probability of these possible r. The resulting lookup table models all possible transitions from l to r appearing in the analyzed wordlist. Figure 4.8 shows an example of such a lookup table. In practice, it can grow rapidly, especially for common fragments l which will have a large number of possible r elements.

In the second step, the guesser starts by taking the specific element $l =$ `START` and takes the n most probable followers to continue with. For each follower, it recursively goes through their most probable followers again until a specific limit or an $r =$ `END` fragment is reached. All the resulting fragments from these paths are combined and printed as candidates, as shown in Figure 4.9. As the lookup table has the followers ordered by frequency, the printed candidates will be in decreasing order of frequency (based on the analyzed wordlist).

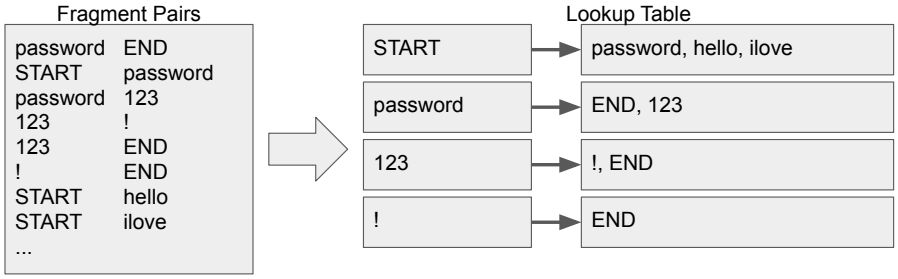


Figure 4.8: Building the lookup table for the BiFragmentMixer module from a given list of fragment pairs ordered by frequency. *START* and *END* denote the begin and the end of a password combination respectively.

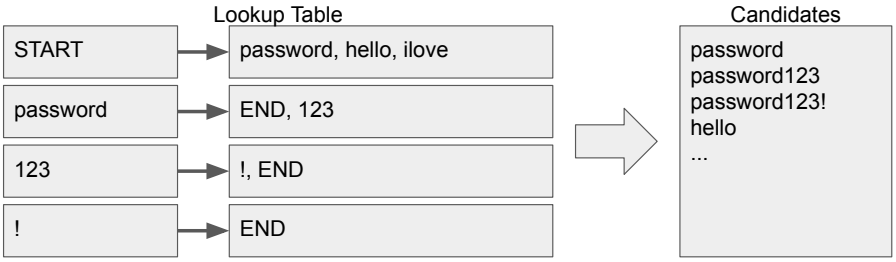


Figure 4.9: Generating candidates using the lookup table, beginning from *START* and its followers.

4.4 Preprocessor

In addition to the mentioned preprocessors in 3.4, we want to include our simple password guessers which take the analysis results and generate candidates to be piped into Hashcat. At least some of these guesser modules should allow the use of chunking (skip/limit). It will enable having at least two preprocessors ready to be used in Hashtopolis. Therefore, the password guesser needs to be capable of chunking the output space to make it possible to distribute the generation of candidates.

Having a password guesser working with the analysis results allows comparing guessing results (and therefore also the analysis results) against existing approaches. Additionally, having a general inclusion of preprocessors into Hashtopolis might motivate developers of preprocessors to include support for distribution to allow running large-scale password recovery tasks. Preprocessors like OMEN and PCFG-cracker have some internal structures which already allow some splitting of the generation process. Therefore only small changes are missing to add support for Hashtopolis.

Figure 4.10 shows the workflow which will happen when a preprocessor is used in Hashtopolis. The preprocessor receives a specific range in which it should generate the candidates. These candidates are printed to STDOUT and piped into the Hashcat process. Typically (primarily, when having faster hash algorithms), a so-called amplifier is used on the Hashcat process. This consists of using a rule file to maximize out the hardware capabilities with Hashcat better because on faster algorithms, the pipe is not fast enough to provide enough candidates per second. The rules additionally increase the chance to recover the password(s).

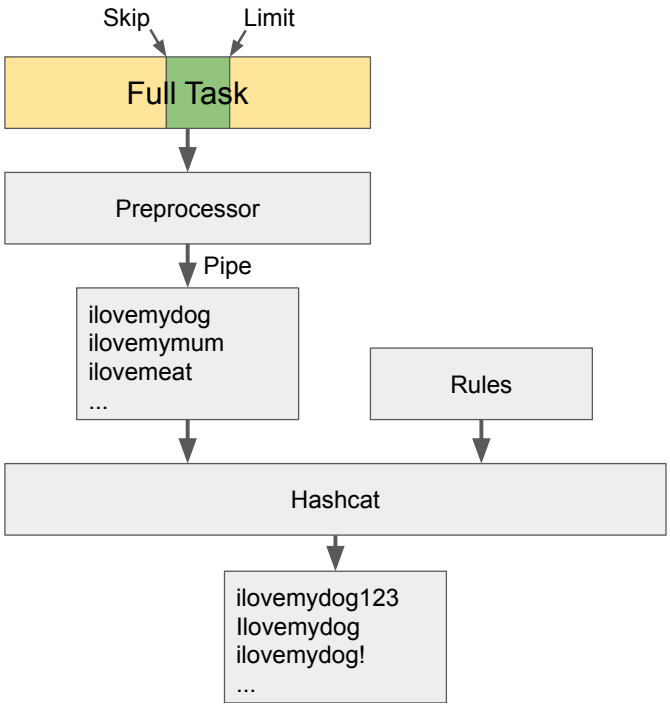


Figure 4.10: Workflow of the preprocessor integration in Hashcat. The preprocessor gets a range in the full task keyspace and needs to cover this part which is piped into Hashcat.

5 Implementation

In this chapter, we present some details about the implementation of Óðinn and the decisions chosen during this process. Furthermore, we line up some difficulties we faced and the workarounds or solutions we used.

As the analysis component of Óðinn is the computationally most expensive part (especially the semantic module), it was programmed to allow parallelization, so all CPU threads available can be used. The visualization and guesser processes are single-threaded only. During the development, we found out that it is inefficient to multi-thread over the maximum available threads for all modules. For example, with most of the filter modules, using many threads lead to high overhead, the same also applies to specific analysis modules (e.g. length distribution). For the sake of optimizing, we introduced two modes which a module can be executed with. There are two number of threads defined (*–low-threads* and *–threads*), one for modules which only require limited computation and should not be multi-threaded to many cores, and the other which sets the high maximum of threads to be used. Each analysis module got assigned to one of these two modes in order to use an appropriate number of threads.

5.1 Filtering Modules

The filtering modules serve two purposes. First, their primary function is to check for *bad* entries before they are analyzed. They can be used to filter out very lengthy strings, emails, or other unwanted patterns. The other use is if, for any reason, these specific matched entries should be extracted from the wordlist. It can be set to save all matches of the filters, and if not needed, the analysis can be turned off. Essentially, this allows just to run one or multiple filter modules through a wordlist and retrieve the matched entries.

In the following, we describe some implementation details for these filter modules.

5.1.1 Email Filter

It is obvious what the purpose of this filter is based on its name. However, in practice, it is not trivial to determine if a particular string is an email or just a password which

has some pattern in it. The email filter supports two variants of detecting if a string is an email or not. The first one only classifies emails that are using a top-level domain (e.g. *john.smith@example.org* would be seen as email, but *john.smith@sub.example.org* not). The second variant classifies everything as an email that has a valid looking subdomain ending, so an arbitrary number of dots, as long as it is a valid subdomain (e.g. *john.smith@sub.domain.org* would be accepted, but *john.smith@sub..domain.org* not). So, if it looks like a valid email domain ending, it would be classified as email. In contrary to other email validators, this way we still can classify most of the emails correctly, but it does not take in false positives like *MyP@ssword* (a very simple validator would look for the @ and therefore see it as an email address).

5.1.2 Length Filter

This filter removes entries exceeding a particular length or if desired, entries which are shorter than needed.

5.1.3 Mask Filter

Using the same notation as Hashcat, this filter can remove entries that match a particular mask (e.g. *?d?d?d?d* for all numbers with four digits). It can primarily be useful if someone wants to get all entries with this mask from the wordlist and then proceed these further.

5.1.4 Fragment Count Length Filter

This filter allows setting restrictions in which range the length of passwords should be and how many fragments they should consist of. It allows having a closer look at a specific range from the Fragment Count Heatmap (see Section 5.3.2) in order to explore which entries caused this specific part. Alternatively, if not wanted, these entries can be removed.

5.1.5 Semantic Filter

With this filter, specific semantic matches can be filtered out. It can be used to have a look at which passwords get classified as such from the wordlist. Alternatively, to exclude specific known patterns from the analysis.

5.2 Analysis Modules

There are two types of analysis modules, the ones which produce a result output which will be saved (Collector) and the second group which applies specific changes to the input which will be processed afterward by another module (Processor).

5.2.1 Simple Splitter (Processor)

This module is responsible for splitting each input line into fragments according to their character class. There, the differentiation is made between three classes: alpha (a-z, A-Z)[19], numeric (0-9), special (everything else). With this module, a simple fragmentation is done. If there are phrases or multiple words used sequentially, this module might not be enough. For example, when having the password *correcthorsebatterystaple*[20], it will not be split at all, as it consists of alphabetic characters only.

5.2.2 Full Splitter (Processor)

The *Full Splitter* uses the already separated fragments from the *Simple Splitter* and tries to split the alpha elements further into their single words (if there are multiple words). In order to achieve this, two components were used:

Symspellpy The python implementation of Symspell[21] is used to split a longer string (possibly containing multiple words) into their single parts. Symspellpy uses a frequency dictionary to evaluate which separation is the most likely one. The quality of the segmentation highly depends on the quality of the loaded dictionary.

Frequency Dictionary As passwords often are not from proper English language only, the default dictionary provided with Symspellpy was not good enough to separate most of the passwords correctly. In order to cover more than just English and as well cover Internet slang, we used a provided collection of Reddit[22] comments data[23] to extract all words and run a frequency count over this list. As even with heavy cleaning of these texts, there are still some unwanted words contained, we cut the frequency dictionary afterward below a set count threshold[24]. With this wordlist, we expect to cover most of the currently used language (e.g. new words, slang), under the assumption they are used in passwords.

After certain tests, we found out that some words were meaningless combinations of multiple words in the dictionary (e.g. *true*, *blue* and *trueblue*). Symspellpy consequently did not separate *trueblue* into two words, making infeasible the following classification by the semantic module. In order to avoid cutting off a lot more of the dictionary (we have adjusted the cut threshold to a higher value, resulting in the loss of some valid entries), we created a second dictionary where we set the cut threshold relatively high. When now trying to split into fragments, the small dictionary is used first. If there is a possible segmentation, we use this one. In case there is no segmentation found, the same process is done again with the full dictionary. This way we make sure that single words (which occur much more

[19] This includes as well valid UTF-8 characters which are considered as alpha from non-English alphabets, e.g. the German **ö** or French **é**
[20] https://xkcd.com/936/
[21] https://github.com/wolfgarbe/SymSpell and https://github.com/mammothb/symspellpy
[22] https://reddit.com
[23] http://files.pushshift.io/reddit/comments/
[24] Currently this threshold is set to 2 (we included words from the years 2008-2010 so far), when having more words from more years, the threshold can be set higher.

often) get segmented even if their concatenated version occurs in the dictionary as well, as the small dictionary gets prioritized.

In order to allow the most flexibility, it is configurable if the small dictionary should be used or not. It heavily depends on the words in the wordlist and the capabilities of the further analysis (e.g. semantic analysis).

5.2.3 Semantic Analysis (Collector)

Being able to analyze the meaning of only one single word semantically is challenging. Depending on the input, it is nearly impossible to be sure about the meaning. There are several reasons for this:

- Very short strings can be abbreviations, but often these also can have multiple matching meanings (e.g. *he* could stand for *Helium* or be the male pronoun).
- Some words could be written identically but have a whole different meaning. Without having additional information, it is impossible to detect which one is the correct one (e.g. *can* could be a verb or a noun with different meanings).
- There is no guarantee that the word is from the English language. Therefore it could happen that even if the word has a meaning in English, it could mean something different in another language (e.g. *die* which in English comes from *to die* or the singular of *dice*, but could also be the German female article). As the *Full Splitter* is also able to handle non-English input, words could mean something different or would not mean anything in English.

Still, English is the most commonly used language in Wordlists. Therefore, it is essential to be at least able to classify these. We decided to go for WordNet[25] which offers classification for single words into the so-called *synsets* which can be set into relation with other synsets (e.g. hierarchically). Most other natural language processing tools do not offer a classification of single words but require to have a full sentence, which we do not have in our scenario.

WordNet uses synset groups to provide different options if a word could have multiple meanings. In that case, they are numbered and ordered in their frequency in the English language. As we have no way to determine which one is the correct one, we always select the most common one.

Even if WordNet covers the English language pretty well, there are certain limitations and issues for our use-case:

- As soon as there is a typo in a word, it will not be able to classify the word, as it has to match exactly.
- Short strings can be classified as some strange synsets (e.g. *he* would be classified as a chemical element because it is used in the periodic table for Helium).

[25] https://wordnet.princeton.edu/

- Names, brands, games, and similar types of words are completely unknown to WordNet. For example, passwords often contain names; therefore, WordNet misses a lot of important classes in our case.
- WordNet classifies very specifically, so even similar words whose meaning would belong to the same higher group of words are classified with different synsets.

