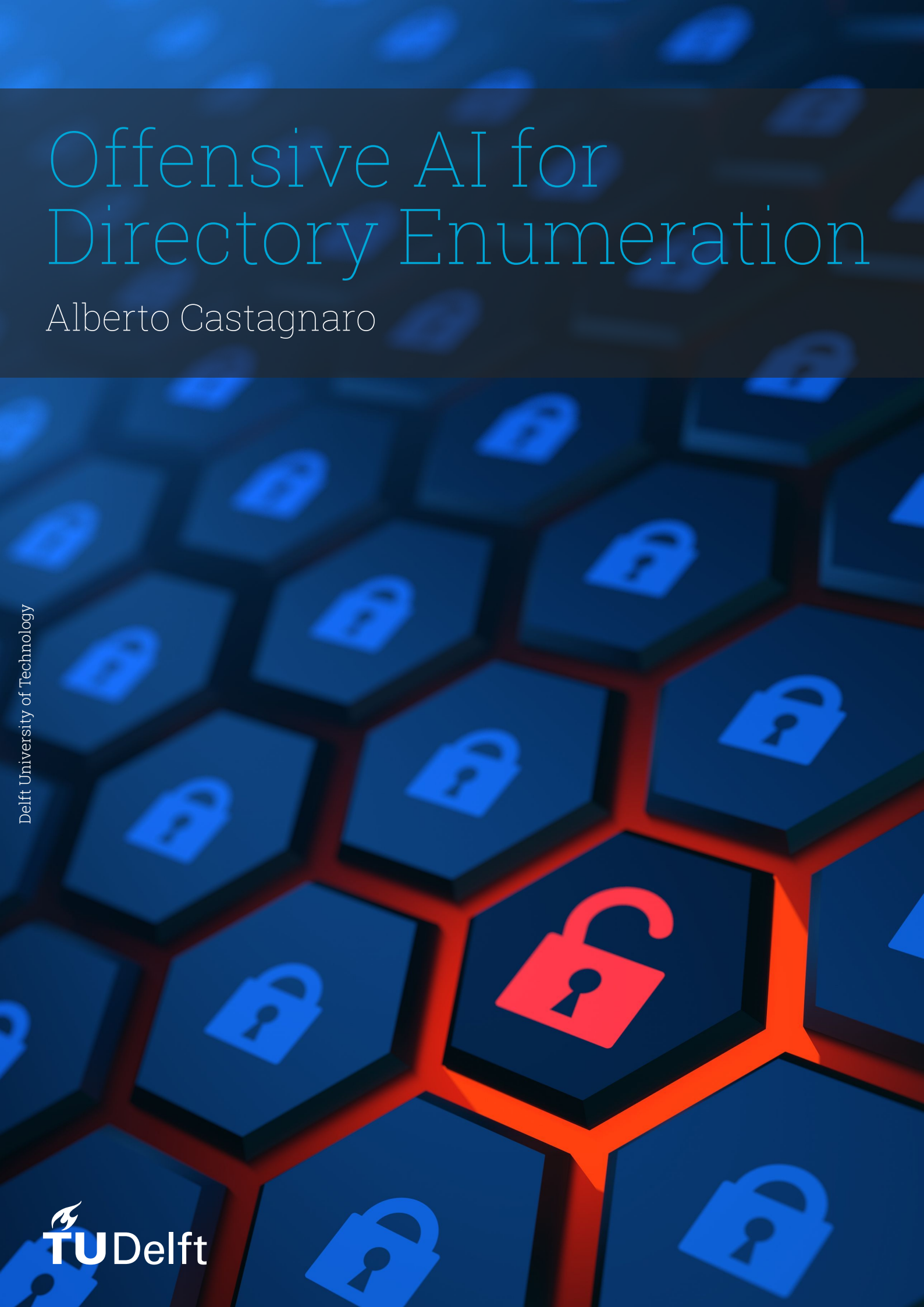
Offensive AI for Directory Enumeration

Alberto Castagnaro



Delft University of Technology

TU Delft

Offensive AI for Directory Enumeration

by

Alberto Castagnaro

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday May 22, 2024 at 10:00 AM.

Alberto Castagnaro: 5861489
Project duration: October 2, 2023 – May 22, 2024
Thesis committee:

Prof. Mauro Conti,	Full professor	TU Delft, supervisor
Prof. Georgios Smaragdakis,	Full Professor,	TU Delft
Prof. Jie Yang,	Assistant Professor,	TU Delft
Dr. Luca Pajola,	External Member,	Daily supervisor

An electronic version of this thesis is available at
http://repository.tudelft.nl/.

TU Delft

Acknowledgments

This thesis project marks the end of my academic journey in the Master's program in Computer Science, a journey filled with learning, growth, and invaluable experiences.

I would like to express my deepest gratitude to all those who have contributed to the successful completion of this thesis project.

First and foremost, I am deeply indebted to my supervisors, Prof. Mauro Conti and Dr. Luca Pajola. Your unwavering support and guidance, the opportunities you gave me to explore new topics I was interested in, and your expertise in supporting me throughout a project I am proud of have been incredibly valuable.

I would also like to express my heartfelt appreciation to my family and friends for their support throughout this journey.

To my mom, thank you for giving me this opportunity and continually encouraging me to pursue my dreams.

To my dad, who, even though he is no longer with us, continues to be an endless source of inspiration, thank you for teaching me to smile even in the face of adversity.

To my girlfriend, Eleonora, thank you for unconditional support even in the toughest times and for always being there for me.

To my housemates, Pietro, Nicola and Alessio, for sharing this incredible journey on a daily basis.

To my friends I met here in the Netherlands: Pavlos, Jorge, Rodrigo, Carlo, Ziad, Anastasis, and Ansh. Thank you for the moments and memories we shared and for making me relax and smile when needed.

To my Italian friends, F5Z, who every time I came home made me feel like I never left.

Alberto Castagnaro

Abstract

Web Vulnerability Assessment and Penetration Testing (Web VAPT) is an important cybersecurity practice that thoroughly examines web applications to uncover possible vulnerabilities. These vulnerabilities represent potential security gaps that could severely compromise the web applications' integrity and functionality if exploited by malicious entities. One of the attacks employed in the Web VAPT process is the **Directory Brute-Forcing Attack**. This attack aims to identify hidden directories and files not adequately secured in a web application that contain sensitive information or critical functionalities. The attack methodology involves sending many requests of possible directories or files to the target web application, where brute-force generation of requests is performed using a wordlist. Due to its brute-force nature, this attack methodology often results in enormous quantities of requests sent for a small amount of successful discoveries.

With AI's quick progress and diffusion, the paradigm of Offensive AI emerges, where AI-based technologies are employed in traditional cyber attacks to make them more sophisticated and effective. This research explores whether AI can enhance the standard directory enumeration process. We propose two novel attack methodologies for performing directory brute-forcing attacks that leverage probability and Language Models (LM). Our experiments - conducted on a testbed consisting of around 1 million URLs from various domains of web applications (academic institutions, hospitals, government agencies, and business corporations) - demonstrate the superiority of our approaches over the standard brute-force attacks. In particular, the LM-based attack results in an average discoveries increase of 969%, and the probabilistic attack is more efficient at sending successful requests in the early stages of attacks in more than 94% of cases.

Contents

1

Introduction

1.1. Context

The word *hacking*, in its broadest sense, is the act of altering or manipulating a system's features to accomplish an objective beyond the original intent of the creator. In cybersecurity, hacking is often associated with unlawful cyber activities that aim to compromise devices and networks, gain unauthorized access, or disrupt systems. These activities usually also target critical infrastructure or sensitive information and are carried out by various actors, from cyber criminals to hacktivists and even entire Nation-state actors.

Cyber attacks have increased in frequency and volume in recent years, reaching an unprecedented impact. It is predicted that global business damage caused by cybercrime will reach USD 10.5 trillion annually by 2025, with an annual increase of 15% [29]. One of the leading causes is new threat scenarios where hackers use increasingly sophisticated techniques to conduct cyberattacks using artificial intelligence, machine learning and other technologies [41]. Some of the most common targets of these attacks are web applications.

However, the hacking process may represent a valuable resource when performed ethically. *Ethical hacking*, also known as *Penetration testing*, *Red Teaming* or *white-hat hacking*, involves skilled professionals that have the goal of identifying vulnerabilities and weaknesses in computer systems, networks and application so they can be fixed before malicious actors can exploit them.

A penetration test, which is a simulated cyber attack against a system, involves different stages that constitute a cyber attack from *Reconnaissance*, where the intelligence gathering of a target takes place, to *Post-Exploitation*, where the test has gained access to the target and try to analyze and exploit it further.

In web applications, the *Reconnaissance* phase of a penetration test may involve *Directory enumeration*. This process consists in discovering directories, files and web paths that may be hidden and contain vulnerabilities, sensitive information or critical functionalities. Discovery attacks, such as directory brute-forcing or *dirbusting* attacks, can be employed for this goal. Specifically, the directory brute-forcing attack involves sending multiple HTTP requests to a target web application using a wordlist to construct new URLs. Due to its nature, this type of attack can be time-consuming and may not always yield useful results.

1.1.1. Artificial Intelligence

Artificial Intelligence (AI) has emerged as a transformative force in the modern world, revolutionizing industries and reshaping how we live and work. AI is projected to have an economic impact on the global economy in 2030 of up to $15.7 trillion USD [11]. Among the most significant advancements of AI, 2023 has been the breakout year for generative AI, with a massive spread of generative AI tools that have been bringing several companies to the adoption and daily use of generative AI among their employees [9].

AI's impact is profound in cybersecurity. AI-driven security systems can predict and thwart cyber threats with unprecedented accuracy, adapting to new risks dynamically. AI has found applications in many fields of cybersecurity, from threat hunting [12] to response and mitigation [14, 18], malware and phishing detection [42, 2], authentication [23] and more.

1.1.2. Offensive AI

Artificial intelligence can be a double-edged sword, allowing attackers to leverage its capabilities for malicious purposes.

This topic is becoming increasingly important in the hacking and ethical hacking world and has been defined as *Offensive Artificial Intelligence*.

Offensive AI refers to the use of artificial intelligence technologies to conduct cyber-attacks, enhance traditional hacking techniques, or create new methods of breaching digital security systems. The same models that can predict and prevent attacks can be inverted to find and exploit vulnerabilities more efficiently than ever before.

The progress in AI not only presents opportunities for defensive strategies but also poses significant challenges as it becomes a potent weapon in cybercriminals' arsenal.

1.2. Motivation

The motivation behind this research relies on the possible usage of Offensive AI to enhance the penetration testing process, uncover more vulnerabilities, speed up the process, and test systems more thoroughly.

Additionally, showing the feasibility of these enhanced attacks can highlight how advanced malicious actors can exploit offensive AI to conduct cyberattacks and can help design new defence mechanisms for these new attack scenarios.

The scope of this research, based on web applications, does not focus on specific technologies or vulnerabilities but wants to build on a general context to demonstrate the feasibility of using offensive AI to improve the standard methodology of bruteforce directory attacks.

1.3. Contribution

This thesis project introduces two novel approaches to boost the performance of directory brute-forcing attacks. The first approach employs a probabilistic strategy, while the second approach uses a Language Model for web path generation.

Both techniques exploit prior knowledge from various web applications to predict possible directories and implement dynamic decision-making when choosing what HTTP requests to send.

The contributions of this thesis project can be summarised as follows:

- A novel dataset containing data from four distinct types of web applications

frequently targeted for cyber attacks: commercial, government, hospital, and universities. The dataset contains more than 1 million URLs.

- A novel directory brute-force attack methodology that employs a probabilistic strategy and dynamical decision-making.
- Another novel directory brute-force attack methodology that leverages a Language model to generate possible directories to discover.

The two proposed novel approaches are systematically evaluated on 8 different baselines based on the standard directory brute-force approach. The results highlight the superiority of the proposed approaches, with an average performance increase of 969% for the Language model-based approach and an optimal use of the probabilistic strategy in stealthier situations.

Ethical Disclaimer: The techniques and methods discussed in this research are intended for educational purposes and ethical security testing only. The authors do not condone the use of these methods for malicious purposes and strongly advocate for responsible disclosure and remediation of identified vulnerabilities. We hope that this research will contribute to the development of more secure web environments and the advancement of cybersecurity practices. For this reason, we do not share publicly the collected dataset and code. Researchers willing to reproduce our experiment are invited to contact the authors.

1.4. Report Structure

The report is structured in the following manner to describe the concepts, the analysis, and the results of this thesis project. In Chapter 2, background theory and related works are presented. In Chapter 3 the threat model is discussed. In Chapter 4, the intuition, methodology and implementation of the standard directory brute-force approach and the two novel approaches are presented. In Chapter 5, the data collection and the dataset analysis are reported. In Chapter 6 there is an overview and discussion of the results obtained. Finally, in Chapter 7, the conclusions, limitations and future works are presented.

2

Background

This chapter delves into Language models, providing concepts and theory behind their functioning and research works on the topic.

In addition, this chapter introduces the theme of Offensive AI, presenting state-of-the-art research that investigates its application in cybersecurity and related works to this project.

2.1. NLP

Artificial Intelligence (AI) is a branch of computer science that aims to create systems capable of performing tasks that typically require human intelligence. These tasks are almost unlimited and can range from learning from experience to understanding natural language, recognizing patterns, and making decisions. A field that has gained increasing attention in recent years alongside generative AI is Natural Language Processing (NLP).

NLP is a branch of artificial intelligence that deals with the interaction between computers and human language. NLP techniques combine computational linguistics with statistical and machine learning models to comprehend, recognize, and produce text and spoken language. Some of the main applications of NLP are machine translation [43, 21], speech recognition [28], text summarization and generation [1, 24], and information retrieval [25].

Additionally, a foundational concept of NLP that we use in this research and that we will explore in more detail is Language modelling and language models.

2.2. Language Models

2.2.1. Overview

Language Models (LMs) are statistical models used to predict the next word in a sentence given the previous words, essentially learning the probability distribution of a sequence of words.

The theory behind these models is based on probability theory and word embeddings [32, 36, 34]. These models are often employed in understanding and generating sequences of words. Their goal is to accurately predict the likelihood of words following the context of a preceding sequence of words. Mathematically, considering a sequence of words as:

$$x = (x^{(1)}, \ldots, x^{(t)}). \tag{2.1}$$

The probability of the sequence is obtained by applying the chain rule of proba-

bility:

$$\begin{aligned} P((x^{(1)}, \ldots, x^{(T)})) &= P(x^{(1)}) \cdot p(x^{(2)}|x^{(1)}) \cdot \ldots \cdot \\ & p(x^{(T)}|x^{(T-1)}, \ldots, x^{(1)}) \\ &= \prod_{t=1}^{T} (x^{(t)}|x^{(t-1)}, \ldots, x^{(1)}). \end{aligned} \tag{2.2}$$

Then, the probability distribution of the next word $x^{(t+1)}$ is computed as follow:

$$P(x^{(t+1)}|x^{(t)}, \ldots x^{(1)}), \tag{2.3}$$

where $x^{(t+1)} \in V = w_1, \ldots, w_{|V|}$, and V is a fixed vocabulary.