In order to tackle the challenges with Óðinn, we used the following three types of classifications detailed afterward:

WordNet Classify the word with WordNet to a synset and then try to find a parent synset to have broader coverage for the meaning. This merging process is more exactly described below.

Class Dictionary Provide lists of words assigned to a specific meaning to classify. For example, have a list of female first names to classify all these words as female names.

Class Function Using programming logic to determine if a given input matches a certain class. For example, having one function to detect if a given input is a year. In the function, we define that all inputs with two or four digits, which are in a reasonable year range, are classified as years.

5.2.3.1 WordNet Classification

If we assume the words *love, hate, emotion* as an example, we see that these words semantically could be in one group, but WordNet classifies each of them into a separate synset. If we only aggregate all classifications directly from the synsets, most or all synsets get low counts, and it is challenging to create a decreasing order of combinations as there would be too few occurrences.

To improve this, we need to match synsets together to broader classes. WordNet allows getting the parental synset, as the synsets are built into a tree with one root when traversing up by parental nodes. Unfortunately, there are significant differences between synsets on how many parents have to be traversed until reaching the root. Therefore, it is not possible to have a single number of steps to go up to reach a broader synset. We used an empirical approach to determine common synsets which could be grouped, by using the semantic distance calculation provided by WordNet (*shortest_path_distance()* function). We took all synsets with a frequency higher than a threshold (at least 400 occurrences) from the Rockyou wordlist analysis and calculated the distance between each of them. We grouped those which had a small enough distance between them. During the analysis, a classified synset will be checked against each of these broader groups. If the semantic distance is small enough, the broader group will be used as the classification. An example of this process of finding a higher group to assign a synset to is shown in Figure 5.1.

The method to assign synsets to larger classes has certain costs as it is expensive to compare to each of the groups. Up to a certain level, this can be reduced by caching these assignments and checking these beforehand. As the number of same synsets is decreas-

ing exponentially, caching the most frequent ones already drastically reduced the required computation effort.

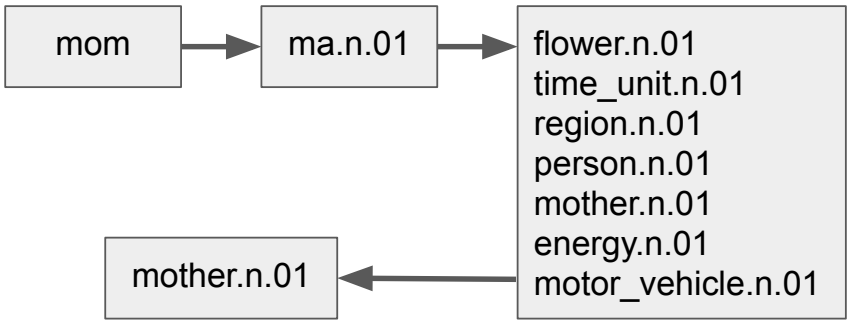


Figure 5.1: Example of the assignment of a classified synset to a higher one, based on the distance to the known higher synsets. *Mom* is classified to the synset *ma.n.01* which is defined as "informal terms for a mother" by WordNet. This synset is then compared by using the *shortest_path_distance()* and the *path_similarity()* functions of WordNet to find the best matching higher synset. In this example, the distance to *mother.n.01* is the smallest and therefore *mom* will be classified as *mother.n.01*. In case there are not any of the higher synsets close enough to the initial synset, it will be kept as it is without assigning it to another one.

5.2.3.2 Class Dictionaries & Functions

As mentioned beforehand, WordNet is only able to classify words from the English language. The class dictionaries and functions aim at covering the missing words. We first tried to let WordNet classify the words, and if this was not successful, we used our own dictionaries to find a match. Using this approach, we noticed the above-described issue that WordNet misclassified some special words. To tackle this, we split our class dictionaries and functions into two groups. One of them being tested before (*pre-wn* classes) WordNet and one afterwards (*post-wn* classes). The pre-wn classes contain tests against names, countries, cities, and months. These were sometimes classified as something completely different, e.g. the female name *Anna* is classified as *anna.n.01* which is a hyponym of the synset *indian_monetary_unit.n.01*, which is not what the meaning of this name should be in our case.

5.2.4 Fragment Counter

This module is taking the overall count for all fragments which are found in the wordlist. It will show the most common fragments appearing in the analyzed passwords. This information can additionally be used to guess new passwords with either combining the common fragments or combine them with existing other wordlists, described more detailed in Section 6.3.3.

Additionally, the number of fragments appearing for a given password length is counted. It means that the module counts the frequency of the number of elements for each password length present. For example, for the password *Password123*, the counter for length eleven

and two fragments is incremented. This data can be used to analyze some structures of the wordlist, described in Section 5.3.2.

5.2.5 Bi-Fragment Counter

Having the passwords fragmented, we can do the same as what is done with character n-grams. Instead of just splitting the password by a specific number of characters and look at the frequency of tuples like OMEN [10], we can create pairs of fragments to generate chains. For example, if we take the word *mymomisgreat*:

$$\begin{aligned} \texttt{my} - \texttt{mom} - \texttt{is} - \texttt{great} &\Rightarrow \texttt{START} - \texttt{my} \\ &\Rightarrow \texttt{my} - \texttt{mom} \\ &\Rightarrow \texttt{mom} - \texttt{is} \\ &\Rightarrow \texttt{is} - \texttt{great} \\ &\Rightarrow \texttt{great} - \texttt{END} \end{aligned}$$

The Bi-Fragment counter counts the occurrences of these pairs in order to see the most frequent or all possible followers of a particular fragment found in the wordlist.

5.3 Visualization Modules

Visualization modules serve two functionalities. First, as the name suggests, they visualize specific data from the analysis. Second, they can provide LaTeX code to either include their generated plots into the report or provide textual content or other structures from the analysis. In case of being interested in the plots only, it is possible to open the plots directly with matplotlib[26] and interactively explore certain parts of them (e.g. zoom, move).

Most of the modules are trivial in their function and what they should plot. We are describing the two most relevant modules in what follows. We discuss the output and discoveries of all the modules in the results chapter.

5.3.1 Fragment Count Distribution

The distribution of fragment lengths allows seeing if the trend is exponentially decreasing as expected by Zipf's law[27]. Figure 5.2 shows an example where we clearly see that the number of passwords is decreasing with an increasing number of fragments. Having significantly different plots than such a decreasing pattern would show that there might be unwanted password candidates in the wordlist.

[26] https://matplotlib.org/
[27] Such law is followed by all languages and passwords as well, as shown by Wang et al. [23]

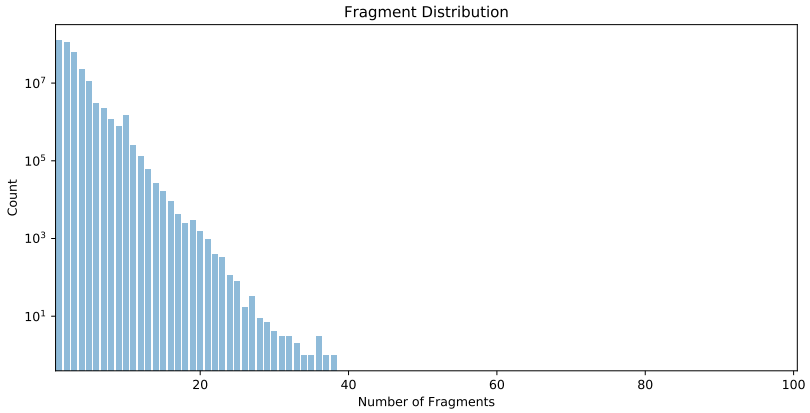


Figure 5.2: Example of a Fragment Distribution Plot.

5.3.2 Fragment-Heatmap

The fragment heatmap uses the frequency of the number of fragments for certain lengths from the fragment counter analysis module. Common passwords are in a smaller range of fragment counts and password length, having a heatmap makes it possible to have a first impression with a single view. Figure 5.3 shows an example of such a heatmap. The color denotes the frequency of the specific length and fragment count.

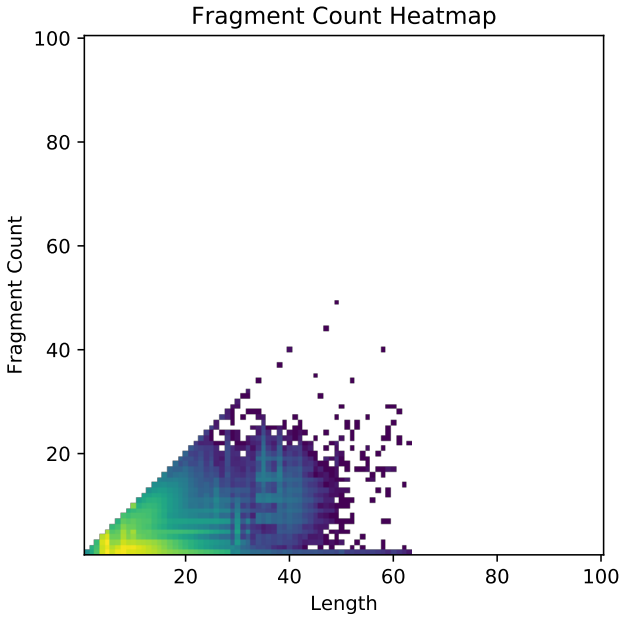


Figure 5.3: Example of a Fragment Heatmap Plot.

5.4 Guesser

Guesser modules are used to produce candidates to be tested against a hash. They use result data from the analysis and optionally additional input. As the modules were explained in detail in the architecture chapter, we focus on the two essential modules, which were also the most challenging ones to implement.

5.4.1 Semantic Guesser

This module takes one or multiple semantic constructions (e.g. *fname-year*) out of which it will generate candidates with this semantic structure. Depending on the semantic structure, it uses different approaches to retrieve words for a specific class:

WordNet Classes If the class is a wordnet synset (e.g. *emotion.n.01*), it uses WordNet's functionality to get hyponyms for the given synset, then traverses the trees of the children and takes all lemmas (in WordNet those are words in their canonical form with a single meaning) of these synsets.

Class Dictionaries The guesser starts guessing with the words present in the dictionary of that class and goes through them in the order they are in the file.

Class Functions The functions from the analysis which classify these specific classes provide additional functionality which generates candidates matching this class. For example, for the class *year* this will produce the possible numbers which are classified as years by the semantic analysis.

Compared to the two other variants, the class dictionaries allow that the candidates can be enforced to be in a specific order (if for certain reasons these entries are more likely to be correct). On WordNet, the guesser module has not any control in which order it returns the requested synsets and their lemmas. With the class functions, it could be possible to add this, but it would be very use-case dependent and requires additional programming effort to make this efficiently happen.

5.4.2 Bi-Fragment Mixer

This module makes use of the pairs of fragments that can follow after each other to build new combinations of fragments as candidates. The main challenge with this module is its memory consumption. Due to the enormous amount of different fragments and pairs, when loading the structure which is built to generate candidates grows quickly. Especially as the result file first needs to be loaded and then processed into the structure required by the module. For example, to load the analysis result of the Rockyou list into this guesser module, having 8GB of memory was not sufficient[28].

In order to overcome this specific case where loading the results file and parsing afterward is too much to handle on smaller hardware, we added the possibility to convert a results file into a CSV style file with only the required data for this guesser module. The module then

[28] The Rockyou list contains 14'344'392 lines.

alternatively can load the data directly from this prepared file instead of the results file and can handle it using significantly less memory. An example of this method can be found in Appendix A.6.

With the depth-first approach we used to go through the possible candidates with the Bi-Fragment guesser module, it happens that more probably candidates appear much later than they should. For example, if the first fragment is the most likely one, that does not mean that all the right-hand fragments are also that probable. Early right-hand fragments of the next first fragment are more common, as their frequency count is higher. In order to improve the ordering of the candidates to be closer to a strictly decreasing frequency, we added the *iterative* overlay. We sum up the position count of the fragments in the right-hand lists of a candidate to denote its probability compared to other candidates. As we have to explore the candidates in order to determine this number, we set a range in which the sum should be and only print candidates being in this range. This process we do until we covered the whole range of these sums, we start at 1 and end at `max_top * max_steps + max_steps` (according to the flags set on the guesser).

5.5 Hashtopolis Preprocessors

Hashtopolis should be able to handle multiple preprocessors with their settings and allowing the execution of tasks using these preprocessors.

5.5.1 Integration

Preprocessors are defined on the web interface in the server configuration. For each preprocessor, it should be defined how the binary is called, where it can be downloaded and which flags it uses for handling chunks. This information is mostly used by the agent, as it will take care of retrieving the binary and call it appropriately when executing a task.

When a task is being created, it can be selected that a preprocessor should be used for this task, selecting from the configured ones on the server. As each preprocessor has some unique configurations, an input field allows setting any required command line parameters to get the desired result. Also, files from the server can be entered there, similar to the regular task command line.

5.5.2 Limitations

Ideally, a preprocessor should provide the functionality to chunk the entire work into pieces. Similar as it is done with the flags *–skip* and *–limit* on Hashcat. Without having any flags to tell the preprocessor to start at a certain point and stop after a specified limit, an agent cannot start at the point where the start of the chunk is set.

In order to still support preprocessors that do not have any functionality to skip and limit the covered keyspace, we implemented a workaround using the command line utilities *head* and *tail* to just cut out the correct lines. Of course, this solution has a particular overhead, as the preprocessor always has to start from the beginning of its guessing, and the skipped lines will just be ignored. Still, this solution allows some integration, which is

better than having no possibility at all. Also, it has to be noticed that the preprocessor should always produce the same lines if called with the same arguments. If this is not the case, it is entirely impossible to include it in Hashtopolis.

6
Evaluation / Results

In this chapter, we analyze the functionality and abilities of the developed parts of Óðinn and its integration into Hashtopolis. We also compare the guessing abilities of some guesser modules to state-of-the-art preprocessors. Further, we discuss some example results showing interesting outcomes which can assist the user in understanding what is inside a wordlist.

For most of the modules, it does not make sense to evaluate their speed as their most important factor is how well they can do what they are supposed to do. We show examples from the modules, to demonstrate what they are able to do and what their limits are.

6.1 Analysis

The startup of the analysis generally is delayed by a few seconds because it needs to load the frequency dictionary for the *Full Splitter* and initialize WordNet. By far, the slowest module is the semantic analysis module as it needs to compute all the distances to optimize the classifications as described in Section 5.2.3.1. Table 6.1 shows the times required for all modules in the analysis of the Rockyou wordlist[29]. It shows that around 97% of the total time was used for the semantic analysis. If there is no need to have the semantic results for a specific task, we recommend to turn it off (for that run) to avoid a significant amount of time wasted.

6.1.1 Full Splitter Module

The *Full Splitter* module is supposed to retrieve strings which are split into their character classes (alpha, numeric, special) and test if there is further splitting possible on the alpha part. For example, having the password *greatpassword123!* which is split into three fragments *greatpassword - 123 - !*, the *Full Splitter* should split *greatpassword* further into the two words *great* and *password*.

Using Reddit comments as the source for words and building the frequency dictionary to be used by the *Full Splitter* had several advantages and disadvantages:

[29] The Rockyou wordlist contains 14'344'392 unique lines.

Table 6.1: Analysis times for modules and module chains (wordlist Rockyou), running with an Intel Core i7-4710HQ+ 2.50GHz (8 threads)

Module(s)	**Time**	**% of total Time**
lengthDistribution	105s	0.4%
charsetDistribution	104s	0.4%
advancedMasks	203s	0.7%
languageDetection	132s	0.5%
simpleSplitter-fragmentCounter	261s	1.0%
simpleSplitter-fullSplitter-simpleSemantic	25'911s	97.0%

- Since Reddit is not exclusively English, we have coverage as well for other languages. Additionally, it also includes slang words and often used expressions that are not grammatically correct. For example, using *lemme* instead of *let me*.
- The Reddit comments also included many names, brands, and expressions from games. These are not contained in standard English dictionaries.
- We used scripts to remove unwanted content from the comments (e.g. URLs, numbers, special chars only). Still, there were some expressions in the data which we could not filter out as they do not differ from any normal words. These expressions were mostly multiple words which were written together, for example, *mymom* instead of *my mom*, which caused the concatenated word also to appear in the frequency dictionary.

We discuss some examples showing what can be split and which are not optimal and show the reasons. Listing 6.1 shows correctly split examples. The most frequent wrong split happens with the mentioned concatenated words. Such examples are shown in Listing 6.2. A more rare case is shown in Listing 6.3 which occurs because the correct splitting would consist of one more word than the proposed one. Symspell prefers as few as possible words, and as *od* appears in the frequency dictionary (multiple thousand occurrences counted), it selects the wrong split. Listing 6.4 shows another possibility of why strings could be split incorrectly. Since *leville* is not contained in the frequency dictionary, Symspell tried to find another possible split which is wrong.

Listing 6.1: Correctly split strings.

```
jeparlefrancais123          -> je parle francais 123
correcthorsebatterystable   -> correct horse battery staple
ichliebedöner               -> ich liebe döner
mysafepassword              -> my safe password
dallisismybabygirl          -> dallis is my babygirl
bravegiveup123!             -> brave give up 123 !
verbiermilkshake.13         -> verbier milkshake . 13
ninalinda4ever              -> nina linda 4 ever
shannenismybabe!            -> shannen is my babe !
```

Listing 6.2: Not fully split strings.

```
mymomlovesme            -> mymom loves me
trytoguessmypassword    -> try to guess mypassword
imnotboosted            -> imnot boosted
```

Listing 6.3: Incorrectly split string due to minimal word count.

```
doyoubelieveingod?  ->  do you believeing od ?
```

Listing 6.4: Incorrectly split string due to the name *leville* not being contained in the dictionary.

```
levilleismyboo -> lev illeism y boo
```

6.1.2 Language Detection Module

The language detection was quite problematic. There are only very few characters which, without any doubt, would suggest one single language. Most of the special characters which could hint for a language can appear in at least two languages. For example, the German umlaut *ü* could not only be German but also Turkish, so without any additional information, we cannot determine which language the password is from. Natural Language Processing libraries typically require a full sentence to assign a given input to a language as they use so-called stop-words to identify it. To make language decisions based on single words, again hand-crafted dictionaries could be used to extend the detection, but this again would require much manual work and still could be prone to errors.

6.1.3 Semantic Analysis Module

The semantic module classifies each fragment into a class that will produce a list of classes for an input password. A class either is a WordNet synset (denoted by the class, the word family and the position from different possible meanings) or a class from a wordlist/function which is denoted by its name. If a fragment is not classifiable, it will be denoted by a *x*. Below we discuss several examples of classifications in order to show the possibilities and challenges which are still open.

Listing 6.5 Taken from the famous XKCD example about password strength[30]. Óðinn can classify all fragments with WordNet, but for two of the classified elements, they seem not to be the most appropriate one if compared to the original meaning from the comic. The word *correct* is interpreted as the verb from fixing something and not the adjective of being correct. The word *battery* gets classified as a size unit used in army language instead of the word used for power cells.

Listing 6.6 This example consists of two names and two additional fragments. WordNet typically is not able to classify names (and the crafted dictionaries are used), but in

[30] https://xkcd.com/936/

that case, it recognizes *nina*. The name is known, because in WordNet, it is defined as *the Babylonian goddess of the watery deep and daughter of Ea*.

Listing 6.7 The fragments from this example get classified accurately except the word *babe*. This word typically and also, in this case, is used, for example, for someone's girlfriend. WordNet associates it to the meaning of *child*, which is wrong in this context.

Listing 6.8 & Listing 6.9 These examples show that with the current WordNet data we used it is completely impossible to identify any non-English words. As in the first example, all fragments are unknown, a single x is used as classification. In contrary to the second example where the digits were identified as common-number (which means it is a combination of digits typically used in passwords), therefore it prints a classification for all fragments.

Listing 6.10 The classification of the fragment *password* may sound inappropriate, but indeed it is correct. WordNet defines this synset as *evidence proving that you are who you say you are; evidence establishing that you are among the group of people already known to the system; recognition by the system leads to acceptance* which matches for the meaning of *password* in this case. On the other side, *safe* is wrongly classified because WordNet identifies it as a noun used for safes to store valuable content and not as the adjective of something being safe.

Listing 6.11 In this example, surprisingly, the name *dallis* is not known, but the word *babygirl* is contained in the female names dictionary.

Listing 6.5: Semantic classification example #1

```
correcthorsebatterystaple
-> change_state.v.01|horse.n.01|army_unit.n.01|artifact.n.01
```

Listing 6.6: Semantic classification example #2

```
ninalinda4ever
-> nina.n.01|fname|digit|ever.r.01
```

Listing 6.7: Semantic classification example #3

```
shannenismybabe!
-> fname|be.v.01|pronouns|child.n.02|special
```

Listing 6.8: Semantic classification example #4

```
ichliebedöner
-> x
```

Listing 6.9: Semantic classification example #5

```
jeparlefrancais123
-> x|x|x|common-number
```

Listing 6.10: Semantic classification example #6

```
mysafepassword
-> pronouns|strongbox.n.01|positive_identification.n.01
```

Listing 6.11: Semantic classification example #7

```
dallisismybabygirl
-> x|be.v.01|pronouns|fname
```

Table 6.2: Most common semantic structures found in Rockyou.

Structure	**Found Occurrences**
number	2'345'568
char | number	146'460
number | char	104,674
mname | number	65'253
fname | number	54'689
metallic_element.n.01 | number	35'855
city | number	28'158
char | char | number	27'650
person.n.01 | number	24'780

Table 6.2 shows the most common semantic structures from the analysis of the Rockyou list. It only considered the combinations where all fragments were identified. We see that persons and city names appear frequently and that numbers are prevalent and are contained in nearly every structure. The class of *metallic_ element.n.01* is an additional example where issues with WordNet come into place. Typically, these structures consist of two characters followed by digits; in many cases, these two characters are the short version of an element from the periodic table (which WordNet can recognize). On the other hand, the classification into *person.n.01* most likely is correct, as such passwords can consist of relatives and humans as *dad*, *mother* or *son*.

The examples show that WordNet often has problems to provide the correct synset as the most likely one. We also think that WordNet did not keep up with the development of the language used in familiar environments and modern expressions used. This property makes it challenging to use it correctly in many cases for the classifications, and we need to rely mostly on the created dictionaries to identify fragments.

6.2 Visualization and Reports

To have a good overview of the analysis results, Óðinn can create plots for specific modules and also can summarize everything in a PDF report generated with LaTeX. In this section, we will discuss some exciting results visible in plots and what can be said about the quality of a wordlist based on them.

6.2.1 Length Distribution Module

As the name is already saying, this module plots the length distribution of the passwords in the analyzed wordlist. With this information, a first impression is given about the quality of the list. Figure 6.1 shows an expected result from a wordlist. Most common lengths are between 6 and 10, for longer passwords it is exponentially decreasing with having no entries with more than 63 characters. In Figure 6.2, on the other hand, there are several peaks at certain lengths (e.g. 32, 64) and the majority of entries is longer than 20 characters. There are lines in this wordlist being longer than multiple hundred characters. This plot clearly shows us that in this wordlist, there are a lot of non-password lines, such as hashes or other nonsense.

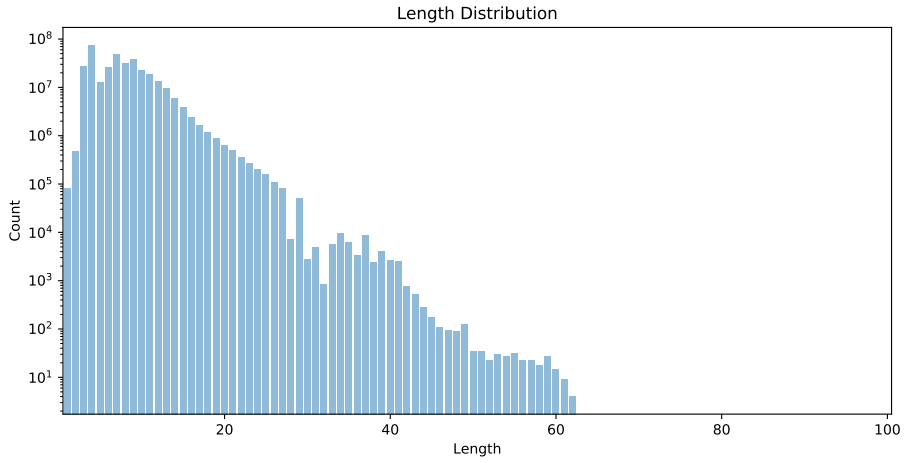


Figure 6.1: Length distribution of passwords in the Hashes.org 2015 list.

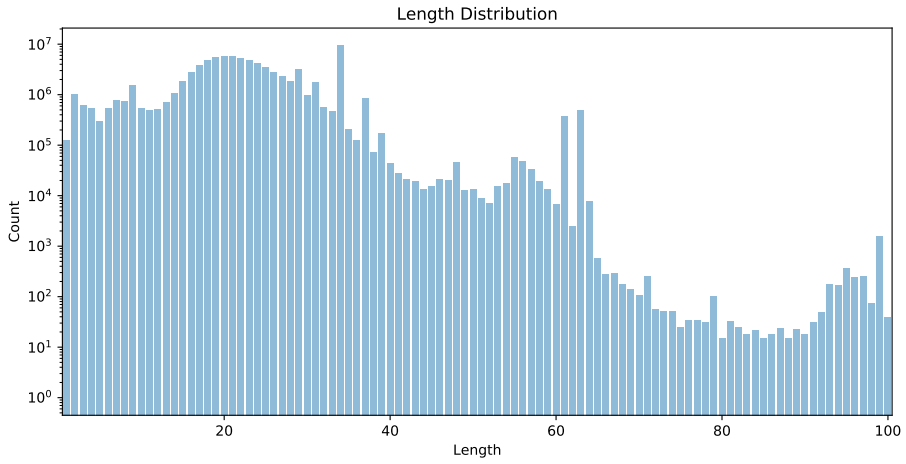


Figure 6.2: Length distribution of passwords in the Hashes.org 2015 junk list (https://hashes.org/left.php). Hashes.org uses regular expressions to filter out common non-password entries from their wordlist, these removed lines are available in the junk lists.

6.2.2 Fragment Distribution Module

Similar to the length distribution module, the fragment distribution module allows getting a quick insight into the structures of the lines of the wordlist. Most of the passwords typically are consisted of one or only a few fragments. Therefore we expect a fast decreasing amount of entries when increasing the number of fragments. An excellent example of such an expected outcome is shown in Figure 6.3. There can be different insights provided than with the length distribution. For example, the often-used Rockyou wordlist (which shows a *good* result on the length distribution) shows some structures which most likely are no passwords, as shown in Figure 6.4. In Rockyou certain lines contain HTML strings and other code elements, definitely not being passwords. These lines cause the fragment distribution to show the entries with more than 40 fragments, suggesting that this wordlist ideally should be cleaned from such entries.