To give a practical example, let us consider a language model that is trained with the following sentences:

1. “The cat ate a mouse.”
2. “The dog chases the cat.”

The model will learn the probabilities of word sequences like “the cat”, “cat ate”, “chase the cat,” and so on. Now, suppose we want to predict the next word after “the”. In this case, the model will assign some probabilities to the vocabulary words (composed of all the unique words in the two training sentences). Among the words, the model will likely predict “cat” as the most likely word since it is the only word the appears after “the” more than once.

Traditional Language Models were based on N-grams [31]. N-grams are sequences of *n* words: for example, “I love” is a bigram (*n*=2), and “I love football” is a trigram (*n*=3). N-gram language models relied on counting the frequency of sequences in the training data to estimate the likelihood of a word based on the preceding N-1 words. However, these models struggled with data sparsity and could not capture long-range dependencies and context beyond the previous N-1 words.

Recent progress in AI, especially in Deep Learning, has led to more advanced LMs that utilize neural networks to learn complex relationships between words and context, overcoming previous limitations. Specifically, Recurrent Neural Networks (RNNs) [35], Long Short-Term Memory (LSTM) networks [17], and Gated Recurrent Units [8] (GRUs) are designed to handle sequential data processing that we can make use of for our purposes.

- RNNs are neural networks that maintain a *hidden state* that acts as a form of memory, using this state to retain information from previous steps. At each time step, a RNN calculates the output y^t and the new hidden state h^t using the input x^t and the hidden state h^{t-1} from the previous step. However, standard RNNs suffer from the vanishing gradient during training [16], resulting in difficulties with learning and remembering long-term dependencies.
- LSTMs are a particular type of RNN designed to overcome the vanishing gradient problem, employing a specific cell state and gating mechanisms.
- GRUs are another type of RNN designed to solve the vanishing gradient problem with a more straightforward structure than LSTM networks. This structure makes them computationally more efficient but with the cost of reduced capacity for complex patterns.

LMs are trained on extensive text corpora. The parameters of these models are learned to optimize the likelihood of the observed sequences, a process known as maximum likelihood estimation.

Considering a sequence, at each step t of the training, a loss function is defined as the cross-entropy between the predicted probability distribution $\hat{y}_t$ and the actual next word y_t. Mathematically, this can be expressed as:

$$J^{(t)}(\theta) = CE(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = -\sum_{w \in V} \mathbf{y}_w^{(t)} \log \hat{\mathbf{y}}_w^{(t)} = -\log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}. \tag{2.4}$$

By averaging the loss function over the entire training set, we obtain the overall loss as follows:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^t(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}. \tag{2.5}$$

2.2.2. Embedding

Embeddings play a vital role in the performance of LMs. To understand why, suppose we are building a language model to predict the next word given a sentence. Our training sentences are:

- "I love to play football."
- "He loves to play basketball."
- "She likes to play tennis."

Now, using the model, let us predict the next word in the sentence: "They love to play …". Without word embeddings, our model would treat each word as a separate entity. It would not understand that "football", "basketball", and "tennis" are all related because they are sports or "I", "He", and "She" are related because they are pronouns. With word embeddings, each word is represented by a vector in a high-dimensional space. Words with similar meanings are close together in this space. So "football", "basketball", and "tennis" would all be close together because they are all sports. So if we have to predict the following word in the sentence "They love to play …", the model can use these embeddings to understand that the next word is likely to be a sport, even if it has not seen the exact sentence in the training data.

This example shows how word embeddings allow language models to extrapolate the context from sequences and understand the semantic relationships between words, leading to more accurate predictions.

Theoretically, an embedding is a numerical vector that provides a representation for elements of language - words, phrases, sentences, or even entire documents. These vectors, often high-dimensional, encapsulate the semantic meaning of the text. Since the simple text is unsuitable for directly feeding to machine learning models, embeddings can convert text into a format that can be comprehended and manipulated. Additionally, this transformation process allows the models to capture the nuanced meanings of words based on their context, an indispensable feature for various tasks such as translation and sentiment analysis. Traditional word representations as discrete symbols or one-hot vectors were sparse and did not capture the semantic relationships between words. Embeddings address this issue by representing words as dense vectors in a lower-dimensional space. These dense vectors are learned from the data and can capture different semantic properties. There are several methods for embeddings, such as Word2Vec [26], GloVe [32] and FastText [19], that, with different approaches, aim to learn rich semantic representations of words.

2.3. Related Works

The latest progress in artificial intelligence and its widespread diffusion marks a new chapter where Offensive AI is utilized to conduct advanced and automated attacks [20, 27]. These attacks constitute a novel landscape that brings forth substantial challenges and opportunities in cybersecurity, particularly with the rise of LLMs and generative AI.

The exploration of generative AI to bolster directory brute-forcing attacks remains uncharted. However, some works have similarly explored enhancing directory brute-forcing attacks employing AI.

The nearest attempt is presented by He et al. [15], where the authors designed an approach utilizing semantic clustering of sentences to attack medical systems. Despite that, limited information was presented regarding the attack's data, methodology, experiments and outcomes. Adopting similar reasoning, Antonelli et al. [3] introduced a novel method to optimize directory brute-forcing using the Universal Sentence Encoder (USE) for semantic analysis and the K-means algorithm and the elbow method for clustering. The approach improved performance by up to 50% for each of the experiments conducted on eight different web applications tested.

Numerous other studies have analyzed the threat that offensive AI poses to organizations in different types of attacks and contexts. Bontrager et al. [6] exhibited the potential of AI-generated fingerprint deepfakes to undermine biometric systems through dictionary attacks. Al-Hababi et al. [13] probed man-in-the-middle attacks leveraging machine learning to identify services in encrypted network flows. Li et al. [22] demonstrated a generative adversarial network designed to evade PDF malware classifiers, highlighting how easily AI can circumvent traditional cybersecurity defences when maliciously used. Nam et al. [30] developed a recurrent GANs-based password cracker without user intervention. In the same context, Trieu et al [38] designed sophisticated AI-based password brute-force attacks by cleverly constructing the attack dictionary. Finally, Petro et al [33] implemented an AI-based hacking tool that learns to exploit databases in web applications using reinforcement learning.

Although these work can be employed for beneficial purposes by testing and increasing the security of systems and organizations, it also underscores how offensive AI is a rapidly growing field that exponentially increases the attack surface and poses new cyber challenges and threats.

3

Threat model

This chapter presents the threat model, from how the attack works to the targets and possible ways of conducting it.

3.1. Attack Description

A directory enumeration brute-force attack is a technique that probes for and tries to access hidden directories and files on a web server that are not referenced but still accessible.

This attack is executed by generating many requests, each associated with a different URL, and sending them to the server. Once the server responds by sending back HTTP responses, the attacker can examine the response content and the status code associated with the responses: if the response has a status code different than 404 (which indicates “Page not found”) but with other status codes such as 200 (request succeeded) or others, then the URL specified in the request and therefore the directory or file is likely to be available in the web application.

The attack often relies on a wordlist for the brute-forcing component, a compilation of words used to construct the new URLs that will be used to send the requests.

Due to its brute-force nature, directory brute-force attacks could be both time and resource-consuming, involving an enormous amount of requests to be sent.

The potential consequences of a directory enumeration attack are not to be underestimated. They can lead to the discovery of hidden files, directories, backup files, or administrative interfaces that may contain sensitive data or configuration details. If these resources are not adequately secured, they can be exploited to gain unauthorized access, escalate privileges, or initiate additional attacks.

To illustrate with a practical example, let us consider this scenario:

- a web server with a directory named *“/admin”*, which is not linked anywhere within the application but is still accessible if the exact URL is known.
- the web server home URL *http://example.com*
- a wordlist [*home, about, admin*].

An attacker could launch a directory enumeration attack using the mentioned wordlist in the following way:

1. Construct the new URLs: [*“http://example.com/home”*, *“http://example.com/about”*, *“http://example.com/admin”*].

2. Send sequentially HTTP requests associated with the newly created URLs.
3. Analyze the HTTP responses. For example, suppose the server responds with a response with a status code 200 for the URL *http://example.com /admin*. In that case, the attacker can infer the presence of the directory *admin* and explore the directory content, potentially launching further attacks.

Therefore, it's essential to ensure all directories and files are adequately secured, even if they are not directly linked within the application. Enhanced directory brute-force attacks that are able to detect vulnerabilities could be launched from penetration testers looking for vulnerabilities to fix but also from malicious actors.

3.2. Automated Tools

Various commercial and open-source tools are frequently employed to execute directory brute-force attacks. These tools can either be specialized for this specific attack type or more comprehensive, offering a range of functionalities, vulnerability scanners, proxies, and much more that come handy in security assessments.

These tools can ease the penetration test process and the launch of directory brute-forcing attacks, providing graphical user interfaces (GUI) to visualize attack progress in real-time, advanced attack implementations with multithreading available and custom options such as redirection following and response filtering. Additionally, these tools often come equipped with default wordlists.

Some of the most commonly used tools include:

- *Dirbuster*: This Java-based, multi-threaded tool was developed to brute force directories and file names on web or application servers. It was implemented and released open-source by the Open Web Application Security Project (OWASP). [1] It comes with nine different default wordlists, giving users a wide range of wordlist options. The tool is freely available and can be downloaded from the Kali Linux tools repository. [2]
- *Wfuzz*: This open-source security tool is designed to launch different types of brute-force attacks against web applications by fuzzing input parameters. Wfuzz is versatile and designed to perform various attacks, including brute-forcing, fuzzing, and injection attacks. Wfuzz also provides several wordlists employable for different types of brute-force attacks, among which directory brute-forcing attack is one possible option. The tool is freely available and can be accessed at its official documentation site. [3]
- *Burpsuite*: This commercial platform provides a user-friendly graphical tool for conducting security testing on online applications. It supports the entire security testing process, from initial automated mapping and analysis of an application's attack surface to discovering and exploiting security flaws, manual analysis and more. Among its many capabilities, Burpsuite can perform brute-force attacks to enumerate directories, given a target and a wordlist. The tool is available under different licenses and can be obtained from the official PortSwigger website. [4]

While these tools incorporate different features that may lead a user to pick one over another, the methodology by which they conduct a directory brute-force

[1] `https://owasp.org/`
[2] `https://www.kali.org/tools/dirbuster/`
[3] `https://wfuzz.readthedocs.io/en/latest/`
[4] `https://portswigger.net/burp`

attack is standardized and follows the same process.

On the other hand, choosing a proper wordlist can influence the obtained results much more, since it is the core component from which new URLs are generated. Therefore, some tools may provide a better variety of default wordlists than others.

It is again worth mentioning that although their use purpose is focused on legitimate security testing, they can also be misused by malicious actors.

3.3. Wordlists

In the context of directory brute-force attacks, wordlists are lists of words that may correspond to directory or file names.

As mentioned before, selecting an appropriate wordlist when conducting a directory brute-force attack is crucial, as it can significantly influence the outcomes and the vulnerabilities detected.

For this reason, a wide array of wordlist categories are available for various needs. Some of these categories are:

- general purpose wordlists: These wordlists contain general words that are likely to correspond to directories or files without any linking to a specific technology or web application context.
- backup-file wordlists: Wordlists in this category usually contain common names used for backup files.
- CMS-specific wordlists: These wordlists are designed to be used in web applications that utilize specific Content Management Systems (CMS) since they contain standard files and directory names associated with the specific technology.

Even if several automated tools come pre-packaged with various wordlists, the internet is also a rich source of user-created wordlists. There are numerous examples of user-created repositories where people created wordlists based on different criteria and made them available online. In addition, attackers can create custom wordlists that meet their specific requirements.

Four general-purpose wordlists, gathered from different sources and of different sizes, were selected for the purpose and scope of this project.

- `big_wfuzz` [5][**BW**]: This is a default general-purpose wordlist provided by Wfuzz. This wordlist is the smallest among the four, containing 3024 words.
- `top_10k_github` [6][**GH**]: This is a user-created wordlist available online. It contains 10,000 words. As criteria, the author selected 10,000 of the most common directory names found in more than 10 million URLs.
- `megabeast_wfuzz` [7][**MW**]: This is another default general-purpose wordlist provided by Wfuzz. Differently from the other wordlist from Wfuzz, this wordlist contains 45459 words.
- `directory-list_dirbuster` [8][**DB**]: This is a default wordlist provided by Dirbuster. This wordlist represents the biggest one among the selected wordlists, containing 141835 words in total.

[5] `https://github.com/xmendez/wfuzz/blob/master/wordlist/general/big.txt`

[6] `https://github.com/xajkep/wordlists/blob/master/discovery/top-10k-web-directories_from_10M_urlteam_links.txt`

[7] `https://github.com/xmendez/wfuzz/blob/master/wordlist/general/megabeast.txt`

[8] `https://github.com/3ndG4me/KaliLists/blob/master/dirbuster/directory-list-1.0.txt`

There are two main reasons for selecting these four wordlists. The first is the general-purpose nature of this project, which does not target specific vulnerabilities or technologies. The second is the possibility of evaluating the performances of wordlists of different sizes from different sources.

3.3.1. Selected Wordlists Similarity

To give an initial idea and better analyze the results obtained in this project and reported in Chapter 6, it may be useful to analyze the similarity among the four wordlists.

To do so, we will use two metrics: the number of words in common between each pair of wordlists and the Jaccard similarity between the two lists.

Specifically, the Jaccard similarity metric, defined in equation 3.1, can take a value between 0 and 1, where 0 indicates that the two sets are disjoint and 1 that they are identical.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{3.1}$$

In Figure 3.1, we can see the number of words each couple of wordlists has in common, while Figure 3.2 shows the Jaccard similarity for every couple of wordlists.

Both the figures, considering the size differences of the wordlists, highlight how the wordlists mainly differ in their content, even though they are supposed to be used in the same attack context.