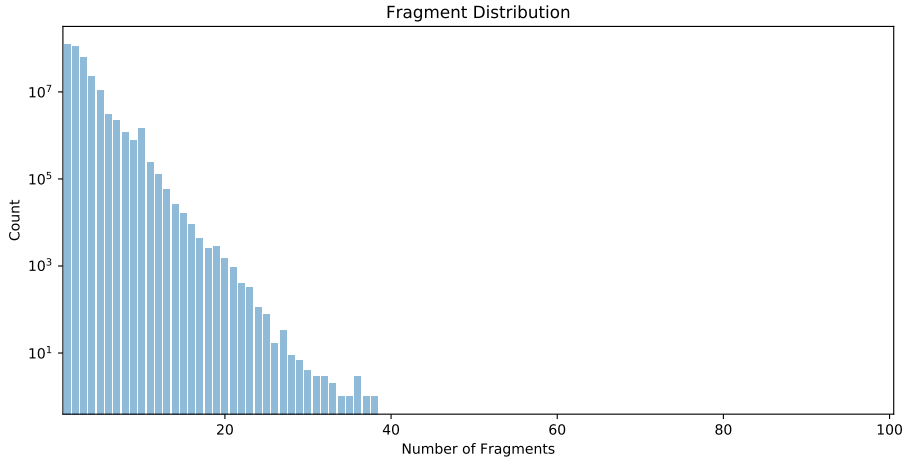


Figure 6.3: Fragment count distribution of passwords in the Hashes.org 2015 list.

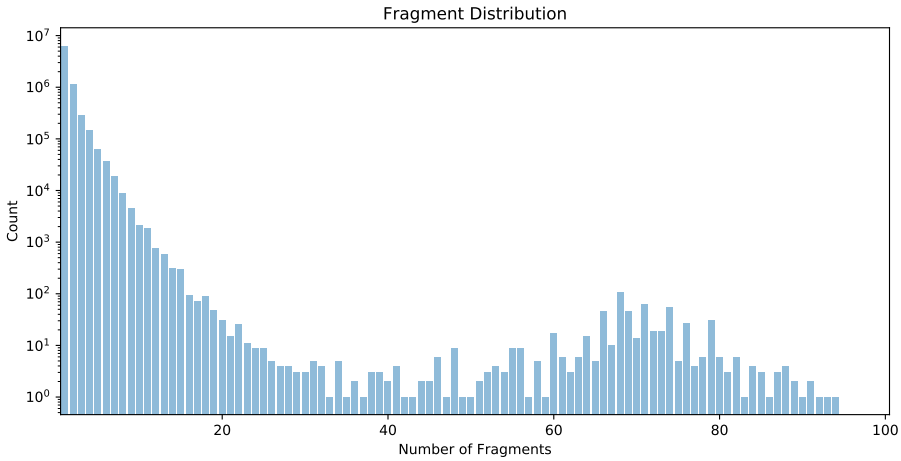


Figure 6.4: Fragment count distribution of passwords in the Rockyou list.

6.2.3 Fragment Heatmap Module

The fragment heatmap plot shows the occurring frequencies of password lengths with a specific number of fragments. This plot helps to identify patterns in the wordlist and also can show if lists contain large parts of generated content. Figure 6.5 shows the comparison of the Hashes.org 2015 against the Rockyou list. Typically these plots show a very high frequency in the range of low fragment counts and password lengths below 15 (yellow area). In this range, both lists are similar (with the Hashes.org list being slightly more erratic). When looking at the higher lengths, we see again that Rockyou must contain some non-common lines shown by the single entries in the right range of the plot. From this comparison, we can see that in the range below length 30, the Rockyou list most likely has a slightly higher quality than the Hashes.org list, but on the other hand, contains more junk lines which are long and have many fragments.

Another fascinating discovery that can be made in the Hashes.org 2015 list (Figure 6.5a) is that in the length range of 20-30, every second frame count is more frequent than the neighboring ones. When filtering out these entries from the list (using the filter features of Óðinn), we see the reason. The brighter lines (which means more frequent) are caused by lines which consist of passphrases separated by spaces or dashes, for example, *correct-horse-battery-staple*. As such combinations of words separated with single chars always have an odd count of fragments, they were causing the pattern in the plot.

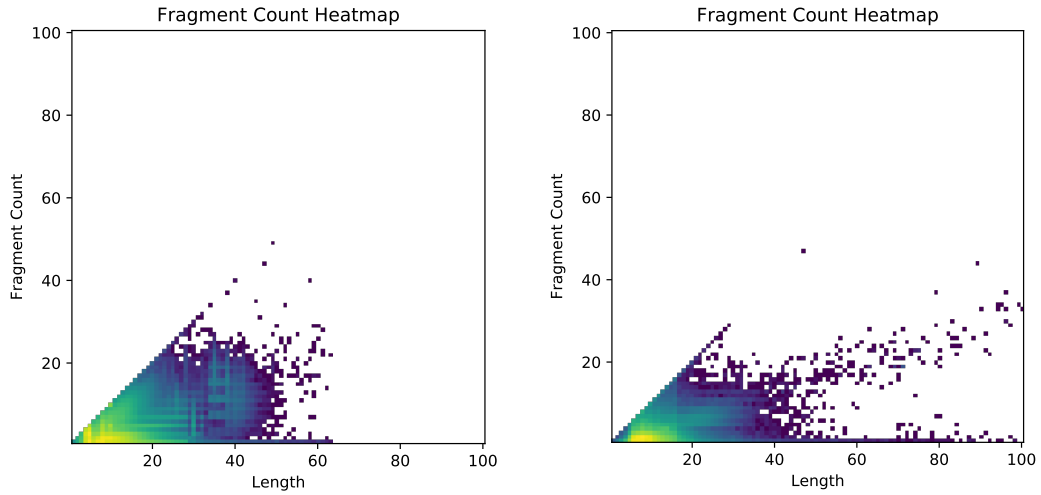
The Hashes.org 2015 junk list allows further interesting insights. Figure 6.6 shows an extended heatmap plot. We again clearly see that this list is made up of many trash lines which for sure are no regular passwords. We will have a closer look at some structures:

- There are certain lengths (e.g. 100, 128, 256) that appear more frequently. This most likely has to do with certain limitations of the original input where the passwords were entered or for generated ones that this was a certain limit.

- The highest frequency counts are not in the normally expected range (as described above) but around the length of 25 and consisting of five or more fragments. Also, there are multiple other areas where suddenly a higher frequency occurs for a single value of length or fragments.

- In the range of 35 to 60 fragments and around length 100 there is a larger cluster of entries. When looking at the lines which caused this section, we see that these entries are all cracked SHA384 hashes with their plaintexts. SHA384 hashes have a length of 84 hex characters, with their plaintexts attached the length gets close to 100. So, the lines were in the form of `98796d...9b1b:password`. The hexadecimal hash consists of characters and digits; this then caused many fragments as there are a lot of changes between digits and chars in the hash value.

The heatmap plots allow detailed insight into wordlists, together with the filter function to extract particular ranges of the plot, this is a robust tool to find unwanted entries and rating wordlists.



(a) Fragment heatmap of passwords in the Hashes.org 2015 list.

(b) Fragment heatmap of the passwords in Rockyou.

Figure 6.5: Comparison between two wordlists with the fragment heatmap.

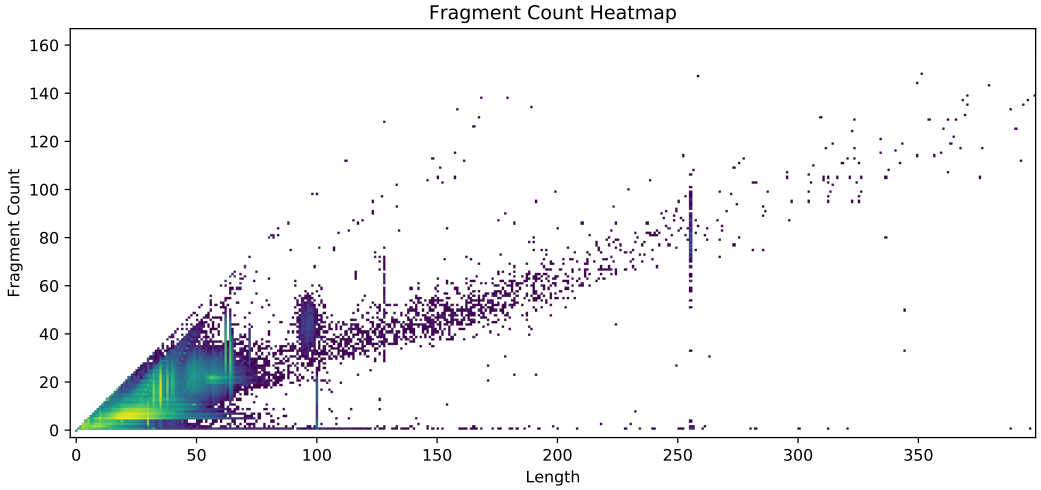


Figure 6.6: Fragment heatmap of the Hashes.org 2015 junk list.

6.3 Guessing

On the semantic analysis, the quality of the result is highly dependent on the dictionaries used to improve the classification of entries that are unknown to WordNet. Also, the guessing is depending on the dictionaries available and also on which semantic structures should be guessed. Because of that, we can not run a general evaluation of the semantic guessing. Examples on the usage of this guessing can be found in Appendix A.7. Therefore, we focus on the Bi-Fragment guessing module in this section.

6.3.1 Comparison on Password Leaks

We wanted to compare the Bi-Fragment guessing module to the other state-of-the-art tools. To make this possible without a lot of manual work, we used the Password Guessing

Framework (PGF)[31]. This allowed us to easily compare guesses of all the tools we wanted to evaluate with different testing sets. The following guessers should be evaluated:

Óðinn Using the Bi-Fragment guesser module, the specific configuration for the generation is described below.

OMEN Latest available version in the repository[32] as of August 2019.

PCFG Using the latest available version in the repository[33] as of August 2019.

PRINCE Using the latest release from the repository[34], version 0.22.

JTR Markov From John the Ripper using the Markov generation functionality, using the latest available version as of June 2019 (1.9.0-jumbo-1+bleeding-1a06dc4).

The PGF allows using wordlists and hashlists (as long as unsalted lists are used) to be used to evaluate. Also, it does not need any GPU resources to run. The checks against the hashlists are made using John the Ripper, so everything is running on CPU. The abilities of the system where the evaluation was running on, is not relevant for the outcome of the evaluation, therefore we do not go into details on that.

In order to keep the amount of resources needed to train all the tools to a reasonable time, we went for the commonly used Rockyou list as the training list. This list contains 14'344'392 unique password entries. As we wanted to find out how the guessing tools behave against other lists, we decided to use the full list as the training set and run the guessing runs against other lists only. To have some variety of results, we decided to use the following lists as testing sets, which also are commonly used in evaluations:

000webhost The hosting company was breached in 2015[35]. They stored the passwords of their approximately 15 million customers in plain text. The resulting unique wordlist contains 10'619'843 unique entries.

eHarmony The dating website eHarmony suffered a breach in 2012[36]. The passwords were stored using the MD5 hashing algorithm. The resulting hashlist contains 1'516'834 unique hashes.

Linkedin The social network Linkedin suffered a breach in 2012[37]. Four years later it became public that a much larger amount of user accounts were affected in the same hack in 2012[38]. In total more than 110 million user accounts were affected whose passwords were stored using the SHA1 hashing algorithm. The resulting hashlist contains 61'829'262 unique hashes.

[31] https://github.com/RUB-SysSec/Password-Guessing-Framework and https://password-guessing.org/
[32] https://github.com/RUB-SysSec/OMEN
[33] https://github.com/lakiw/pcfg_cracker
[34] https://github.com/hashcat/princeprocessor
[35] https://www.000webhost.com/000webhost-database-hacked-data-leaked
[36] https://www.eharmony.com/blog/update-on-compromised-passwords/
[37] https://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised
[38] https://www.linkedin.com/help/linkedin/answer/69603/notice-of-data-breach-may-2016?lang=en

PwndSub10m The breach notification service *Have I Been Pwned* released breached passwords from a variety of leaks totaling in nearly 550 million entries[39]. In order to protect the passwords, they were hashed using SHA1 and later also with NTLM. As the full list is fairly large, we took a random subset of these hashes with the size of 10 million entries.

Yahoo The Yahoo list consists of 6'458'020 SHA1 hashes. It is available on Hashes.org since 2013[40]. Most likely it was retrieved as a partial list of the 500 million large breach of Yahoo which got public in 2014[41].

Table 6.3: Removed entries from testing lists by matching against training data.

Test List	Total Entries	Removed Entries	Testing Entries
000webhost	10'619'843	451'096	10'168'747
eHarmony	1'516'834	18'491	1'498'343
Linkedin	61'829'262	3'344'334	58'484'928
pwndSub10m	10'000'000	259'772	9'740'228
Yahoo	6'458'020	93	6'457'927

To make the comparison between the tools more independent from the training set, before the tools were able to guess against the testing lists, all matches which occurred in the training list (Rockyou) were removed. Table 6.3 shows the removed entries for the tested lists. On the first run, each tool was allowed to make 100 million guesses against the remaining testing set where the matches from Rockyou were removed.