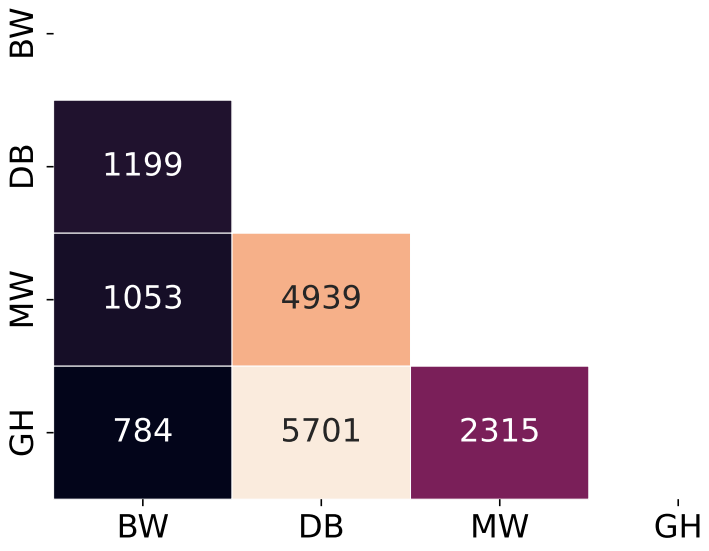


Figure 3.1: Number of words in common for each couple of wordlists

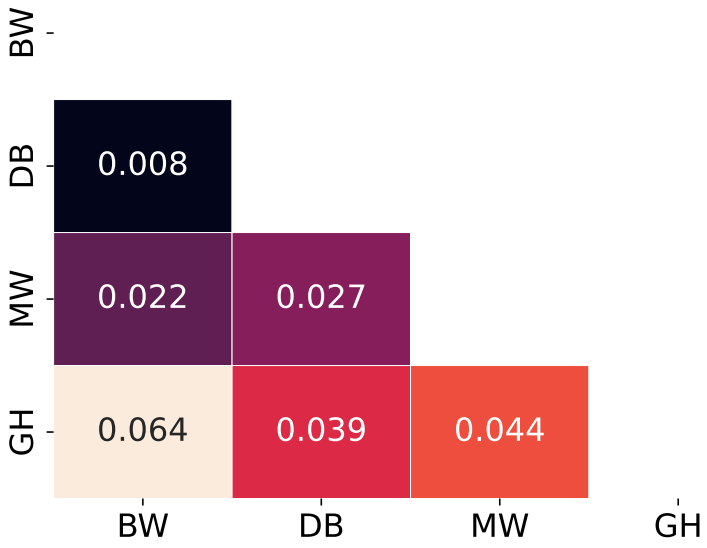


Figure 3.2: Jaccard similarity for each couple of wordlists

4

Attack Methodology

This chapter presents an overview of the standard approach to perform a directory brute-force attack.

Additionally, we introduce the two enhanced approaches that are the subject of this project, alongside the intuition behind them, how they work, and how they can be implemented.

4.1. Overview

Traditional brute-force attacks are inefficient due to their nature. In the context of directory enumeration, sequentially sending numerous requests can be both time-consuming and resource-consuming since the number of possible URLs to send requests is limitless.

To conduct directory brute-force attacks, we have to generate numerous URLs and send many HTTP requests associated with those URLs to the web application. Then, we must wait for the web server to answer and analyse the HTTP responses received.

This strategy also raises an additional problem: sending an enormous number of requests in a limited amount of time may trigger cyber defensive systems such as Intrusion Detection Systems (IDS), Firewalls and Black-holing. Furthermore, if the web server cannot handle all the incoming requests, this attack can lead to a Denial of Service or a service disruption of the web application.

In this study, we want to explore the feasibility of enhancing this simple strategy by implementing attacks that exploit information and smart strategies that the standard directory brute-force approach does not consider. This can potentially boost the performance of this type of attack. In particular, we aim to leverage the two features: **Prior Knowledge** and **Adaptive decision-making**.

The intuition behind prior knowledge is simple: when considering web applications, several can be implemented with the same technologies, such as web servers, databases, and CMS. In addition, web applications to the same context and category, such as university or hospital websites, may use a similar structure and convention in indexing folders and files (considering web applications that use the same language). This work investigates whether leveraging this information when conducting a directory brute-force attack can yield better results.

On the other hand, the reasoning behind adaptive decision-making lies in choosing the next request to send. Instead of trying sequential words to construct a

new URL, the following HTTP request to be sent should be decided based on what directories and files we have discovered so far in the web application and dynamically adjusted at run-time. If we can use some criteria to select the next request to be sent.

4.2. Tree Reconstruction

Before explaining the standard and enhanced approaches, it is helpful to understand how we manage to perform directory brute-force attacks while maintaining an ethical posture that does not harm any web application. We decided to run offline simulations of the attacks, reconstructing the filesystem of every web application target in our tests. To do so, we used the data described in Chapter 5 , which consists of HTTP responses crawled from various web applications. For each URL in a HTTP response, we can consider the domain as the root of the filesystem, which is the same for every HTTP response received from the same web application. Then, since the filesystem of a web application has a hierarchical tree structure, the paths can be used to reconstruct the directory-subdirectory relations. In this project, the filesystem reconstruction is performed using the `AnyTree` [1]. For example, let us consider the following URLs that are part of the crawl of a web application:

- "`http://www.example.com/`"
- "`http://www.example.com/news`"
- "`http://www.example.com/home`"
- "`http://www.example.com/register`"
- "`http://www.example.com/news/2024`"
- "`http://www.example.com/news/today`"
- `vhttp://www.example.com/news/weather`"

From these URLs, we can extract "`www.example.com`" as the domain, and the paths "`/news`", "`/home`", "`/register`", "`/news/2024`", "`/news/today`" and "`/news/weather`".

Considering the root node as "/", a visualisation of the reconstructed tree corresponding to the web application filesystem can be seen in Figure 4.1.

[1] `https://anytree.readthedocs.io/en/latest/`

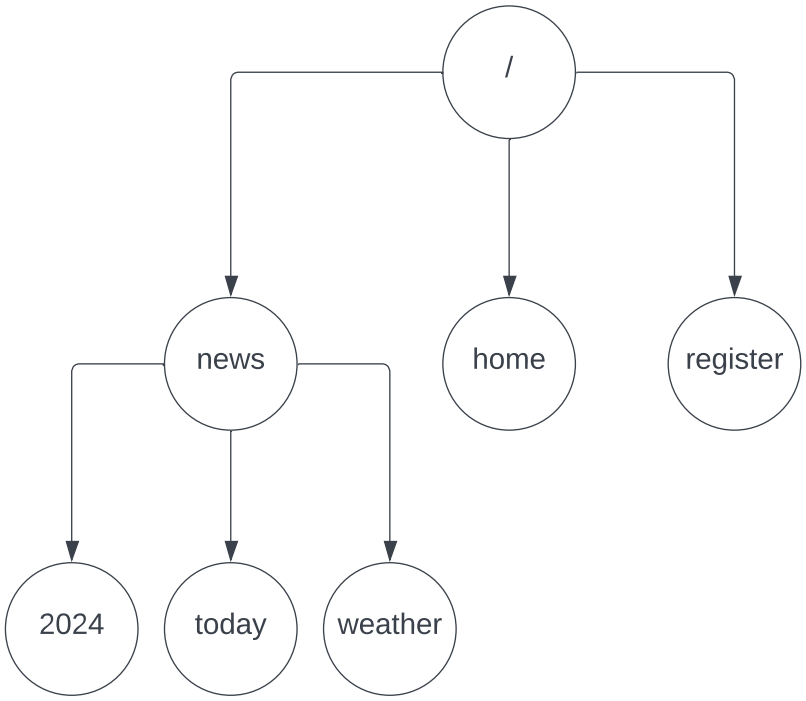


Figure 4.1: Visualization of a reconstructed tree.

When conducting an offline attack simulation, the attacks will generate new URLs. From these URLs, we will need to extract the path. Then, we will try to traverse the tree starting from the root: for every node of the tree, which represents the current directory, we check if in its children, which represents the subdirectories, there is a word corresponding to the next one in the path. If that is not the case for every word in the path, the generated URL is not valid; elsewhere, the URL is valid, and the last word in the path is the newly discovered directory.

4.3. Standard Approach

The standard directory brute-force attack based on a wordlist can follow two strategies: *Depth-First* and *Breadth-First*.

In the results comparison of the enhanced approaches results, we will run offline attack simulations employing both these strategies and use the results obtained as comparison baseline.

4.3.1. Depth-First

A directory brute-force attack that follows the depth-first approach prioritizes exploring subdirectories when a new directory is discovered before proceeding to explore other directories at the same hierarchical level.

An attack that employs this approach begins by sequentially dispatching HTTP requests and generating new URLs using the entries from a wordlist. When a positive response is received (denoting the discovery of a valid URL and a new directory), the attack redirects its attention to brute-forcing the subdirectories of this newly identified directory. It conducts an exhaustive search within these

subdirectories (until there are no more words in the wordlist to use to generate new URLs) before it resumes the task of brute-forcing other directories at the same hierarchical level as the previously validated one.

This method ensures an exhaustive search within each directory before advancing to the next, optimizing the probability of revealing hidden directories or files at deeper levels.

For example, let us consider a web application target "*http://www.example.com*" with the corresponding filesystem identical to Figure 4.1 and the wordlist ["*news*", "home"].

If we had to conduct a directory brute-force attack using the depth-first strategy, the following steps show how the attack would work:

1. Send a HTTP request with URL. "*http://www.example.com/news*".
2. Receive a positive HTTP response with status code 200. "*news*" is a newly discovered directory (subdirectory of the root).
3. Explore the subdirectories of "*news*", starting from the beginning of the wordlist.
4. Send a HTTP request with URL "*http://www.example.com/news/news*".
5. Receive a negative HTTP response with status code 404. The URL is not valid, and "*news*" is not a subdirectory of "*/news*".
6. The previous two steps apply for the next URL "*http://www.example.com/news/home*".
7. The subdirectories research is exhausted since there are no more words to generate a new URL. Now, the attack resumes the brute-forcing process at the previous level.
8. Send a HTTP request with URL. "*http://www.example.com/home*".
9. Receive a positive HTTP response with status code 200. "*home*" is a newly discovered directory.
10. Explore the subdirectories of "*home*", starting from the beginning of the wordlist.
11. Send two HTTP requests with URLs "*http://www.example.com/home/news*" and "*http://www.example.com/home/home*".
12. Receive two negative responses with status code 404. Both the URLs are not valid.
13. The directory research at this level is terminated. There are no more requests to generate, so the attack terminates.

A possible implementation of an algorithm that employs this approach is exemplified in Algorithm 1.

In this algorithm, `constructURL(URL, word)` is a function that generates a valid URL given a word from a wordlist, appending it to the URL's current path. Additionally, `isValid(response)` is a function that, given a HTTP response, checks if the response is valid.

Algorithm 1 Depth-First brute-force attack Pseudocode

```
1: procedure DepthFirst(rootURL, wordlist)
2:     for each word in wordlist do
3:         url ← constructURL(rootURL, word)
4:         response ← sendHTTPrequest(url)
5:         if isValid(response) then
6:             DepthFirst(url, wordlist)
7:         end if
8:     end for
9: end procedure
```

4.3.2. Breadth-First

On the other hand, a directory brute-force attack that follows the breadth-first approach prioritizes exhausting the research of directories at a given level before exploring their subdirectories.

An attack that employs this approach initiates similarly as the depth-first strategy, sequentially dispatching HTTP requests and generating new URLs using the entries from a wordlist. Unlike the previous strategy, when a positive response is received (denoting the discovery of a valid URL and a new directory), the attack continues to brute-force other directories at the same hierarchical level, exhausting all the possible URLs that it can generate given the wordlist. Once this research is over, the attack will start to brute-force all the subdirectories of the newly discovered directories, from the first to the last discovered.

This method ensures an exhaustive search for each hierarchical level before moving to deeper levels.

This strategy promises better results than the previous approach since directories at deeper hierarchical levels are likely to be rare and, therefore, not in the wordlist.

Indeed, most commercial tools adopt this strategy, even those that implement additional features such as multi-threading and response filtering.

To visualize how these two approaches work differently, let us consider the previous example where we had the web application target "*http://www.example.com*" with the corresponding filesystem in Figure 4.1 and the wordlist ["*news*", "*home*"].

Conducting a directory brute-force attack using the breadth-first strategy would follow these steps:

1. Send a HTTP request with URL. "*http://www.example.com/news*".
2. Receive a positive HTTP response with status code 200. "*news*" is a newly discovered directory (subdirectory of the root).
3. Continue to brute-force directories at this depth. Send a HTTP request with URL "*http://www.example.com/home*".
4. Receive another positive HTTP response with status code 200. "*home*" is another discovered directory.
5. The research at this level is exhausted. Now start exploring the subdirectories of the first discovered directory "*news*".
6. Send a HTTP request with URL "*http://www.example.com/news/news*".
7. Receive a negative HTTP response with status code 404. The URL is not valid, and "*news*" is not a subdirectory of "*/news*".
8. The previous two steps apply for the next URL "*http://www.example.com /news/home*".

9. The subdirectories research for "*news*" is exhausted since there are no more words to generate a new URL.
10. Start brute-forcing the subdirectory of the following discovered directory "*home*".
11. Send two HTTP requests with URLs "*http://www.example.com/home/news*" and "*http://www.example.com/home/home*".
12. Receive two negative responses with status code 404. Both URLs are not valid.
13. The subdirectory research for "*home*" at this level is terminated. There are no more discovered directories to explore their subdirectories. The attack is terminated.

We also present Algorithm 2 that implements the breadth-first strategy.

In this algorithm, `constructURL(URL, word)` `isValid(response)` are the same functions presented in Algorithm 1. Additionally, the algorithm uses a `queue` to store the discovered directories and to maintain the order of which directory the attack should first brute-force the subdirectories.

Algorithm 2 Breadth-First brute-force attack Pseudocode

```
1:  procedure BreadthFirst(rootURL, wordlist)
2:      queue ← new Queue()
3:      queue.enqueue(rootURL)
4:      while queue is not empty do
5:          currentURL ← queue.dequeue()
6:          for each word in wordlist do
7:              url ← constructURL(currentURL, word)
8:              response ← sendHTTPrequest(url)
9:              if isValid(response) then
10:                 queue.enqueue(url)
11:             end if
12:         end for
13:     end while
14: end procedure
```

4.4. Probability Based Approach

The first enhanced approach that we have designed is a **probabilistic approach**.

This section presents how the approach works, its building blocks, a practical example, and a possible algorithmic implementation.

4.4.1. Approach

Leveraging prior knowledge can significantly enhance the effectiveness of a directory brute-force attack. The underlying concept is quite simple: if many web applications contain similar paths such as `/login` and `/register`, the web application target of an attack will probably contain these paths and, hence, the directories that form the paths. Consequently, an algorithm that prioritizes directories based on prior knowledge can be highly effective.

The first novel approach we introduce refines the standard approach discussed in Section 4.3. This approach introduces the usage of prior knowledge and exploits it to implement dynamic decision-making regarding URL generation for the subsequent HTTP request.

The goal is to maximize the number of successful requests, selecting the most probable URL to send a HTTP request to each time, while minimizing the number of unlikely and incorrect ones.

The prior knowledge, or training dataset, comprises crawled paths from various web applications. Ideally, these applications should belong to the same category as the target of the attack that will be conducted. This ensures that the strategy is tailored to the specific characteristics of the target application, thereby increasing the likelihood of a successful attack.

The integration of prior knowledge into the attack methodology can be achieved in two distinct ways:

1. Building a **Weighted Training Tree**. Similarly to how we have described how to reconstruct a filesystem from the paths of a web application, a possible way to integrate prior knowledge in the attack methodology is by building a unique filesystem tree that will unify all the paths from our training dataset, even from different web applications. This tree has to be weighted, meaning that for each new node in the tree (which corresponds to a directory), we maintain a counter that indicates the frequency of that particular node. It is essential to highlight that for each node its weight represents how many times that directory is at that depth level in a path: if we consider the two paths "*/news*" and "*/home/news*", we will have one node at depth 1 for "*news*" with weight=1, and another node "*news*" as well at depth 2 with weight=1.

   A visualization of how the filesystem tree reported in Figure 4.1 would be transformed into a weighted training tree is shown in Figure 4.2.

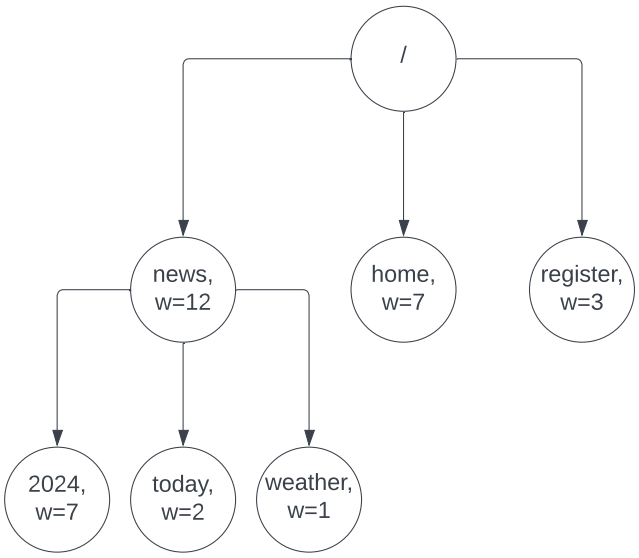


Figure 4.2: Visualization of a Weighted Training Tree, obtained merging paths from a Training Dataset.

2. **Building a Weighted Wordlist Tree**. This strategy involves constructing a weighted training tree as described in the previous point, using only the words from a given wordlist. Starting from a general wordlist and a training dataset, we construct a weighted tree similar to the Weighted Training

Tree. However, this tree only includes words from a wordlist and every node that includes a directory, not a word in the wordlist, is pruned. The weight of each node (directory) in this tree is determined based on the training set. For instance, if we consider the wordlist [“*news*”, “*home*”, “*2024*”, “*today*”, “*about*”] and the Weighted Training Tree shown in Figure 4.2, the corresponding Wordlist Weighted Tree can be visualized in Figure 4.3. In this case, directories such as `register` and `weather` are not included in the tree, as they are not part of the original wordlist.

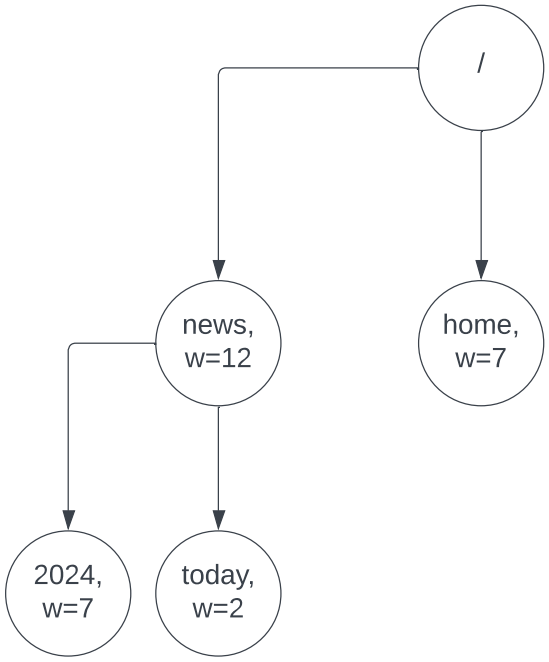


Figure 4.3: Visualization of a Wordlist Weighted Tree, based on a wordlist and a Weighted training tree.

By adopting these strategies, we can effectively utilize the parent-child relational information between directories and subdirectories and assign weights to words in the wordlist, which we can use to decide the next HTTP request.

Additionally, in the case of the weighted wordlist, we apply a process of pruning all the wordlist’s words that do not appear as directories in the training dataset and vice versa. Pruning a node means that that node has weight 0.

This pruning of unlikely words aims to minimize the number of less probable requests that we would otherwise consider.

For a clearer understanding of how pruning operates, let us consider Figure 4.2 and Figure 4.3. In this example, we can observe that the word “`about`”, initially part of the wordlist used to create the tree, is not included in the Wordlist Weighted Tree (thus, it has a weight of 0 as a possible subdirectory). Similarly, “`news`” has a weight of 12 as a subdirectory of the base root, but it is pruned from being a child node of “`home`”.

Finally, it is worthwhile mentioning the correlation between a weighted training tree and a weighted wordlist tree: a weighted training tree is simply a weighted

wordlist tree where the corresponding wordlist is the set of all directories that are in the paths in the training dataset. We will refer to this wordlist as **train-set**.

4.4.2. Algorithm

Our probabilistic attack algorithm utilizes a max heap to integrate the dynamical decision-making feature. This data structure maintains the element with the higher value of a given property at its top and allows efficient insertion and removal of elements. This heap contains tuples of base URLs, a word, and its assigned weight. The tuples in the max heap are arranged according to their probability, allowing us always to select and send the request with the highest probability.

In order to properly use the prior knowledge, the probability assigned to a word to be a valid subdirectory of a directory is dynamically calculated by dividing the weight of the word in the weighted tree by the total weight of all possible words that can be subdirectories of that word.

At the start, the algorithm initiates by pushing all possible subdirectories of the root directory "/" from the weighted tree into the max heap, each with its corresponding probability, given a target base URL. Then, it selects the tuple at the top of the heap, constructs a new URL using the base URL and the word from the tuple and sends a request. When a successful response is received, the algorithm pushes all potential subdirectories of the response URL and their probabilities into the heap.

The process of sending each time HTTP requests utilizing the tuples at the top of the heap (i.e., the one with the highest probability) enables the implementation of the adaptive decision-making feature. The described algorithm is presented in algorithm 3.

Here, `getWordswithWeights()` returns the pairs of words and weights from the specified URL in the weighted tree (i.e., our prior knowledge, while `getProbability()` computes the probability of a word as previously described.

Algorithm 3 Probabilistic brute-force attack Pseudocode

```
1:  procedure Probabilistic(rootURL, weightedTree)
2:      maxHeap ← new MaxHeap()
3:      rootTuples ← getWordswithWeights (rootURL, weightedTree)
4:      for each word, weight in rootTuples do
5:          prob ← getProbability(word, weight, rootTuples)
6:          maxHeap.push((rootURL, word, probability))
7:      end for
8:      while maxHeap is not empty do
9:          currentURL, word, probability ← maxHeap.pop()
10:         url ← constructURL(currentURL, word)
11:         response ← sendHTTPrequest(url)
12:         if isValidURL(response) then
13:             newTuples ← getWordswithWeights(url, weightedTree)
14:             for each word, weight in newTuples do
15:                 newProb ← getProbability(word, weight, newTuples)
16:                 maxHeap.push((url, word, newProb))
17:             end for
18:         end if
19:     end while
20: end procedure
```

The pruning process may prune the majority of possible HTTP requests that

the attack would send for different reasons, from incompatibility between the training dataset and the selected wordlist or due to a small dataset.

Considering a given budget of requests, the attack may terminate early before exhausting the number of possible requests to send.

To address this problem, we suggest integrating a standard breadth-first directory brute-force attack after the termination of the probability attack if the requests budget has not been fully consumed, with a slight difference: it is important to save the URLS that have been used in the probability attack to send HTTP requests, so during the standard attack, we avoid sending requests that have already been sent.

In our offline attack simulations, we chose to use two strategies.

To give an example of how this attack methodology works, let us again consider the web application target "*http://www.example.com*" with the corresponding filesystem shown Figure 4.1 and our prior knowledge corresponding to the weighted wordlist tree in Figure 4.3. A directory brute-force attack employing the probability-based approach would follow the following steps:

1. The attack starts pushing the possible subdirectories of the root directory "/" in the max heap. In this case, "*news*" will have a probability of 63%, while "*home*" 27%. After the operation, considering the left-most element as the maximum element, the heap will be [("/", "*news*", 63%), ("/", "*home*", 27%)].
2. Pop the top element of the heap. Construct the URL "http://www.example.com/news" and send a request. The heap becomes [("/", "*home*", 27%)].
3. Receive a positive HTTP response with status code 200. A new directory has been discovered. Now, for the discovered directory, push all the possible subdirectories retrieved from prior knowledge in the heap. After this operation, the heap will be [("*/news*", "*2024*", 78%), ("/", "*home*", 27%), ("*/news*", "*today*", 22%)].
4. Pop the top element of the heap. Construct the URL "http://www.example.com/news/2024" and send a request. The heap becomes [("/", "*home*", 27%), ("*/news*", "*today*", 22%)].
5. Receive a positive HTTP response with status code 200. A new directory has been discovered. Since in the prior knowledge we do not have any possible subdirectory for "*/news/2024*", no element is pushed in the heap.
6. Pop the top element of the heap. Construct the URL "http://www.example.com/home" and send a request. The heap becomes [("*/news*", "*today*", 22%)].
7. Receive a positive HTTP response with status code 200. A new directory has been discovered. Since in the prior knowledge we do not have any possible subdirectory for "*/home*", no element is pushed in the heap.
8. The previous two steps apply to the last element in the heap.
9. The attack terminates.

Additionally, we could launch the modified breadth-first directory brute-force attack as described before.

4.5. Language-Model Based Approach

The probability approach described in Section 4.4 presents how prior knowledge can be exploited to improve the standard methodology of directory brute-force attacks.

Upon closer examination of this approach, it becomes evident that the context plays a pivotal role in predicting new directories, underscoring the decision-making strategy. Being able to extrapolate the context from the information we have discovered so far during an attack can be the key to further boosting the attack's performance.

To achieve this, we have decided to design a new attack methodology that employs AI to refine the attack further and predict directories more efficiently and qualitatively.

4.5.1. Approach

The second enhanced approach proposed in this project employs a neural network mechanism that uses Language Models to predict likely subdirectories for a given path.

The reasoning behind this choice is simple: Given a URL, we can consider its path as a sequence of words divided by "/". This sequence can be input to the Language model, which predicts the words most likely to follow the input sequence. These predicted words can form new URLs and initiate new HTTP requests.

This method refines the probability-based approach by harnessing the strength of custom embeddings (i.e., embeddings trained on the corpus) and addresses the pruning implications. Specifically, the probabilistic method determines relationships among directories in similar URLs in the training dataset.

Additionally, embeddings help to extrapolate the context from paths and create connections between similar directories that will be used to generalize the predictions. For example, consider that in our prior knowledge, we have paths such as:

- `"/account/setting/info"`
- `"/account/setting/password"`
- `"/account/setting/logout"`
- `"/profile/setting/password"`
- `"/profile/setting/info"`

The directories `account` and `profile` are used in a similar context, and hence their embeddings will be similar.

During inference, a Language Model might predict the URL `"/profile/setting "/password"` even if this information was not in our prior knowledge. A probabilistic approach would have assigned 0% to this path, which would have led to avoiding this request.

4.5.2. Model Architecture

The neural network architecture that we use in our enhanced approach is based on several core components.

The first crucial component is the **vocabulary**. Since normal text cannot be fed directly to neural networks, the vocabulary is responsible for several operations:

- it collects all the different words (i.e., directories) in the dataset by filtering out the rarer ones that might create overfitting during model training. Given an input sequence, when the language model needs to predict the words that should follow the input sequence, it will take the vocabulary words and assign a probability to each.
- it maps word sequences into integer sequences that the Language model can process.

- it handles special cases, including words given unknown input and variable-length paths. The handling of particular cases and variable-length paths is performed using specific tokens: `UNK` (Unknown word), `PAD` (Padding token, used to pad sequences to a fixed size), `SOS` (Start of sentence token, used to highlight where a sequence start) and `EOS` (End of sentence token, used to highlight where a sequence end).

When it comes to the neural network architecture, we have designed several layers that compose our LM:

1. **Embedding Layer**: The initial layer of the model is an embedding layer. This layer converts input words into dense vectors of a predetermined size, known as the embedded size. The representations for these embeddings are learned during the training phase.
2. **LSTM Layer**: Following the embedding layer is an LSTM layer, essential for recognizing patterns in sequential data and exploiting the extrapolated context. This layer takes the word embeddings as input and produces its hidden and cell states.
3. **Dropout Layer**: To avoid overfitting our architecture (which could lead to poor results), dropout layers are incorporated after the Embedding and LSTM layers. Dropout is a regularization method that randomly sets a portion of input units to 0 at a certain probability rate during each training step.
4. **Fully Connected Layer**: The outputs (hidden states) from the LSTM are then fed into a fully connected layer. This layer transforms the LSTM outputs into the required output shape corresponding to the size of the vocabulary.
5. **Softmax Function**: Finally, a softmax function is used to convert the output from the fully connected layer into probabilities assigned to each word in the vocabulary.

Figure 4.4 shows an overview of the model architecture and exemplifies the entire process of prediction generation. In the figure, we can see how an input sequence is first split into tokens, encoded to a sequence of integers and special tokens, and fed to the language model that will then assign a probability to each word of the vocabulary. Finally, the word with the highest probability is chosen to generate a new path that can later be used to send a new HTTP request.