The configuration settings for generating the guesses from Óðinn were the following. From the start element it should use maximum one Million followers (*–guesser-bifragment-max-outerloops*) and for the further elements maximum the top 300 following fragments (*–guesser-bifragment-max-top*). We allowed up to 4 fragments being combined (*–guesser-bifragment-max-steps*). As we set it using the iterative procedure, a smaller number of more likely fragments was guessed first.

Results

For normal guessing, we allowed a maximum of 100 million guesses. Figure 6.7 shows the comparison against the 000webhost testing list. On the first few million guesses, Óðinn is able to outperform all other tools, shown in Figure 6.8. Afterward, the PCFG guesser is leading and outperforms all other tools.

The result when comparing the guessers against each other varies significantly, depending on which testing set is used, with the exception that the PCFG guesser is leading. For example, when we look at the evaluation against the Linkedin set, as shown in Figure 6.9,

[39] https://haveibeenpwned.com/Passwords
[40] https://hashes.org/leaks.php?id=38
[41] https://yahoo.tumblr.com/post/150781911849/an-important-message-about-yahoo-user-security

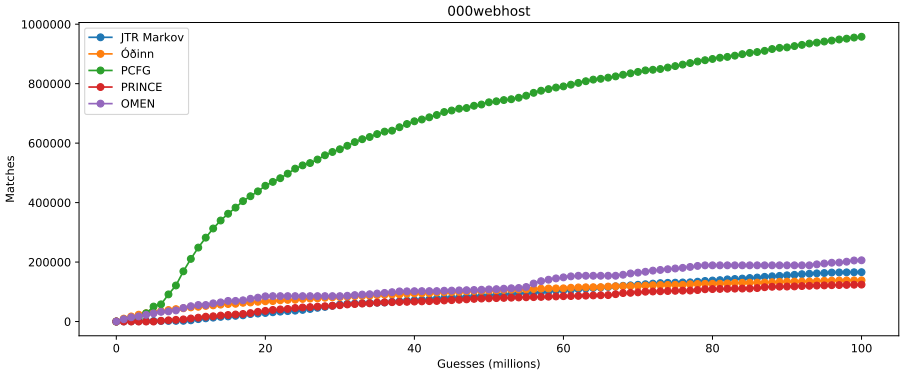


Figure 6.7: Evaluation against the 000webhost list, with a maximum of 100 million guesses.

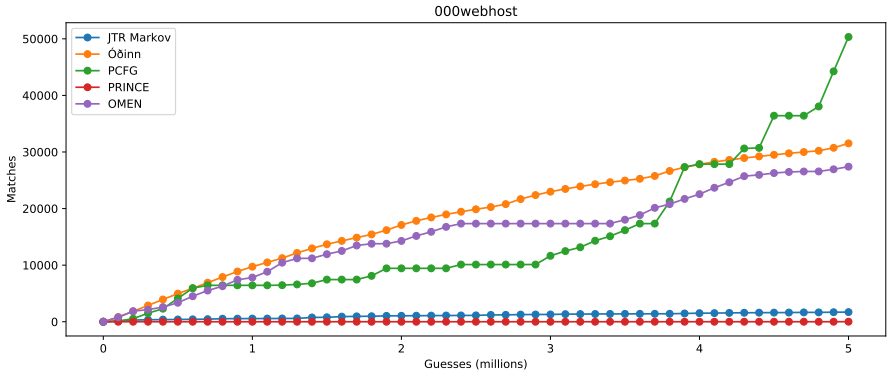


Figure 6.8: Evaluation against the 000webhost list, with a maximum of 5 million guesses.

we see that Óðinn got significantly fewer matches than some of the other tools. Compared to the others, OMEN and PCFG seem to have some steps which are more or less visible. In Figure 6.10, we see them doing some jumps. We also see that these changes happen at the same point every time in the case of OMEN. Therefore we can assume that OMEN and PCFG are not strictly guessing passwords based on the learned probabilities but use some other ordering. Further, we see that against the eHarmony list, John the Ripper Markov is not as good as against the other testing lists.

Surprisingly, there was one list, where Óðinn performed extraordinary well compared to the other tools (except PCFG). This run is shown in Figure 6.11. It also is clearly visible that the candidates of Óðinn are strictly ordered by probability, therefore the steepness of the curve is decreasing smoothly. To understand why most of the tools are that bad, we need to take a look at the matches which they got. Ten random plaintexts from the matches of each tool are shown in Table 6.4. It seems that the Yahoo dataset contains more complex passwords than would typically be observed in lists. This can be due to the fact that there were other password restrictions in place, or the dataset had the easier passwords removed beforehand. From the examples we see that the other tools mostly guess shorter passwords and get a few which have special chars (language specific chars). Óðinn on the other hand gets matches where we see the combinations of fragments, either multiple words combined

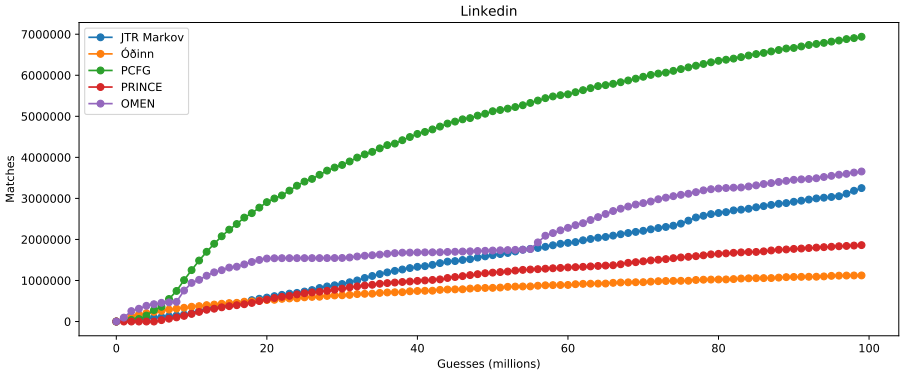


Figure 6.9: Evaluation against the Linkedin list, with a maximum of 100 million guesses.

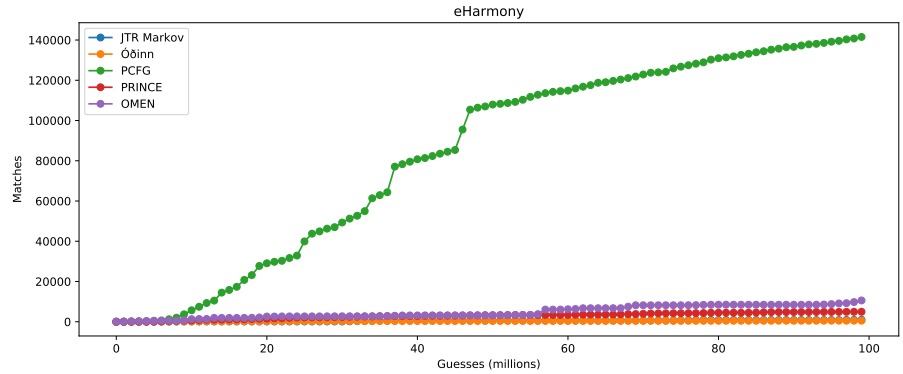


Figure 6.10: Evaluation against the eHarmony list, with a maximum of 100 million guesses.

as part-of-speech or numbers/special chars mixed with simple words.

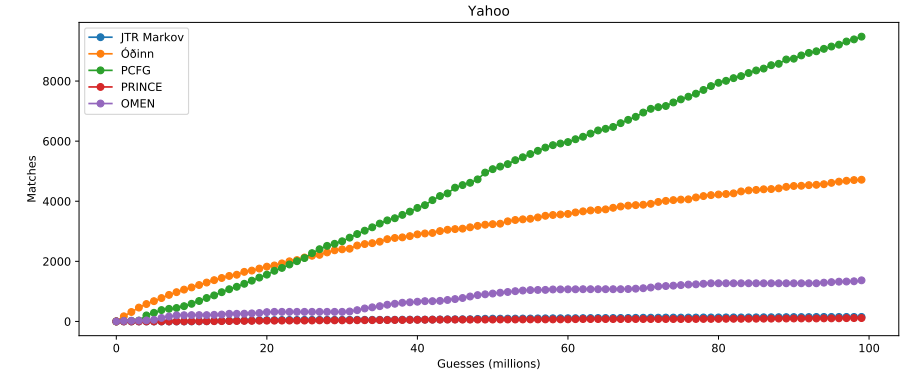


Figure 6.11: Evaluation against the Yahoo list, with a maximum of 100 million guesses.

The outcome of the plaintext comparison between the tools confirms that it is unfair to compare the guessers directly on full testing lists as we omit the case when we have more difficult passwords or a list where easy entries were already cracked. Therefore, we decided to make a second comparison where we are not interested in the absolute number of passwords recovered, but in how many entries a tool guessed which was not guessed by

Table 6.4: Random taken matches against the Yahoo list.

Óðinn	**OMEN**	**JTR Markov**	**PRINCE**	**PCFG**
lisalovesjake	puccibelle	jar-08	københavn	serenDipity
pie rocks	toneybrown	nicolò	nazlıcan	workout#1
librodemormon	robinhollie	pay me	findMe	reesenicole
justgetthere	karmantree	seseña	mvpewok	Gürkchen
cadence4now	pinkdrum	ma07!!	Buckm1	alysaemma
aug10mar	janenia2	bayernmünchen	RobMike	bambidaisy
runner4eva	pridesmith	KORČULA	Godreal	brycegreen
candy03nov	bullermona	janhjan	constança	hallkodi
cottonmouthgirl	scoobylori	arribaespaña	asdfghjklñ	jena!!
action.man	dogtiggy	bel_29	butGod	auntdiana

any of the other tools.
Following is given:

$$
\begin{aligned}
T &= \{\texttt{Óðinn}, \texttt{OMEN}, \texttt{JTR Markov}, \texttt{PRINCE}, \texttt{PCFG}\} \\
G_{T_i} &= \texttt{All guesses from tool } T_i \ (i \in [1,5]) \\
S &= \{\texttt{000webhost}, \texttt{eHarmony}, \texttt{Linkedin}, \texttt{pwndSub10m}, \texttt{Yahoo}\} \\
S_k &= \texttt{All entries in the testing set } k \ (k \in [1,5])
\end{aligned}
$$

We define the matches from each tool for a testing list S_k as the following:

$$M_{T_i S_k} = G_{T_i} \cap S_k$$

To retrieve the unique matches for a tool on the list S_k we use:

$$U_{T_i S_k} = M_{T_i S_k} \setminus \bigcup_{\substack{j=1 \\ j \neq i}}^{5} M_{T_j S_k}$$

PGF saves the founds which each tool gets on the evaluation, but it does not save in which order the passwords were found (it uses a python dictionary internally). Therefore we can not determine $U_{T_i S_k}$ from our previous evaluations. To have some data points in between the start and the maximum number of guesses, we ran multiple evaluations with having a changed number of maximum guesses to have the number of unique founds for certain steps. We set the following positions (in millions):

$$X = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$$

As these steps had to be performed by each tool, this resulted in more than 500 evaluations to be executed. However, it allowed us to retrieve the unique founds for each tool at each step. When we look at the normal results again, as shown in Figure 6.12, we see that Óðinn can not keep up with the other guessers in the number of matches against the

while testing set. It also is still similar when we look at the number of unique matches in Figure 6.13. This comparison still is not fair, as if the total number of matches is higher, it is nearly sure that also the unique count will be higher. Therefore, we compare the number of unique matches with respect to of the number of total matches:

$$U_{T_i S_k rel} = \frac{U_{T_i}}{M_{T_i}}$$

Figure 6.14 shows the relative comparison of unique matches. PCFG is still clearly leading, but we see that the other tools are close together with Óðinn getting approximately ten percent more unique matches. On the other lists, we can observe a similar outcome, for example, as shown for Linkedin in Figure 6.15.

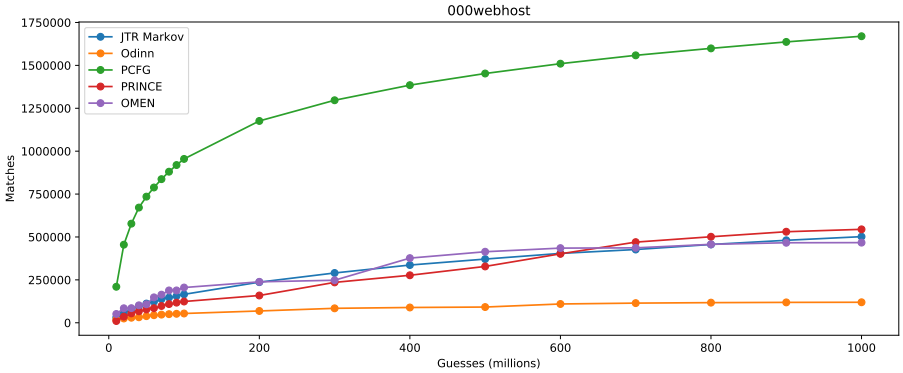


Figure 6.12: Evaluation against the 000webhost list, with a maximum of 1 billion guesses showing the number of matches for each tool.

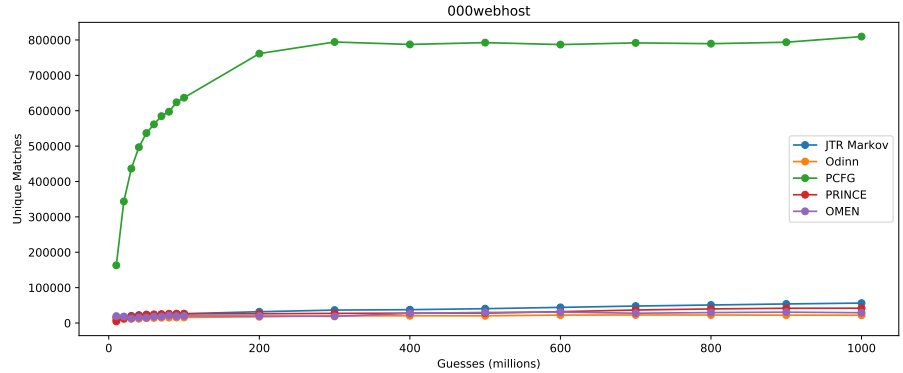


Figure 6.13: Evaluation against the 000webhost list, with a maximum of 1 billion guesses showing the number of unique matches for each tool.

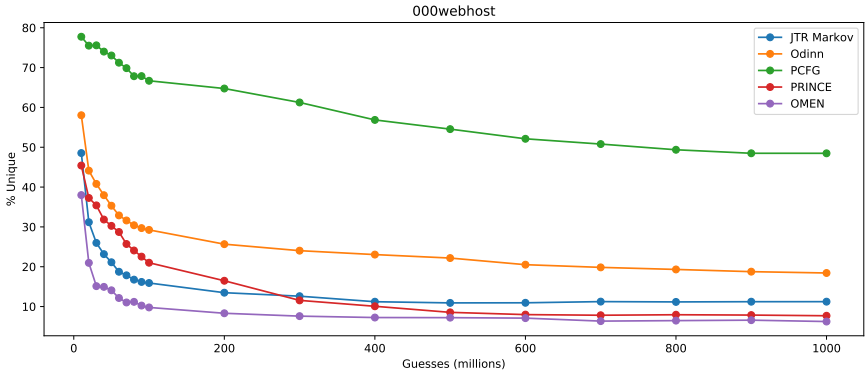


Figure 6.14: Evaluation against the 000webhost list, with a maximum of 1 billion guesses showing the percentage of matches which were unique.

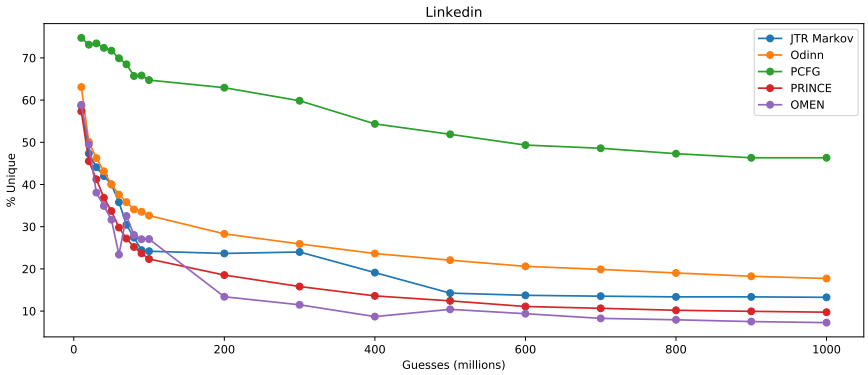


Figure 6.15: Evaluation against the Linkedin list, with a maximum of 1 billion guesses showing the percentage of matches which were unique.

6.3.2 Comparison on Left Hashes

As mentioned in the previous section, we were interested in comparing the tools not only against full test lists but also against more difficult passwords. For this we took the Hashes.org left 32hex list[42] (as of July 15th 2019) as evaluation list. It is ideal to be used as it contains a large number of uncracked hashes collected over multiple years originating from many different sources. Typically, easy passwords are quickly found and do not remain in the left list; therefore, in general, the passwords are more complex and difficult to recover.

We only test the candidates with MD5 (as the hashes in the left list not all necessarily need to be MD5), the list contains 27'643'254 unique hashes. We evaluated smaller guessed lists using rules in order to optimize the GPU usage[43]. All tools were trained on the Hashes.org found list 2015[44], for Óðinn we used the following settings for the generation: 10'000 outerloops, 100 top, three fragments, non-iterative. The guessing lists from all the tools contain 98'761'423 (number of entries in the Óðinn list, enforced on the other tools)

[42] https://hashes.org/left.php
[43] https://hashcat.net/forum/thread-5287-post-28724.html#pid28724
[44] https://hashes.org/left.php

Table 6.5: Number of founds when running guessing against the Hashes.org left list and examples of recovered passwords.

Óðinn	**OMEN**	**PRINCE**	**PCFG**
934 found	**452 found**	**208 found**	**474 founds**
brajaniec	glee1233	ka289220@	adiwaj411
roughh11	carytfishing1	@y05135116	PARTgo25
hanyang@05	trapetsare01	fradio13	love2996
joeyandus719	ted10049!	@>000world	brynn0924
puppyfood.0906	shaunta28	jaia05	prettygurl o5
PaSS18is89	silver2460	sd8125	1005@dsu
gururam@78	shan20110102yc	157930517	infinite04
1215mk	cindy2067	wwjd0707	Qweasd098
hlen$#13	prispimmano	ms2032	y9598!@#
MANI11JUNE	fla3mar2her1	!a1279610	vivian5p11!

and we used the rockyou-30000.rule[45] containing the rules. We were not able to use JTR Markov for this evaluation as it was not able to train on the large Hashes.org 2015 list.

In Table 6.5, we see the number of matches found against the left list by each tested tool together with some examples. Óðinn got the most founds, which shows that it is more suited against difficult lists or such where the simple passwords were already removed.

6.3.3 Generated Rules

From the analysis of wordlists, Óðinn can provide all the fragments it found. With the frequency count, it is possible to have an ordered list of them, with a decreasing order of probability. When recovering long passwords, it can often be observed that rules which prepended certain strings to the candidates were effective. An example of such a prepend rule could be[46]:

```
^o^l^l^e^h
```

This rule, for example, would prepend *hello* to the word *password* resulting in the candidate *hellopassword*. So we translated the list of fragments we extracted from Rockyou with Óðinn into prepending rules. This file contains 752'627 rules in total. In most cases, it is not needed to have that many rules, but as this rule file is in decreasing probability for the fragments, the number of needed rules can just be extracted with *head* to get the N most probable prepend rules. The ten most probable prepends are shown in Listing 6.12.

[45] https://github.com/hashcat/hashcat/blob/master/rules/rockyou-30000.rule

[46] Using the syntax of Hashcat's rule engine: https://hashcat.net/wiki/doku.php?id=rule_based_attack. The Function `^x` denotes prepending the character x to the input. Functions can be stacked and will be executed in order from left to right.

Listing 6.12: Most probable prepends from Rockyou.

^o^t	(to)	^e^m	(me)
^e^v^o^l	(love)	^n^a	(an)
^y^m	(my)	^d^n^a	(and)
^s^i	(is)	^e^v^o^l^i	(ilove)
^y^b^a^b	(baby)	^e^h^t	(the)

We used the same left hashlist from Hashes.org as in the previous section to evaluate how many and what kind of passwords we can recover with the help of these prepend rules. We used the Rockyou list together with all the prepend rules. With the run, 1'052 passwords were recovered in total. Table 6.6 shows examples of these found passwords and which prepend rule was used for this candidate.

If we now use a wordlist which contains more complex passwords than Rockyou, we can use the prepend rules to recover even longer passwords. Listing 6.13 shows examples of passwords recovered from the Hashes.org left list when using the Óðinn generated wordlist (used in Section 6.3.2) in combination with the prepend rules. We can see that the passwords consist of multiple combined elements (words, numbers, mixed case). This outcome shows again that the combination of fragments can allow the reconstruction of longer and more complex passwords.

Listing 6.13: Example founds from using the Óðinn generated wordlist with prepend rules.

kangdongju1-	yumiesme12,
minqueen123!	pharmws10/11
nhvarsity#6	cks124
lilytrewiscool	wtf85bwin123
jarebomar3@	redstonejr10/
CIA^^1st	GANGSTERboy12@
LOVELYteddybear10@	sriroy22@
babumukkaden5	quemaunanuve
yurotube123.	takrlife.7
ncrcdancerox	diflee07!

6.4 Preprocessor Integration

Contrary to the initial plan, the Hashtopolis integration does not strictly require a guesser to have the *–skip* and *–limit* flags to allow them to be used. This way, guessers as OMEN and PCFG can be used without waiting for the developers to offer the necessary flags. Though, it needs to be considered that this only is useful to use if attacking slower hash algorithms. Otherwise, the guesser in most cases is not fast enough to quickly catch up to the position where the specific chunk starts. This would lead to a high overhead when agents have to wait a long time on every chunk until the position to start is reached.

Table 6.6: Example of found passwords with the prepend rules and their accordingly used rule

Wordlist Entry	**Rule**	**Found Candidate**
kim97	^t ^r ^m ^s	smrtkim97
buffet1	^R ^G ^K	KGRbuffet1
VAMPIRIA	^N ^E ^W ^R ^E	ERWENVAMPIRIA
ash145	^A ^T ^A ^L	LATAash145
faruk	^p ^a ^l ^o ^d	dolapfaruk
tpark7	^h ^g ^a ^p	paghtpark7
brown14	^U ^S ^W	WSUbrown14
bottega	^a ^r ^o ^l ^f ^a ^l ^l ^e ^d	dellaflorabottega
pennylin	^A ^N ^N ^I ^T	TINNApennylin
manju56	^t ^e ^e ^n ^v ^a ^n	navneetmanju56

Especially for fast hashes, it is recommended to use a rule file as an amplifier (e.g. the prepend rules described in the previous section). On the one hand, this allows using the GPU resources more efficiently and on the other hand, this way, the guesser has to generate fewer candidates for a chunk. This again makes the overhead smaller when using the same number of chunks.

7 Conclusion

In this thesis, we developed Óðinn, a tool to analyze large wordlists, visualize the outcome, and guess new passwords. It can be used for a variety of use-cases and is easily extensible for further purposes by adding more modules. Additionally, we provided an extension to Hashtopolis by integrating preprocessors to be used in distributed password recovery tasks. In this chapter, we discuss the evaluation outcomes and discuss future work.

7.1 Results Discussion

We showed that Óðinn can split passwords into their basic fragments and find their semantic meaning. The quality of the outcome depends heavily on the used dictionaries for the classification as well as on the source of the wordlist. Generally, it is challenging to have a generic approach to handle this language-independently, as we have to assume that wordlists are built containing mixed language passwords. Having specific dictionaries for different languages may improve this, but it would also impose more sophisticated analysis to be done and introduce additional false hits.

A similar issue appears when trying to classify a password into languages. Beside using specific characters which occur only in very few languages, it is nearly impossible to classify based on a single password. Therefore, the language detection module in Óðinn can only provide minimal information from the analysis.

The visualization of the analysis results allows a quick overview of a wordlist. Odd content in a wordlist can be discovered and more deeply analyzed. It is essential to support the creation of better wordlists and removing junk lines in order to have well suited guesses. Further, it is the first analytical approach to improve wordlists without manual inspection.

Using the guesser modules of Óðinn, we were able to recover long passwords consisting of multiple fragments. We compared Óðinn to other state-of-the-art guessers and it shows that Óðinn is well suited to guess against hashes where other attacks were already exhausted, to recover longer and more complex passwords/passphrases. In such cases, Óðinn outperforms the other benchmarked guessers. Additionally, we showed that based on analysis results from Óðinn, we can create rules which also are effective in recovering difficult passwords.

7.2 Future Work

Since the beginning, we developed Óðinn to be flexible to be extended and changed easily. There are possible improvements and additions for the future as listed below. Further, there still remain some open research questions.

Frequency Dictionary Improve the quality of the frequency dictionary by adding more words and trying to reduce the amount of unwanted content. The detection of concatenated words and filtering out invalid words remains a challenge.

Semantic Analysis More dictionaries with classes can be added, as well as extending the current existing dictionaries by more words. This will allow to classify more of the fragments and improve the quality of the semantic analysis. Additionally, other languages can be included as well, to classify non-English fragments into their correct class. The extension and creation of dictionaries is a tedious and time consuming task but not challenging.

To improve the classifications with WordNet, the grouping can be improved/changed to use better parental synsets and use a faster approach to reduce the computation required. This holds some questions to solve. What are the ideal broader groups to be used, and is there an algorithmic approach to solve this (instead of the used empirical approach)?

Analysis Modules To explore the case structures in the fragments, we can analyze the distribution of the positions in the passwords with capitalized letters. While the implementation of such a module is straight-forward, it needs to be researched if this gives interesting insights and what can be concluded from such results.

Visualization Modules Visualize additional analysis modules, for example, as the abovementioned one and include them in the report. Also, add more control to how the plots should be rendered, which appear in the report. This purely is implementation work to be done.

Improve Speed In order to improve the speed of the analysis, the result merge functionality can be extended to be able to aggregate the results from multiple machines. This ability will allow distributing the analysis over multiple machines instead of over multiple threads only. This requires to introduce an additional layer in the structure of Óðinn, but technically it is ready to be implemented.