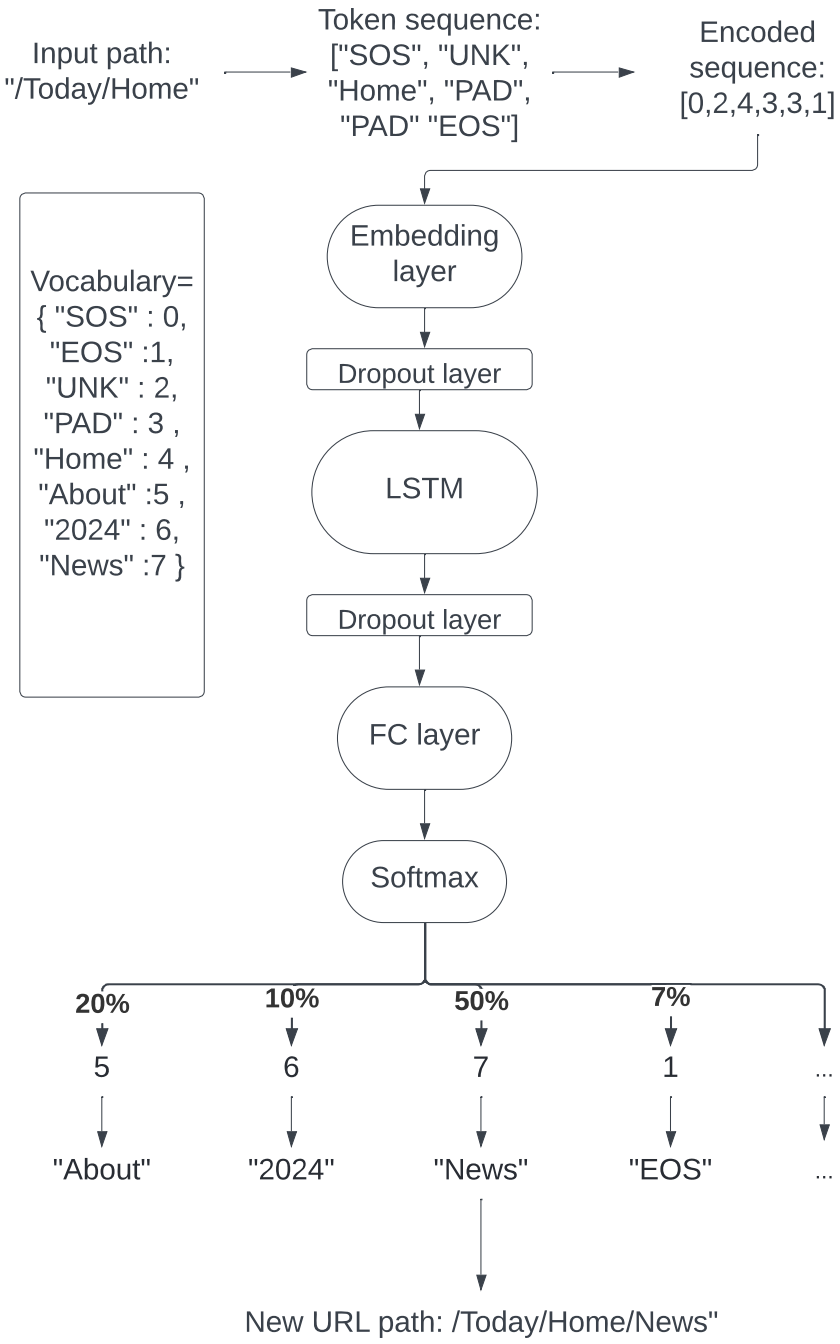


Figure 4.4: Prediction of the next directory from our LM-based architecture.

4.5.3. Training and Validation

The training process of our architecture that follow the standard training of a Language model can be described in more detail as follows: The model is trained by feeding it paths from the training dataset. Then, the model predicts the following directories to the given path.

Once the model generates the predictions, these predictions are compared with the subsequent directory in the path, and a loss function is employed to mea-

sure the discrepancy between the model's predictions and the actual values. This loss quantifies the error made by the model in its predictions, and an optimization algorithm is then used to minimize this loss correcting the model's parameters responsible for the predictions to make them more accurate.

The model's performance is also periodically evaluated on a validation set during training. This technique is typically adopted to evaluate how the model can generalize unseen data. Using the validation set, we also employ an early stopping mechanism to prevent the model from overfitting on the training data. Overfitting is a phenomenon where the model performs well on the training data but poorly on unseen data. The early stopping mechanism halts the training process when the model's performance on the validation set starts to overfit or does not improve for a certain number of epochs, denoted by p. The loss computed on the validation sample fed as input determines the model's performance on the validation set.

4.5.4. Algorithm

The algorithm that implements the language-based approach attack, shown in Algorithm 4, uses a logic similar to that of the probabilistic approach, using a max heap to choose the requests to be sent dynamically. In this case, though, a weighted tree is not used as prior knowledge since it is unnecessary since the Language model has already been previously trained on the training dataset.

In the algorithm, the function `predict()` returns the most likely K subdirectories given a URL, where K is a parameter defined as `topPredicts` chosen by the attacker.

For each new directory identified, the LM is given as input the path to the URL including the new directory. Then, the LM returns predictions of possible subdirectories with a certain probability in output. These tuples of base URL, possible subdirectory, and associated probability are fed into the heap, which will then sort them by probability from highest to lowest.

Algorithm 4 Language-model based brute-force attack Pseudocode

```
1:  procedure LMattack(rootURL, LM, topPredicts)
2:      maxHeap ← new MaxHeap()
3:      rootTuples ← predict(rootURL, LM,topPredicts)
4:      for each word, prob in rootTuples do
5:          maxHeap.push((rootURL, word, prob))
6:      end for
7:      while maxHeap is not empty do
8:          currentURL, word, prob ← maxHeap.pop()
9:          url ← constructURL(currentURL, word)
10:         response ← sendHTTPrequest(url)
11:         if isValidURL(response) then
12:             newTuples ← predict(url, LM,topPredicts)
13:             for each word, prob in newTuples do
14:                 maxHeap.push((url, word, newProb))
15:             end for
16:         end if
17:     end while
18: end procedure
```

To give an example of how this attack compares to the probabilistic approach, let us again consider the same scenario with the web application target "*http://www.example.com*" and the corresponding filesystem shown in Figure 4.1. We also set the prediction parameter `topPredicts` to 3.

A directory brute-force attack employing the Language model-based approach would employ the following steps:

1. The attack starts feeding to the LM the path of the root URL "/". In this case, the LM is fed with the token "`SOS`". For example, the LM may return [("*news*", 70%), ("*about*", 10%), ("*Agenda*", 4%)]. These 3 predictions are then pushed to the heap, which will then become = [("/", "*news*", 70%), ("/", "*about*", 10%), ("/", "*Agenda*", 4%)]
2. Pop the top element of the heap. Construct the URL "http://www.example.com/news" and send a request. The heap becomes [("/", "*about*", 10%), ("/", "*Agenda*", 4%)].
3. Receive a positive HTTP response with status code 200. A new directory has been discovered. Now, feed the path of the newly discovered URL to the LM, get 3 new predictions and push them into the heap. After this operation, our heap may become [("/news", "*2024*", 85%), ("/", "*about*", 10%), ("/news", "*press*", 8%), ("/", "*media*", 6%) ("/", "*Agenda*", 4%)].
4. the attack can continue sending requests and pushing new predictions in the heap every time a new directory is discovered.

This hypothetical attack scenario shows how this novel approach is highly adaptable to suit the attacker's needs and overcomes the pruning problem in the probability-based approach. Since the predictions made by the model can be unlimited, the attacker can set a proper value for `topPredicts` that better fits his requests budget.

5

Datasets

This chapter presents the datasets used in this project. We present how we have collected the data, how we have processed them and the in-depth analysis we have conducted that helped us to design the proposed novel approaches.

5.1. Data Description

5.1.1. Source Of Data

The data utilized for this research is sourced from CommonCrawl. CommonCrawl is a non-profit organization that periodically crawls the internet and freely offers its archives and datasets for public use. We chose CommonCrawl due to its extensive collection of web crawls that are regularly updated; specifically, we used the `CC-MAIN-2023-40` version of the crawl.

This choice simplifies the data-gathering process and also aligns with ethical guidelines. By doing so, we can avoid performing manual spidering and crawling of web applications, which could potentially be classified as brute-force attacks, and prevent overloading web servers with excessive requests.

Since this project aims to demonstrate the feasibility of enhancing the directory brute-force attack methodology, using historical data is perfect for us, considering we do not focus on specific vulnerabilities or technologies.

CommonCrawl provides crawls in different formats: WARC (Web ARChive Format), WAT (Web Archive Transformation), and WET (WARC Encapsulated Text). These contain raw response data, metadata, headers, and more. The data can be accessed by querying for specific data using AWS Cloud or directly downloading the compressed files containing the crawls.

The data we need to collect are the URL and the status code for each HTTP response we want to collect. We will explain how to process this information later in the preprocessing section.

In addition, it is important to consider that web applications can block CommonCrawl from crawling parts or even the entire website if specified in the `robots.txt` file.

5.1.2. Datasets Collected

Given the rising number of cyber attacks and the broad range of potential targets, we decided to select four distinct datasets representing some of the most common categories of organizations at risk of cyber attacks.

Additionally, due to the scope of the search, we restricted our focus to English-

based web applications since different languages in the directory naming convention can significantly increase the complexity of the directory enumeration process.

For each target category, we selected several web applications and specifically queried all HTTP responses in CommonCrawl associated with them.

The four datasets that we have collected from this process are:

- *Universities dataset* [**UNI**]: In this dataset, we choose the first 100 universities listed in QS 2023 World University Rankings. [1] We retrieved the corresponding web applications from this list and collected the associated HTTP responses.
- *Hospitals dataset* [**HOS**]: In this dataset, we choose the first 100 USA hospitals listed in "Ranking Web of World Hospitals". [2] As previously, we retrieved the corresponding web applications from this list and collected the associated HTTP responses.
- *Companies dataset* [**COM**] This dataset comprises the HTTP responses from 100 corporate web applications of companies listed in the S&P 500, selected in order of highest capitalization as of January 2024. Companies whose main web application was e-commerce were excluded since, in many cases, this would add pointless complexity.
- *Government dataset* [**GOV**]: In this dataset, we chose 336 different USA government web applications. [3] We retrieved the corresponding web applications from this list and collected the HTTP responses associated with the web applications.

Having four different datasets can help us evaluate the novel attack methodologies we have designed more thoroughly, comparing them to the standard methodology. In our experiments, we will evaluate the performance of the attacks considering each dataset alone and all the datasets together collectively. We will refer to this comprehensive dataset as the *general dataset* [**ALL**].

5.1.3. Preprocessing Of Data

After the collection of the HTTP responses, we proceeded to process the data.

From the URL of each HTTP response, we extracted the domain, which identifies the web application, and the path, which is the core component for the filesystem reconstruction, the weighted tree and the Language model training.

Our next steps involved applying additional preprocessing to eliminate unnecessary data, ensure accurate filesystem reconstruction of web applications, and facilitate further analysis.

Firstly, we filtered responses by keeping only the HTTP responses with a status code of 200 (representing the vast majority of the collected responses). We chose to do so mainly to make the filesystem reconstruction more efficient and accurate.

Secondly, we removed all the eventual parts from the URL paths that were queries or files. In this way we could reduce the already high degree of complexity of the possible directory words to predict. These specific files and queries are often associated with specific technologies that the web applications employ, and the focus on these technologies is beyond the scope of this project.

[1] `https://www.topuniversities.com/world-university-rankings/2023`
[2] `https://hospitals.webometrics.info/en/americas/usa`
[3] `https://www.usa.gov/agency-index`

Thirdly, we have the **depth** metric: the *depth* of a path is the number of directories that form that path. In other words, considering the path as a sequence of words, the depth is the number of words in the sequence. In this sequence, the order matters since words that come later are deeper into the hierarchical filesystem structure of web applications.

To give an example, "`/news/2023`" has depth = 2, where "`news`" is at depth 1, and "`2023`" is at depth 2).

5.2. Datasets Analysis

After preprocessing, we conducted several analyses on the properties, similarities, and disparities for each dataset and between datasets.

Namely, we present the following analyses:

- *Properties*: Here, we present the datasets' properties, such as the number of domains, paths, directories and more.
- *Wordlist coverage*: Here, we analyze how the four wordlists we have selected cover the directories in the URL paths in the datasets.
- *Stemming*: Here, we present the stemming analysis, where we try to understand how directory names between different web applications differ.
- *Dataset Similarity*: Here, we analyze the degree of similarity between the different datasets presented.

5.2.1. Dataset Properties

We will now provide a detailed description of the four unique datasets that we have collected. For each dataset, we report the following properties:

- Number of Domains (`# Domains`). This refers to the total number of domains in each dataset, representing the number of different web applications for each dataset.
- Number of Paths (`# Paths`). This represents each dataset's total number of paths. For example, if a dataset contains two web applications, each with one domain (e.g., "domain1/home" and "domain2/home"), the number of paths is two, i.e., ["/home", "/home"].
- Average Number of Paths and Standard Deviation (`# Paths AVG` and `# Paths STD`). This is the average number (and standard deviation) of paths that each web application has for a given dataset. For example, if two web applications contain 3 URLs each, the average equals 3, and the standard deviation equals 0.
- Number of Unique Paths (`# Unique Paths`). This is the number of unique paths for a dataset. For example, if the dataset contains the samples "domain1/home/about" and "domain2/home/about", the unique paths are ["/home/about"].
- Number of Directories (`# Directories`). This is the total number of directories in a dataset. For example, given "domain1/home/about" and "domain2/home/news", the dataset contains four directories ["home", "about", "home", "news"].
- Number of Unique Directories (`# Unique Directories`). This is the total number of unique directories in a dataset. For example, given "domain1/home/about" and "domain2/home/news", the dataset has 3 unique directories ["home", "about", "news"].
- Depth of Paths (`Depth AVG` and `Depth STD`). Considering the *depth* metric we have introduced, for a given dataset, we analyze the depth of the

paths that it contains by average and standard deviation. For example, the URL “domain1/home/about” has a path with a depth equal to 2. We hypothesize that a directory with greater depth will likely be more specific and more correlated with the single web application. Consequently, it may be less common. This analysis can reveal interesting insights about web application structures and granularities between different datasets.

- Average Depth and Standard Deviation of URLs Similarities (`Similarity AVG` and `Similarity STD`). This property highlights the similarity between each pair of web applications for a given dataset. The metric considered is the Jaccard similarity previously introduced in Equation 3.1. Considering a web application as a set of the directories that compose its paths, we computed the Jaccard similarity between every pair of web applications. Then, we calculated the average similarity and the standard deviation. The wordlist selected to conduct a directory brute-force attack plays a pivotal role in its performance. Additionally, the wordlist effectiveness is strictly correlated with the directories of a web application, so this analysis can give us valuable insights into whether web applications that belong to a similar category have similar directories (thus, the same wordlist can be effective for many web applications).