Guesser Splitting Find a better way to split the guessing task into a keyspace. Currently jumping to a skip position needs some processing time. Ideally, the keyspace should be created in a way that all positions can be reached quickly and that the workload is distributed (mostly) equally across the keyspace. This challenge is an open research question, which is also still unsolved in other guessers.

Extend Report Extend the content of the report to provide more detailed results of the analysis and provide additional textual support to plots and tables. While being implementation work only, identifying the exact required additional content is a challenge in itself.

Bibliography

[1] Hashcat advanced password recovery. https://hashcat.net/hashcat. Accessed: 2019-01-15.

[2] John the Ripper fast password cracker. https://github.com/magnumripper/JohnTheRipper. Accessed: 2019-01-15.

[3] Password analysis and cracking kit. https://github.com/iphelix/pack. Accessed: 2019-02-18.

[4] PCFG Cracker: Probabilistic context free grammar password research project. https://github.com/lakiw/pcfg_cracker. Accessed: 2019-01-15.

[5] PRINCE: Standalone password candidate generator using the prince algorithm. https://github.com/hashcat/princeprocessor. Accessed: 2019-01-15.

[6] Joseph Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. *Proceedings - IEEE Symposium on Security and Privacy*, (Section VII):538–552, 2012. ISSN 10816011. doi: 10.1109/SP.2012.49.

[7] Joseph Bonneau and Ekaterina Shutova. Linguistic properties of multi-word passphrases. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7398 LNCS:1–12, 2012. ISSN 03029743. doi: 10.1007/978-3-642-34638-5_1.

[8] Joseph A. Cazier and B. Dawn Medlin. Password security: An empirical investigation into e-commerce passwords and their crack times. *Information Systems Security*, 15(6): 45–55, 2006. ISSN 1065898X. doi: 10.1080/10658980601051318.

[9] Hsien Cheng Chou, Hung Chang Lee, Hwan Jeu Yu, Fei Pei Lai, Kuo Hsuan Huang, and Chih Wen Hsueh. Password cracking based on learned patterns from disclosed passwords. *International Journal of Innovative Computing, Information and Control*, 9(2):821–839, 2013. ISSN 13494198.

[10] Markus Dürmuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, and Abdelberi Chaabane. Omen: Faster password guessing using an ordered markov enumerator. In *International Symposium on Engineering Secure Software and Systems*, pages 119–132. Springer, 2015.

[11] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. *Proceedings of the 16th international conference on World Wide Web - WWW '07*, page 657, 2007. ISSN 08963207. doi: 10.1145/1242572.1242661.

[12] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan: A deep learning approach for password guessing. In *International Conference on Applied Cryptography and Network Security*, pages 217–237. Springer, 2019.

[13] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan: A deep learning approach for password guessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 217–237, Cham, 2019. Springer International Publishing. ISBN 978-3-030-21568-2.

[14] Shiva Houshmand, Sudhir Aggarwal, and Randy Flood. Next gen pcfg password cracking. *IEEE Transactions on Information Forensics and Security*, 10(8):1776–1791, 2015.

[15] Shouling Ji, Shukun Yang, Ting Wang, Changchang Liu, Wei-han Lee, and Raheem Beyah. PARS: A Uniform and Open-source Password Analysis and Research System. *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 321–330, 2015. doi: 10.1145/2818000.2818018.

[16] Ashwini Rao, Birendra Jha, and Gananand Kini. Effect of grammar on security of long passwords. *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13*, page 317, 2013. doi: 10.1145/2435349.2435395.

[17] Utku Sen. Generating personalized wordlists with natural language processing by analyzing tweets. https://github.com/tearsecurity/rhodiola/blob/master/rhodiola_paper.pdf, 2019.

[18] Blase Ur, Saranga Komanduri, Richard Shay, Stephanos Matsumoto, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Michelle L Mazurek, and Timothy Vidas. Poster: The art of password creation. In *Proc. IEEE Symposium on Security and Privacy*, 2013.

[19] Blase Ur, Jonathan Bees, Richard Shay, Nicolas Christin, Fumiko Noma, Sean Segreti, Lujo Bauer, and Lorrie Faith Crano. " I Added '!' at the End to Make It Secure ": Observing Password Creation in the Lab. *Proceedings of the eleventh Symposium On Usable Privacy and Security - SOUPS' 15*, pages 123–140, 2015.

[20] Rafael Veras, Julie Thorpe, and Christopher Collins. Visualizing semantics in passwords: The Role of Dates. *Proceedings of the Ninth International Symposium on Visualization for Cyber Security - VizSec '12*, (1):88–95, 2012. doi: 10.1145/2379690.2379702.

[21] Rafael Veras, Christopher Collins, and Julie Thorpe. On the Semantic Patterns of Passwords and their Security Impact. In *Proceedings 2014 Network and Distributed System Security Symposium*, 2014. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23103.

[22] Chun Wang, Steve T K Jan, Hang Hu, Douglas Bossart, and Gang Wang. The Next Domino to Fall : Empirical Analysis of User Passwords across Online Services.

[23] Ding Wang, Haibo Cheng, Ping Wang, Xinyi Huang, and Gaopeng Jian. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security*, 12(11):2776–2791, 2017.

[24] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 391–405. IEEE, 2009.

A

Odinn Usage

Information about all the flags usable with Óðinn can be found when calling the help function as shown in Listing A.1.

Listing A.1: Calling Óðinn help function.

```
$ python3 __main__.py --help
```

To list which modules are available, the list commands can be used, as shown in Listing A.2.

Listing A.2: Listing available modules.

```
$ python3 __main__.py --list-analysis-modules
advancedMasks
charsetDistribution
fragmentCounter
...

$ python3 __main__.py --list-visualize-modules
latexSummary
latexFilter
charsetDistribution
...

$ python3 __main__.py --list-filter-modules
emailFilter
lengthFilter
fragmentCountLengthFilter
...
```

A.1 Analysis

The definition of which modules should be used for the analysis is done in a JSON configuration file. It can not be done on the command line directly, as there would be no

simple way to specify chains of modules together with standard modules. Listing A.3 shows the default configuration which is delivered with Óðinn. The chained modules are ordered in a sub-list and module components can be used multiple times (e.g. the simpleSplitter module).

Listing A.3: Default analysis config.

```
[
  "lengthDistribution",
  "charsetDistribution",
  "advancedMasks",
  "languageDetection",
  [
    "simpleSplitter",
    "fragmentCounter"
  ],
  [
    "simpleSplitter",
    "fullSplitter",
    "simpleSemantic"
  ]
]
```

An analysis can then be executed by calling the command as shown in Listing A.4. The file *analysis-config.json* contains the module configuration as described above, and the *-o* flag specifies where the results file should be saved and *-w* sets the wordlist to be analyzed.

Listing A.4: Executing analysis with Óðinn.

```
$ python3 __main__.py -a -A analysis-config.json -o results.json \
    -w wordlist.txt
```

Óðinn by default will load the lines in batches of 100'000 entries to be analyzed. Depending on the hardware running on, this value can be changed by using the *–batch-size <n>* parameter. Also depending on the hardware, the number of threads can be set with using *–thread <n>*. The difference between the threads and low-threads is explained in Chapter 5.

A.2 Filtering

By default, when running analysis with Óðinn, all filter modules are active, but they are not set to filter anything. Each filter module has its specific flags, which are also listed in the help function. Specifying the filters is done with the *-F* flag with listing all desired modules separated by a comma, e.g. as in Listing A.5.

Listing A.5: Selecting filter modules example.

```
$ python3 __main__.py -a -A analysis-config.json -o results.json \
    -w wordlist.txt -F emailFilter,lengthFilter --skip-emails \
    --min-length 5 --max-length 10
```

If needed, the entries which are filtered out by the modules can be written to file by setting a path with *–write-filtered <path>*. In case there should only be filtered, and no analysis should be executed, the *–filter-only* flag can be set. This can be useful to extract all entries from a wordlist matching a specific setting. An example is shown in Listing A.6 where the entries should be written to *filtered.txt* which match the specific mask *?u?l?l?l?l?l*. Listing A.7 shows some example output from using this mask filter.

Listing A.6: Example of filtering only.

```
$ python3 __main__.py -a -w wordlist.txt -F maskFilter \
    --write-filtered filtered.txt --filter-mask '?u?l?l?l?l?l'
```

Listing A.7: Filtered entries matching from the command in Listing A.6

Quincy	Trunks
Simple	Turtle
Speedy	Alisha

A.3 Generate Report

The visualization modules can create LaTeX code which can then be compiled to a PDF report. To compile the LaTeX code, the distribution needs to be installed, so that *pdflatex* can be called on the command line. If not possible, the LaTeX code can be copied and be compiled on another system later (call the report generation without the *–render-latex* flag). Using the *-V* flag it can be selected, which modules should be included in the report, if not specified, all existing visualization modules will be called. Listing A.8 shows an example report generation call from a results file. The argument *–latex-path <path>* can be used to set a location to save the report to (instead of the default *report* folder). All flags which are used to set plotting specific settings as described in Section A.4 can be used as well, and they will be applied to the plots in the report.

Listing A.8: Example of generating a report.

```
$ python3 __main__.py -v --full-report -R -r results.json -V \
    latexSummary,charsetDistribution,lengthDistribution \
    --generate-latex
```

A.4 Show Plots

Plots can either can be generated and saved to disk or they can be directly displayed in windows to explore them dynamically. Plots will always be saved in PDF and SVG format in the *plots* folder. Listing A.9 shows an example where just two plots are generated and saved, without displaying anything. Listing A.10 shows the same plot generation with the addition that it will open two windows with the two selected plots, offering the ability to move, zoom, and save at additional locations.

Listing A.9: Example of generating a report.

```
$ python3 __main__.py -v -V charsetDistribution,lengthDistribution
```

Listing A.10: Example of generating a report.

```
$ python3 __main__.py -v -V charsetDistribution,lengthDistribution\
    --show-plots
```

A.5 Bi-Fragment Analysis

The Bi-Fragment analysis is not set in the default module config file, as typically this results in large json files. So it should only be run if needed. An example of the command for this run is shown in Listing A.11.

Listing A.11: Example of bi-fragment analysis.

```
$ python3 __main__.py -a -A analysis-config-bifrag.json \
    -w rockyou.txt
```

A.6 Bi-Fragment Guessing

The Bi-Fragment analysis is not set in the default module config file, as typically this results in large JSON files. So it should only be run if needed. An example of the command for this run is shown in Listing A.11.

–**guesser-bifragment-max-top** Sets the number of possible right-hand elements of the current state should be explored. Increasing this number can lead to an exponentially growing amount of candidates.

–**guesser-bifragment-max-steps** Sets the maximal number of fragments to be combined. It typically is set to 3 or 4. Higher numbers will lead to too many candidates.

–**guesser-bifragment-max-outerloops** Sets how many right-hand elements from the *START* element should be taken. It sets how many starting fragments should be used.

–**guesser-bifragment-exact-only** Set this flag if only combinations with the exact number of fragments as configured in the max-steps value should be printed (and none with fewer fragments).

–**guesser-bifragment-skip-nonend** Set if only combinations should be printed, where the last fragment is followed by an *END* element.

–**guesser-bifragment-skip-singlechar** Set if fragments with length one should be skipped.

–**guesser-bifragment-non-iterative** Set that the iterative mode should not be used. This will generate candidates faster, but the order of elements may not be with decreasing frequency.

An example using some of the flags is shown in Listing A.12. The values carefully need to be adjusted as some small changes could mean a massive increase of candidates.

Listing A.12: Example of bi-fragment analysis.

```
$ python3 __main__.py -g -G biFragmentMixer -r results.json \
    --guesser-bifragment-max-top 50 --guesser-bifragment-max-steps\
    3 --guesser-bifragment-max-outerloops 50000 \
    --guesser-bifragment-skip-singlechar
```

In the special case where the results file from the Bi-Fragment analysis gets fairly large, the result can be converted into a CSV file which allows the guesser to sequentially load the file without having to keep it in memory. Listing A.13 shows how an existing results file is used to create a CSV file and Listing A.14 shows an example how the guesser then can be started with loading the data from the CSV file (using the *–guesser-bifragment-load-file* argument). For the results file, only a dummy input is taken, as the argument is required.

Listing A.13: Example of bi-fragment analysis.

```
$ python3 __main__.py --util --util-export-bifrag results.json \
    > results.csv
```

Listing A.14: Example of bi-fragment analysis.

```
$ python3 __main__.py -g -G biFragmentMixer -r dummy.json \
    --guesser-bifragment-max-top 50 --guesser-bifragment-max-steps\
    3 --guesser-bifragment-max-outerloops 50000 \
    --guesser-bifragment-skip-singlechar \
    --guesser-bifragment-load-file results.csv
```

A.7 Semantic Guessing

The semantic results of an analysis can be used to guess candidates based on the structures which were most common in the analysis. The module only takes combinations from the results where all elements were classified (no x fragments). Listing A.15 shows an example where the most common ten combinations are used to generate guesses. The limit flags set the maximum amount of candidates for a class can maximal be taken to avoid too many candidates from certain classes.

Listing A.15: Example of bi-fragment analysis.

```
$ python3 __main__.py -g -G semantic -r results.json \
    --semantic-max-elements 10 --semantic-baselist-limit 100 \
    --semantic-wordnet-limit 100
```

B

Additional Plots

This appendix contains additional plots from the guesser benchmarks which were not used in the thesis text.