Table 5.1 presents the statistics for each dataset.

	Dataset			
features	**UNI**	**HOS**	**COM**	**GOV**
# Domains	88	80	97	336
# Paths	209657	211911	147198	520571
# Paths AVG	2301	2584	1479	1507
# Paths STD	2906	2800	2360	2340
# Unique Paths	201768	205587	143067	502693
# Directories	203613	209945	143620	512595
# Unique Directories	171215	173394	106097	462812
Depth AVG	4.11	3.31	4.43	3.40
Depth STD	1.69	1.72	2.23	1.66
Similarity AVG	0.022	0.019	0.016	0.016
Similarity STD	0.031	0.017	0.023	0.024

Table 5.1: Summary statistics for the four datasets: universities [**UNI**], hospitals [**HOS**], companies [**COM**], and government [**GOV**].

The reported properties leave room for some interesting discussions.

Web applications from different datasets average quite different structures considering the number of web pages (shown in `# Paths AVG`). For example, universities and hospitals have considerably more paths than companies and governments on average.

Additionally, web applications in the same datasets tend to have a high variance of number of pages (`# Paths STD`), a limited difference between the number of directories and the number of unique directories (`# Directories` and `# Unique Directories`), and a Jaccard similarity close to zero (`# Similarity STD`). These results show how even web applications that belong to the same category are substantially different types of directories and structures.

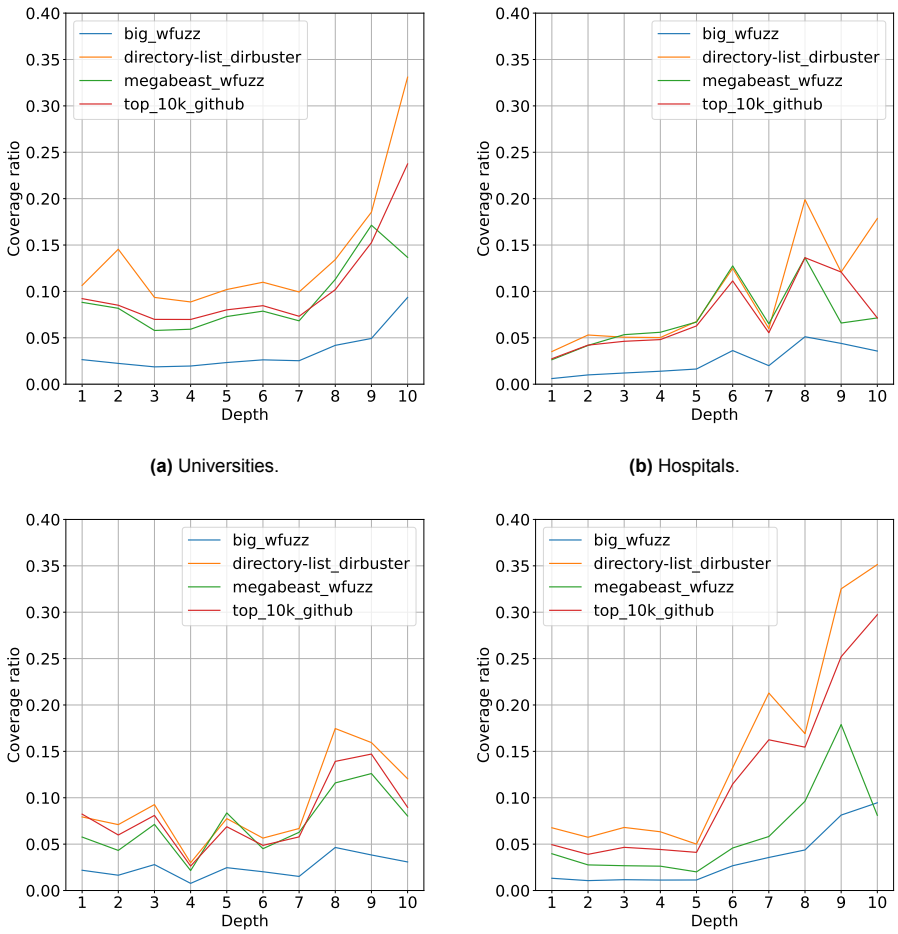
By considering these statistics collectively, we can clearly understand why di-

rectory enumeration is a complex task. Effective brute-force attacks might necessitate a vast number of requests for even a small number of successful discoveries.

5.2.2. Selected Wordlists Coverage Analysis

This analysis aims to analyze the coverage performance of the four selected wordlists at each depth level. We perform this analysis in the following way. For each dataset, we collect the directories at a given depth for each web application in the dataset. Then, we compute the coverage ratio by calculating how many directories are in the wordlists at the given depth and the total number of directories at that depth.

Figure 5.1 shows the results.



(a) Universities.

(b) Hospitals.

(c) Companies.

(d) Government.

Figure 5.1: Coverage ratio at increasing of depth for the four datasets: universities [**UNI**], hospitals [**HOS**], companies [**COM**], and government [**GOV**].

The coverage ratio between different datasets exhibits significant variance. It also reveals an unexpected trend: contrary to our initial hypothesis, the coverage ratio shows an upward trend with increasing depth, suggesting that directories at higher depths are more likely to be found on standard wordlists. However, this

However, this trend has to be carefully considered by taking into account the depth average and distribution reported in Table 5.1 and Figure 5.2. We can see

how the number of directories at that depth is significantly reduced, so the higher coverage ratio is relative to the corresponding limited number of directories.

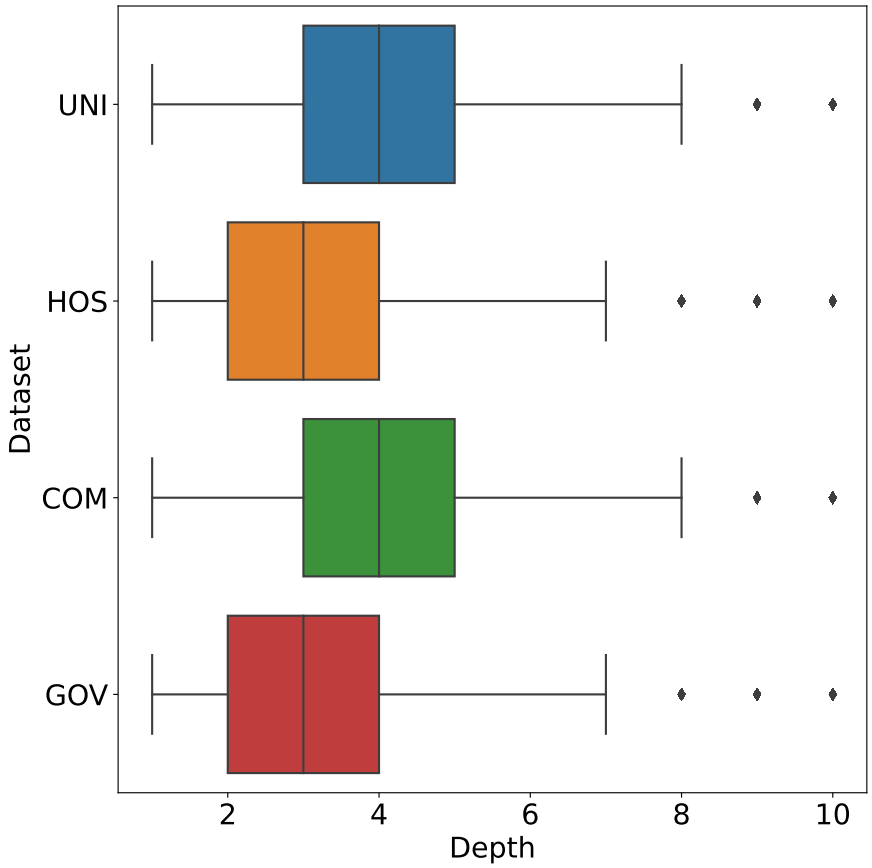


Figure 5.2: Depth distribution box plot for the four datasets: universities [**UNI**], hospitals [**HOS**], companies [**COM**], and government [**GOV**].

Additionally, the poor coverage at early depths is critical considering the directory brute-force attack methodology: in order to discover deeper directories, it is essential to discover the directories preceding them. Thus, the coverage ratio already points out how the standard methodology associated with the selected wordlists may obtain poor results.

5.2.3. Stemming Analysis

Stemming is a linguistic procedure that reduces words to their fundamental or root form, referred to as the stem, typically by eliminating common prefixes or suffixes. For instance, stemming transforms plurals into singulars (dogs $\rightarrow$ dog), converts verbs -ing form into the base form (running $\rightarrow$ run), and more.

Analyzing how stemming impacts the directories in our datasets can help design better attack approaches: having many directories corresponding to few common root forms may require a different strategy than having all directories with different roots.

In our research, we utilized the Porter-Stemmer algorithm [5] to investigate the impact of stemming on our datasets. Specifically, we analyzed how stemming can help reduce the directory dictionary (reducing the word embedding complexity in the language model) and what variations there are for the same roots.

Table 5.2 demonstrates how the directory dictionaries for different datasets would be reduced considering only the root forms of the directories. Stemming seems to have minimal impact, with an average dictionary reduction of 1.4%.

	Dataset			
features	**UNI**	**HOS**	**COM**	**GOV**
# Unique directories	171215	173394	106097	462812
# Unique Roots	168462	171371	104220	457912
Reduction	2753	2023	1877	4900
Reduction (%)	1.61%	1.17%	1.77%	1.06%

Table 5.2: Summary statistics after Stemming process for the four datasets: universities [**UNI**], hospitals [**HOS**], companies [**COM**], and government [**GOV**].

To further explore the impact of stemming, Figure 5.3 shows the distribution of how many roots are base forms of a given number of directories. Here again, we see how most roots correspond to only one word, with a few cases where one root corresponds to multiple directories.

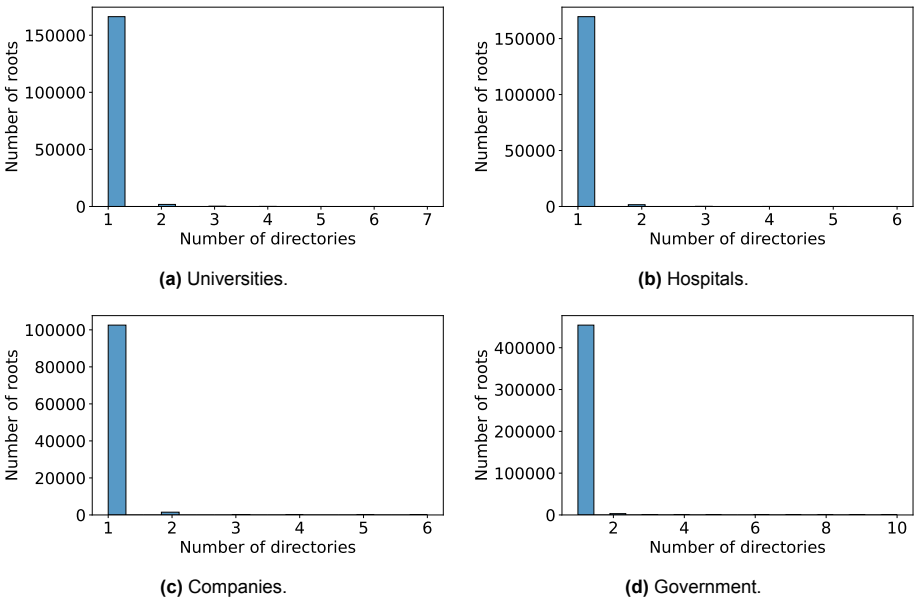


Figure 5.3: Distribution of how many roots correspond to a given number of directories for the four datasets: universities [**UNI**], hospitals [**HOS**], companies [**COM**], and government [**GOV**].

In the few cases where different directories correspond to the same root word, it is mainly due to slight variations in naming conventions, capital/lowercase letters or singular and plural forms. Here are a couple of examples:

1. "*articl*" corresponding to "*article*" in 33.5% of cases, "*articles*" in 14.4%, "*Article*" in 47.3%, "*Articles*" in 4.7%, and "*ARTICLE*" in 0.01%.
2. "*project*" corresponding to "*project*" in 46.78% of cases , "*projects*" in 53.13, "*Projected*" in 0.04% and "*Projects*" in 0.04%.

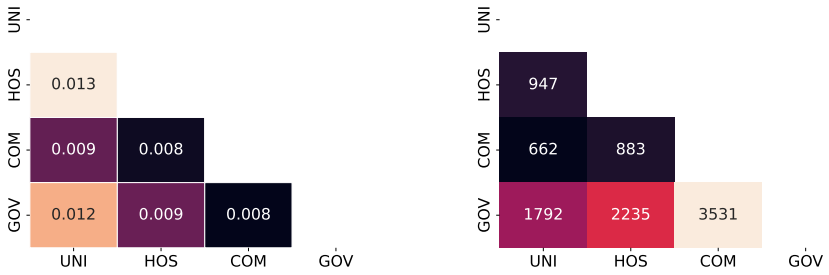
In conclusion, these analyses highlight how the directories are substantially different from each other, with naming conventions or singular/plural forms that have a marginal impact. Thus, our focus will be on exploiting relations between directories and subdirectories and the context extrapolated from word embeddings without tailored strategies to generate different variations of the same directory that may represent other valid directories.

5.2.4. Dataset Similarity Analysis

Finally, we assess the dataset's similarity, analyzing the similarity between each pair of datasets. We employ two metrics for this purpose:

- The Jaccard similarity between datasets. More specifically, for a given pair of datasets, we retrieved the unique directories for each of them, and then we computed the Jaccard similarity of the two sets.
- The number of common paths between the datasets.

The results are illustrated in Figure 5.4.



(a) Jaccard similarities between datasets wordlist.

(b) Number of common paths.

Figure 5.4: Similarity analysis for the four dataset: universities [**UNI**], hospitals [**HOS**], companies [**COM**], and government [**GOV**].

The low Jaccard similarities underscore the first notable result, indicating that each dataset comprises distinct directories. In other words, the structures of university websites are substantially different from those of hospital websites. This observation can be extended to any pair of datasets we used.

Another interesting discovery is the number of common paths among different datasets. For instance, government and company websites exhibit significant overlap compared to other datasets. However, when considering the number of paths presented in Figure 5.1, we can clearly see that these numbers are marginal compared to the total number of paths for each dataset.

In conclusion, considering the low similarity between datasets and between web applications belonging to the same datasets, designing a novel methodology for directory enumeration is a complex and not trivial task.

6

Results

In this chapter, we present the experimental settings, the results, and a discussion of the results obtained.

In the experimental settings, we describe our testbed, the attack simulations we will conduct, the evaluation metrics, and the validation of the choice of the Language model architecture.