B.1 Analysis Plots

In this section the additional plots from the analysis results of the Hashes.org 2015 list, the Hashes.org 2015 junk list and Rockyou list are shown.

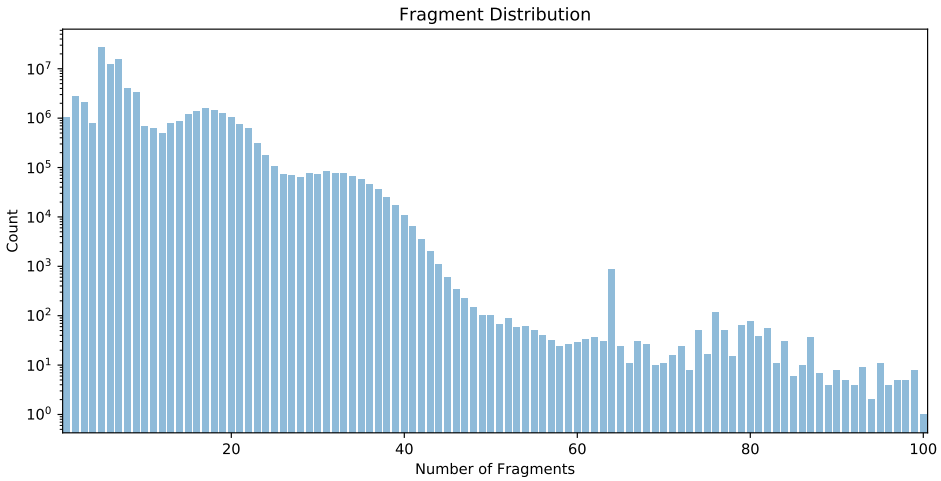


Figure B.1: Fragment distribution of the Hashes.org 2015 junk list.

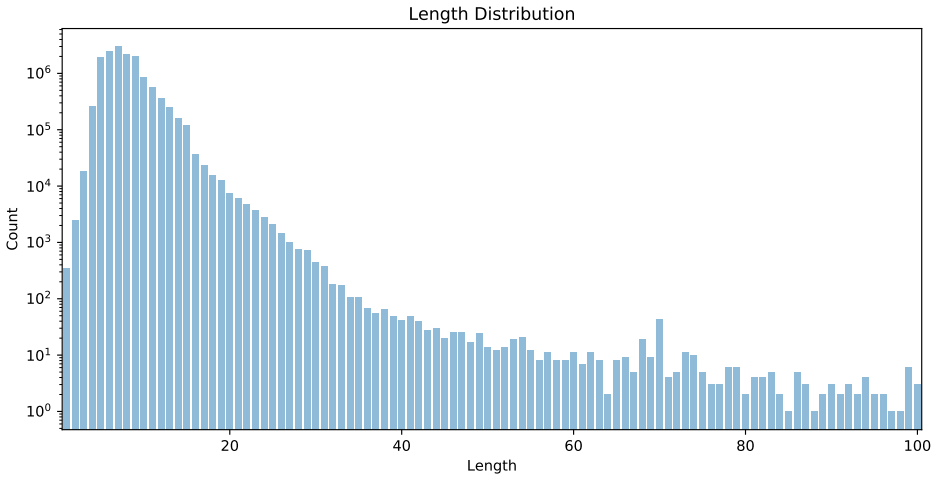


Figure B.2: Length distribution from the Rockyou list.

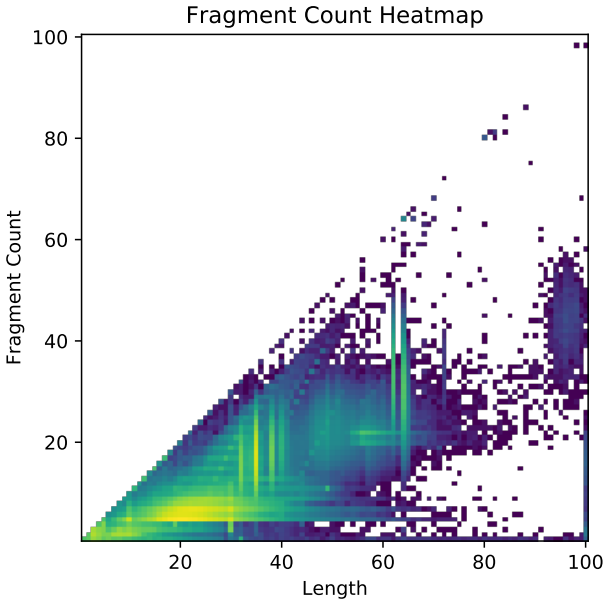


Figure B.3: Fragment Heatmap of the Hashes.org 2015 junk list without the extended axes.

B.2 Comparison with 100m Guesses

This section shows the additional plots from the comparison of the guessers with a maximum of 100 million guesses.

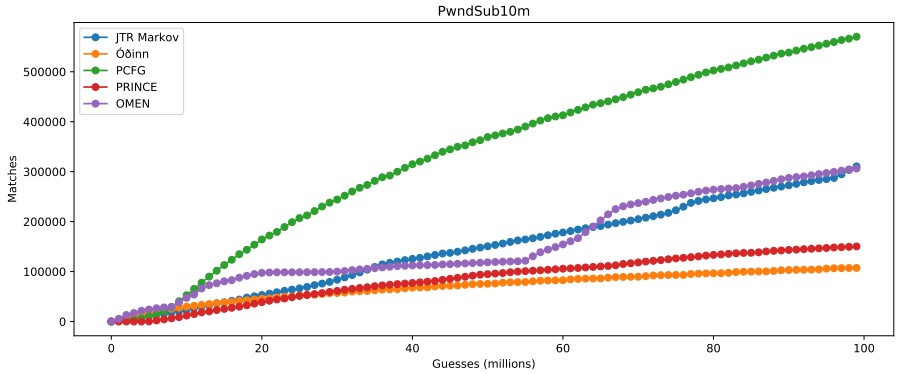


Figure B.4: Guessers against the PwndSub10m testing list with 100 million guesses.

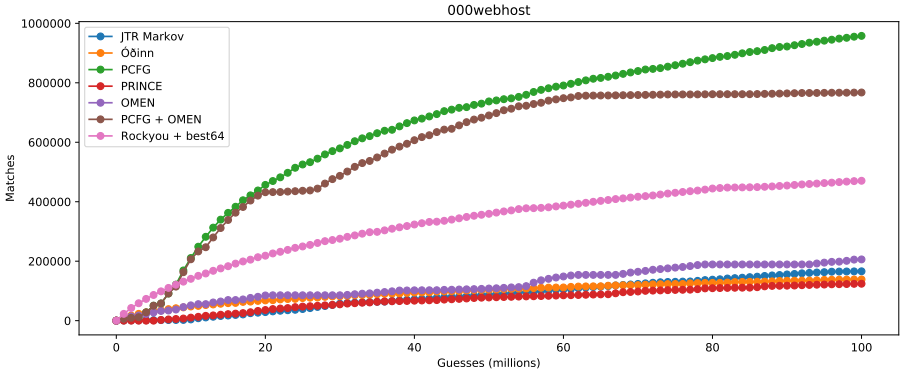


Figure B.5: Guessers against the 000webhost testing list with 100 million guesses, including using Rockyou + best64 and PCFG with OMEN.

B.3 Comparison with Unique Matches

In this section, the remaining plots for the comparison up to 1000 million guesses and the comparison with unique founds are shown.

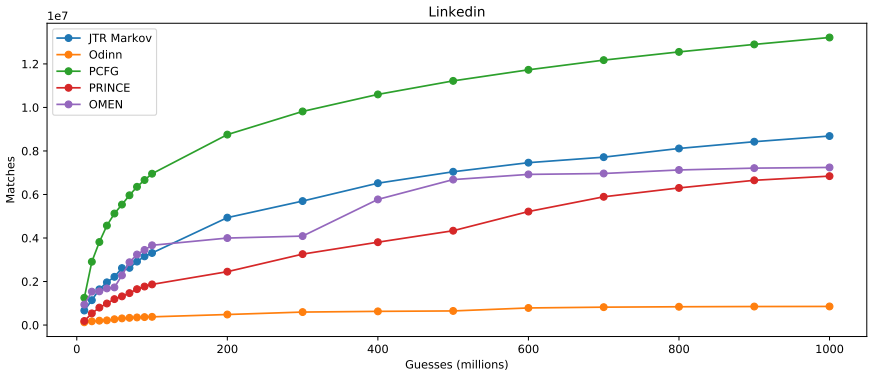


Figure B.6: Guesser comparison against the Linkedin testing list with 1000 million guesses.

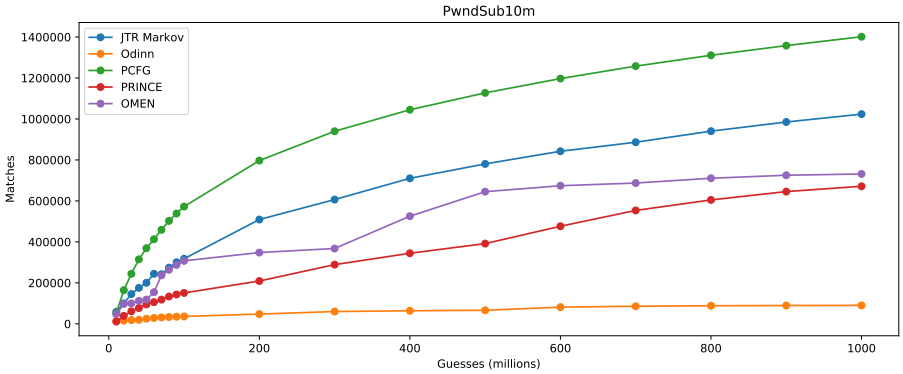


Figure B.7: Guesser comparison against the PwndSub10m testing list with 1000 million guesses.

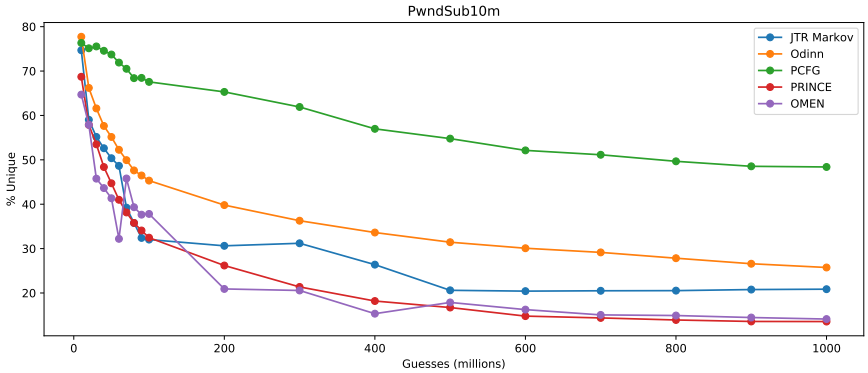


Figure B.8: Guesser comparison with unique founds against the PwndSub10m testing list with 1000 million guesses.

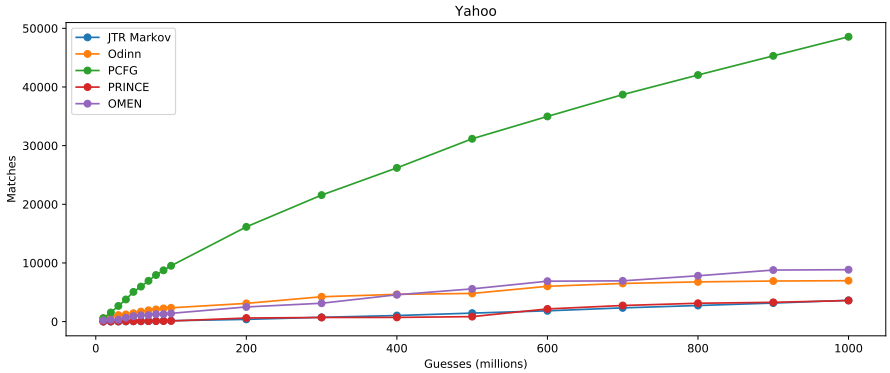


Figure B.9: Guesser comparison against the Yahoo testing list with 1000 million guesses.

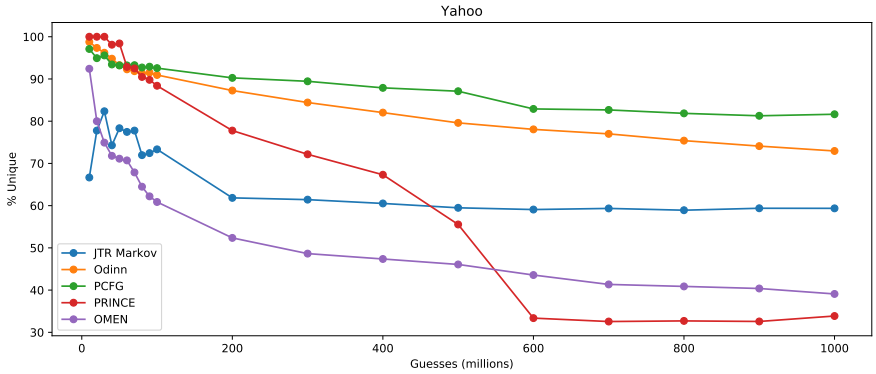


Figure B.10: Guesser comparison with unique founds against the Yahoo testing list with 1000 million guesses.