6.1. Experimental Settings

In the experimental settings, we describe our testbed, the attack simulations we will conduct, the evaluation metrics, and the validation of the choice of the Language model architecture.

6.1.1. Attack Testbed

Our testbed aims to be as comprehensive as possible to test the standard and the novel attack methodologies proposed in this project.

Before setting up what offline simulations to conduct, we split our datasets into training, validation and testing sets. Specifically, we performed a 70-10-20 split ratio. Note that the split is done from a domain perspective, and then the paths associated with each domain are split accordingly. By doing this, all the data of a given web application will appear either in training, validation or test set, avoiding possible *data snooping* [4].

This dataset split is essential for our novel approaches: the Language model necessitates a large number of samples to be trained with (we achieve this by using the paths from all the web applications assigned to the training set from every dataset, and we do the same with the validation set), and the probabilistic approach needs the prior knowledge to build the weighted training/wordlist tree which is essential for the dynamical decision-making strategy.

Additionally, the testing environment is based on offline simulations utilizing the virtual reconstructed filesystems of the web applications assigned to the testing set. This choice allows us to conduct multiple attack simulations with different attack methodologies and parameters without harming real web applications launching real-time directory brute-force attacks.

Finally, since attacks can virtually send unlimited requests, we decided to set a requests budget of 100,000 requests that each attack simulation can spend before termination. This choice aligns with our goal of maximizing successful responses while minimizing the request volume.

6.1.2. Language Model Validation

The choice of an appropriate language model is vital for the performance of the LM-based approach. Through the training set, the Language model can learn meaningful associations between directories, extrapolating the context.

The validation of the best hyperparameters that help achieve this can significantly impact the results that we will obtain with this methodology in the simulations. We employed a grid-search validation over the following hyperparameters to select the model:

- *Input data manipulation*. Firstly, we identified hyper-parameters correlated with the training samples: the maximum length a path can have and the minimum frequency that a directory must have not to be discarded and marked as "Unknown." For the former, defined as `max_depth`, we chose [5, 10] as possible values. For the latter, defined as `min_freq`, we chose [3, 5] as possible values.
- *LM architecture*. Secondly, we identified hyper-parameters related specifically to the neural network architecture. Specifically, these parameters are: `embedding_size` **[ES]** = [128, 256, 512], `n_layers` **[NL]** (number of layers in the LSTM) = [2, 3, 4], `dropout_rate` **[DR]**=[0.2, 0.4, 0.6].

In the training, we have also used an early stopping mechanism, as described in Section 4.5 with a patience of 10 epochs. We used the Adam [10] optimizer and the CrossEntropy loss function in the learning phase.

Then, we selected the model that obtained the lowest loss in the validation set during the training phase as a metric for selecting the best model.

Lastly, we decided to test an additional parameter that can define the performance of the LM attack methodology. This parameter, defined as `topPredicts` introduced in Algorithm 4, is the number of predictions the Language model outputs each time a new HTTP response is received. We tested the values [100, 250, 500, 750, 1000, 2000, 5000, 10000] for this parameter.

6.1.3. Evaluation Metrics

We decided to utilize two evaluation metrics:

- **Average Successful Responses**. This metric represents the average amount of successful responses received during the attack simulations (each response is associated with a newly discovered directory) over a testing dataset that consists of multiple web applications
- **Mean efficiency ratio in bins**. This involves calculating the mean of the total number of directories successfully discovered for each web application tested within a specific range of requests. The ranges of requests we consider are [0-100, 101-1000, 1001-10000, 10001-50000, and 50001-100000]. This metric allows us to evaluate the efficiency of our dynamic decision-making strategy and how the two novel approaches perform compared to the standard attack methodology.

In the evaluation of the attacks, we did not consider the execution time as a metric for the following reasons:

- The execution time depends on several implementation factors, such as the number of threads used in each attack and the programming language with which the attack is implemented.
- The time between a request being sent and a response being received can vary between web applications.
- Our testing environment comprises offline simulations. Thus, the execution time would not be meaningful.

6.2. Experimental Results

This section presents an analysis of the *topPredicts* parameter and an overview of the results obtained in the two metrics we selected.

6.2.1. K Top-Predictions Analysis

Testing the parameter `topPredicts` of the LM-based approach has revealed an interesting trend.

Figure 6.1 shows the average successful responses received in the general testing dataset (considering all the web applications assigned to the testing set) at the varying of `topPredicts`.

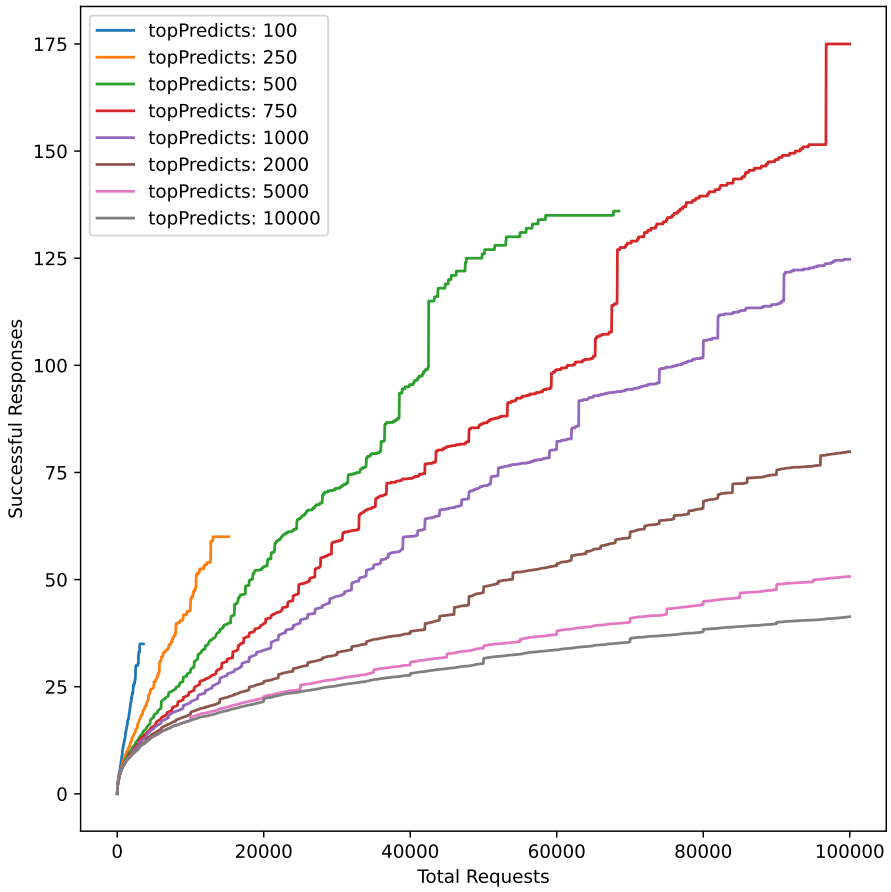


Figure 6.1: Evolution of average successful responses for different topPredicts values.

The average number of successful responses received decreases as the number of the most probable predictions generated each time increases.

Moreover, while a smaller `topPredicts` value results in a higher initial average number of successful responses, the simulations terminate before exhausting the requests budget since the attacks have no more requests to send.

This trend highlights a crucial aspect when selecting an appropriate value for this parameter to maximize the outcomes and entirely use the pre-set request budget.

If our budget is constrained, it may be preferable to set `topPredicts` to a lower number, as the model achieves the highest number of successful responses using fewer requests. Conversely, larger values such as 500, 750, and 1000

are more suitable when the budget permits a comprehensive search.

6.2.2. Average Successful Responses Results

Table 6.1 presents the comprehensive results derived from our experiments, considering different datasets, wordlists, and attack methodologies. The LM-based attack surpasses all other baseline methods across all datasets, thereby underscoring the superiority of the Language model-based technique over standard and probabilistic attack methodologies.

Interestingly, the LM's performance is not uniformly distributed across all datasets. For example, the LM encounters difficulties with websites related to universities and companies, while it exhibits robustness with websites associated with hospitals and government entities.

The probabilistic-based method demonstrates a good performance improvement over standard directory brute-force attacks. This is probably due to the enforced request budget that limits the discovery of the standard attacks.

Regarding the standard methodologies, the breadth-first strategy emerges as a better attack strategy than the depth-first.

Lastly, it is interesting to note how requests budget affects the results obtained from different wordlists. The standard and the probabilistic approaches achieve almost the same results using the smallest wordlist `big_wfuzz`. This happens because, in the simulations, the attacks come to exhaust the possible requests to be sent before reaching the set budget. In contrast, in the other three wordlists, we can see how the results change depending on the attack methodologies: here, the strategy by which it is decided which request to send is crucial in getting better or worse results before exhausting the requests budget.

Wordlist	*Dataset*				
	UNI	**HOS**	**COM**	**GOV**	**ALL**
Breadth					
big_wfuzz	28.0	22.0	27.0	35.0	35.0
directory-list_dirbuster	8.0	10.3	9.6	11.8	10.5
megabeast_wfuzz	10.5	11.6	11.0	12.4	11.7
top_10k_github	21.3	42.6	26.8	27.0	28.6
Depth					
big_wfuzz	28.0	22.0	27.0	33.0	33.0
directory-list_dirbuster	0.5	0.5	0.7	0.4	0.5
megabeast_wfuzz	2.6	2.9	2.7	2.7	2.7
top_10k_github	10.1	10.1	10.1	10.0	10.1
Probability					
big_wfuzz	28.0	22.0	27.0	34.5	34.5
directory-list_dirbuster	14.0	13.1	11.1	17.3	25.4
megabeast_wfuzz	12.5	13.9	11.8	13.8	13.8
top_10k_github	23.4	42.9	26.1	27.1	26.7
train-set	31.9	60.4	27.6	30.8	42.5
LM					
train-set	**90.0**	**175.0**	**89.0**	**128.0**	**175.0**

Table 6.1: Average successful responses for each approach achieved for different test-sets at the varying of the datasets. In bold the best results.

6.2.3. Mean Efficiency Ratio Results

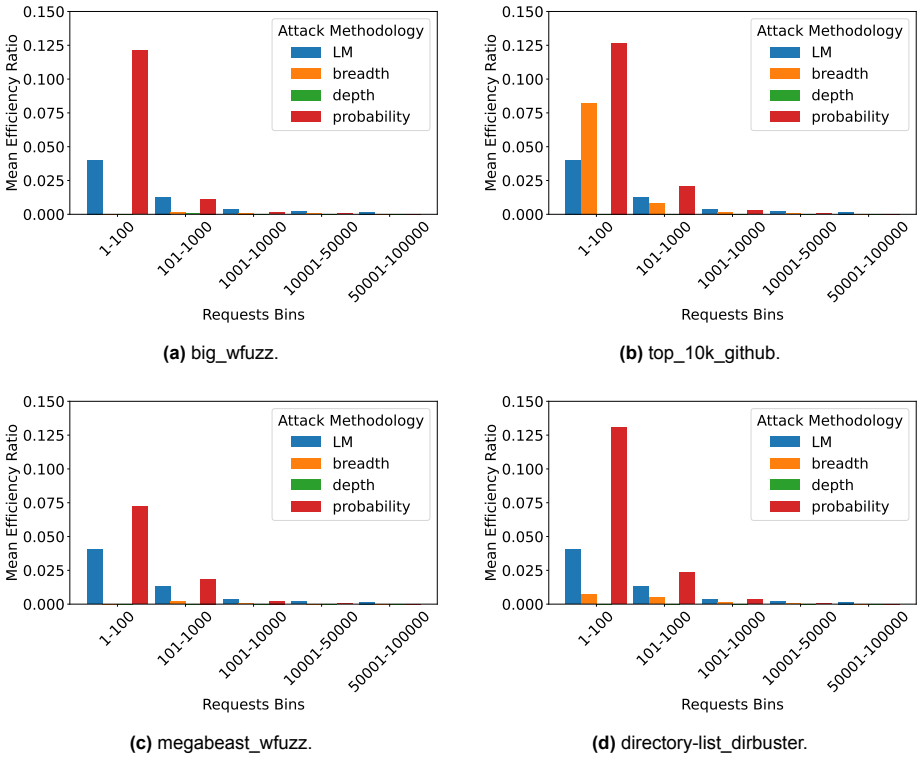
The second metric considered, the mean efficiency ratio, can give us valuable insight into the efficiency of the requests sent in the early, mid, and late stages of directory brute-force attacks.

This analysis can, in particular, show how the dynamical decision-making strategies in both proposed approaches compare to the sequential sending of requests employed by the standard attack methodology.

Figures 6.2, 6.3, 6.4, 6.5 and 6.6 present a comprehensive overview of the mean efficiency ratio considering different datasets, wordlists and attack methodologies.

In the early stages (1-100, 101-1000) of the attack simulations, the probabilistic approach emerges as the most efficient approach in the vast majority of cases. Specifically considering the 1-100 and 101-1000 bins, the probabilistic approach is more efficient than the depth-first approach in 100% of the cases, the breadth-first approach in 94% of the cases, and the LM-based approach in 81% of the cases.

On the other hand, the LM-based approach is generally less efficient than the probabilistic approach in the early stages, except for some cases with equal or slightly better efficiency. However, this approach manages to be more consistent in the middle (1001-10,000) or later (10001-50000, 50000-100000) stages of the attack, outperforming other attack methodologies. This is also validated by correlating the results from the previous metric. Specifically considering the 10001-50000 and 50001-10000 bins (related to 90% of the total requests sent) the LM-based approach outperforms the other three attack methodologies in terms of efficiency in 100% of the cases.



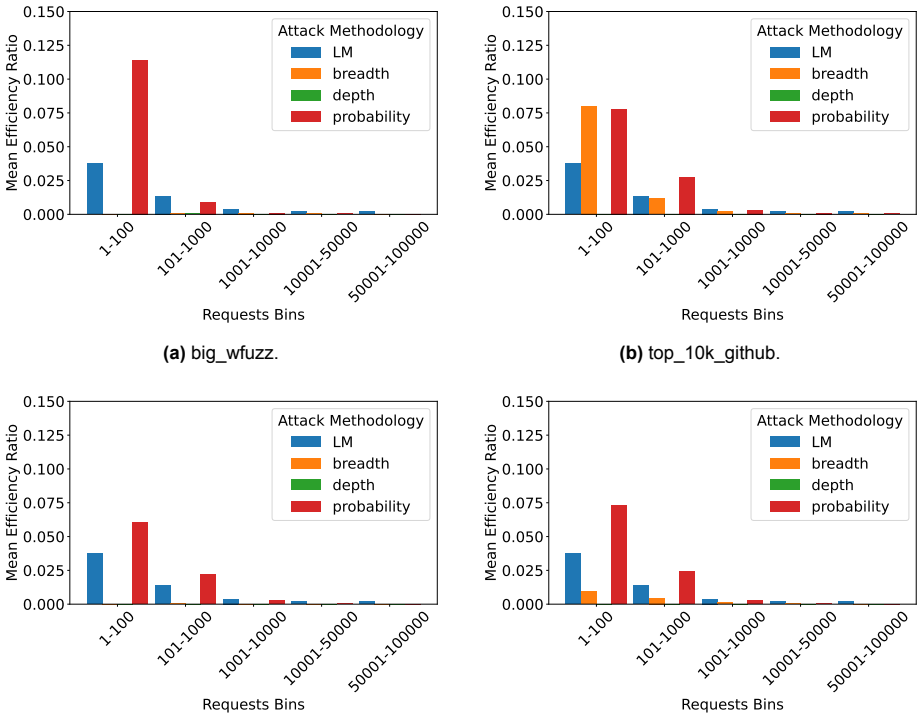
(a) big_wfuzz.

(b) top_10k_github.

(c) megabeast_wfuzz.

(d) directory-list_dirbuster.

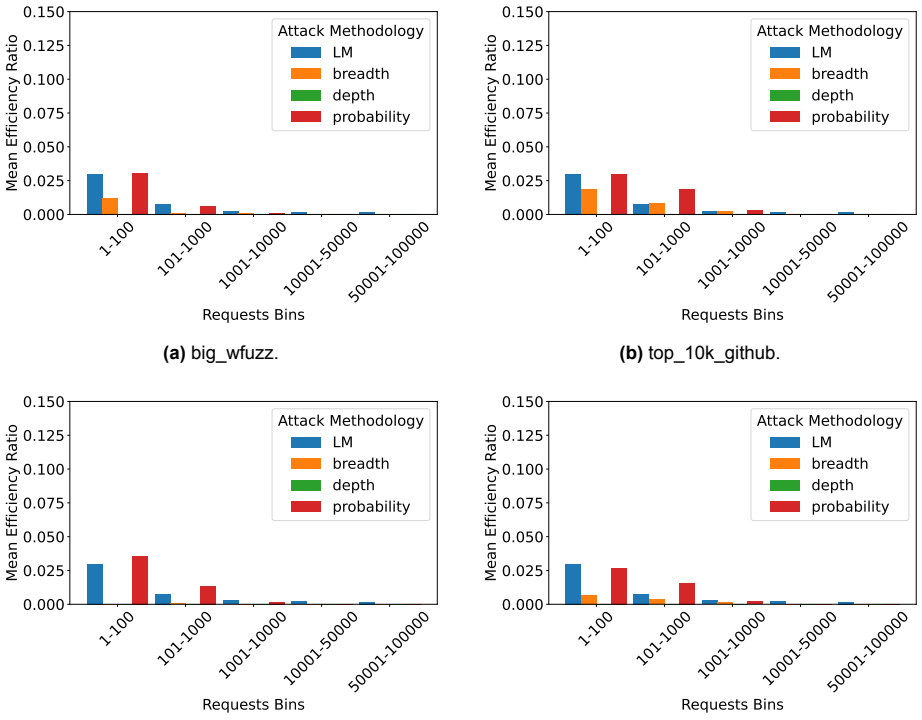
Figure 6.2: Mean Efficiency Ratio of the four approaches on different bins for **Universities dataset** [**UNI**] and 4 wordlists: `big_wfuzz` [**BW**], `top_10k_github` [**GH**], `megabeast_wfuzz` [**MW**], and `directory-list_dirbuster` [**DB**].



(a) big_wfuzz. **(b)** top_10k_github.

(c) megabeast_wfuzz. **(d)** directory-list_dirbuster.

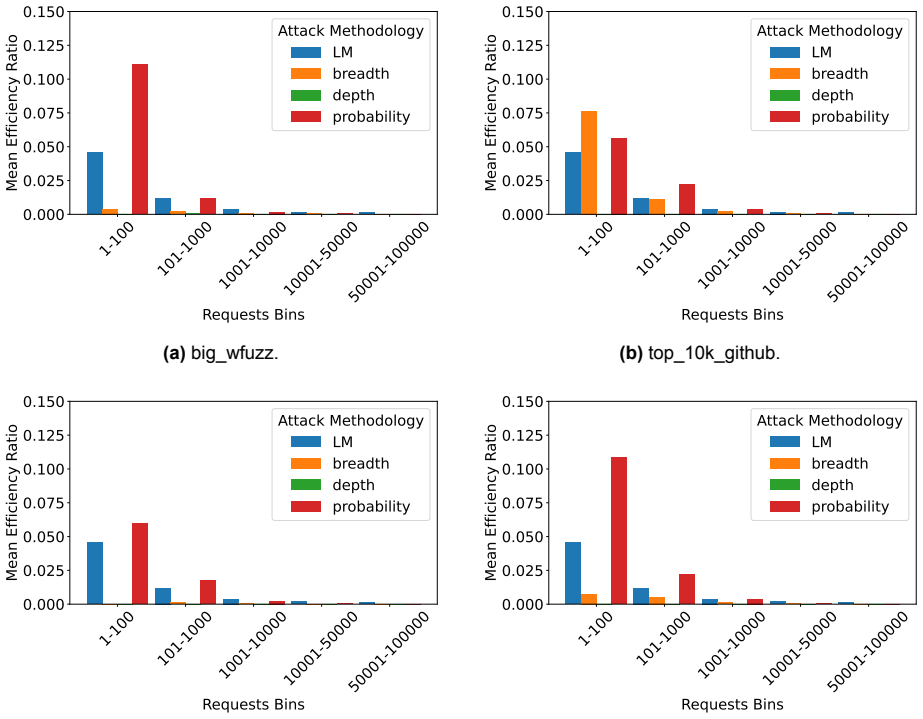
Figure 6.3: Mean Efficiency Ratio of the four approaches on different bins for **Hospitals dataset** [**HOS**] and 4 wordlists: `big_wfuzz` [**BW**], `top_10k_github` [**GH**], `megabeast_wfuzz` [**MW**], and `directory-list_dirbuster` [**DB**].



(a) big_wfuzz. **(b)** top_10k_github.

(c) megabeast_wfuzz. **(d)** directory-list_dirbuster.

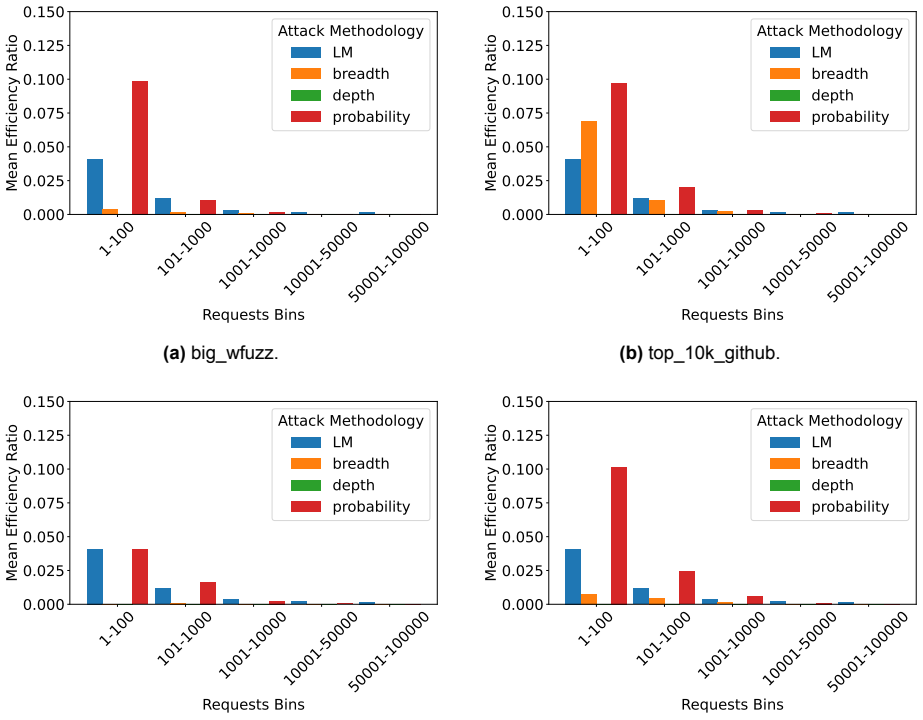
Figure 6.4: Mean Efficiency Ratio of the four approaches on different bins for **Companies dataset** [**COM**] and 4 wordlists: `big_wfuzz` [**BW**], `top_10k_github` [**GH**], `megabeast_wfuzz` [**MW**], and `directory-list_dirbuster` [**DB**].



(a) big_wfuzz. **(b)** top_10k_github.

(c) megabeast_wfuzz. **(d)** directory-list_dirbuster.

Figure 6.5: Mean Efficiency Ratio of the four approaches on different bins for **Government dataset** [**GOV**] and 4 wordlists: `big_wfuzz` [**BW**], `top_10k_github` [**GH**], `megabeast_wfuzz` [**MW**], and `directory-list_dirbuster` [**DB**].



(a) big_wfuzz. **(b)** top_10k_github.

(c) megabeast_wfuzz. **(d)** directory-list_dirbuster.

Figure 6.6: Mean Efficiency Ratio of the four approaches on different bins for **general dataset** [**ALL**] and 4 wordlists: `big_wfuzz` [**BW**], `top_10k_github` [**GH**], `megabeast_wfuzz` [**MW**], and `directory-list_dirbuster` [**DB**].

6.3. Results Discussion

In this section, we discuss the results obtained, analyse what improvements have been made, and provide possible use cases and explanations of the novel approaches.

6.3.1. Results Analysis

The two methodologies proposed demonstrate significant enhancements in both metrics evaluated. Among the two conventional strategies, the breadth-first technique proves to be superior.

The probabilistic method enhances the performance of the breadth-first approach in 65% of cases when considering the four default wordlists. In the remaining cases it achieves equivalent or marginally lower results. Depth-first strategies are surpassed in 100% of instances. Specifically, when considering the breadth-first approach, the probability-based method using the train-set wordlist exhibits the following average improvements: University +141%, hospital +281%, companies +85%, government +78%, and **ALL** +159%.

The primary advantage of the probabilistic method is its ability to generate successful responses using a limited number of requests, which significantly outperforms all other methods. The efficiency metric also supports this choice.

The Language-based approach surpasses the standard approach in 100% of the simulations and emerges by far as the best attack methodology. The LM-based model approach demonstrates the following average improvements over the breadth-first baselines: University +582%, hospital +1004%, companies +499%, government +639%, and **ALL** +969%.

These results highlight how an attacker can leverage prior knowledge to boost the performance of standard directory brute-force attacks.

When considering the relative improvements of the two proposed novel approaches, the probabilistic approach performs better in University and Hospital datasets, and the LM-based approach performs exceptionally in the hospital dataset.

This discrepancy may be due to the more significant heterogeneity of web applications contained in companies and government: companies of government sites in different sectors may deal with different topics and have more diverse folders among the various web applications, compared to universities and hospitals that might instead have more similar folders and structure.

6.3.2. Exploiting The Context

In a directory brute-force attack, the improvements shown by the novel approaches show how an essential component that allows dynamically selecting which request to send is the context to which the directories belong.

The capacity of embeddings to extrapolate the context from web application paths and generalize is crucial for predicting possible existing directories. In particular, the results underscore how the Language Model (LM) approach effectively leverages the context derived from embeddings to yield significantly superior results compared to other methods.

This can be observed by presenting two examples: given two directories *"article"* and *"about"*, we employ Cosine similarity to measure the top 10 directories with the highest cosine similarity, indicating belonging to a similar context.

1. *article*: ('stories', 0.48), ('academics', 0.43), ('press-release', 0.39), ('press-releases', 0.38), ('video', 0.32), ('authors', 0.32), ('spotlight', 0.32), ('articles', 0.31), ('case', 0.3), ('impact', 0.29).

2. *about*: ('locations', 0.79), ('about-us', 0.75), ('research', 0.75), ('programs', 0.74), ('conditions', 0.7), ('services', 0.68), ('resources', 0.68), ('alumni', 0.68), ('careers', 0.67), ('contact', 0.66).

In both instances, we observe that the words determined as similar by the embeddings represent the same word but with slight variations, such as '`about`' with '`about-us`' or '`article`' with '`articles`'. Additionally, we find other words that relate to the context created by the word under consideration, such as '`authors`' or '`stories`' for '`article`'.

These two examples demonstrate how language models can leverage embeddings to extrapolate the context and attest to their capability to predict valid directories that adhere to recurring patterns. Regarding this, it is also interesting to report some predicted subdirectories given two distinct paths:

1. path: */campus-life-events/calendar*: ["05", "06", "08", "11", "may", "jun"] are some of the most probable subdirectories predicted, which are correlated to days or months of a calendar.
2. path: */media*. ["press-releases", "news"] are some of the most probable subdirectories predicted. These two words are correlated to the same context of "media", and the corresponding paths "*/media/press-releases*" and "*/media/news*" are also URL paths of the training dataset.

Again, these examples show the power of exploiting the context to enhance the directory enumeration process through a dynamic and smart approach that maximizes directory discoveries while reducing the number of requests needed.

7

Conclusion

The current directory brute-force attacks are widely recognized for their inefficiency. This is primarily due to their reliance on standard brute-force strategies and strong dependency on a wordlist. These two components lead to a massive number of requests being sent for a relatively small number of successful discoveries that are entirely dependent on the selected wordlist for the attack.

In this study, we explored the possibility of leveraging prior knowledge to enhance the efficiency and effectiveness of these attacks. To this end, we introduced two novel attack methodologies where prior knowledge is a pivotal component: a probabilistic approach that primarily exploits information between directories and subdirectories and a Language model-based approach where a Language model is employed to predict probable directories.

We collected a novel dataset comprising approximately 600 different web applications from different domains (universities, hospitals, companies, and government web applications) and more than 1 million URLs. We then used this dataset to extensively test our novel approaches.

The results of our experiments underscored the superiority of our proposed methods. The LM-based approach outperformed the standard directory brute-force methodologies in every scenario, registering an average performance increase of 969%. Moreover, the probabilistic approach proved to be effective and employable in scenarios where the requests budget is limited and the attack has to be stealthy.

7.1. Future Works

Unfortunately, during our testing phases, the results we obtained led us to choose another approach based on a language model, which ended up being the second improved approach of this project. The research in this thesis project establishes a foundation for numerous potential directions for future exploration. The application of Artificial Intelligence in crafting sophisticated attacks is a rapidly evolving and expanding field in cybersecurity, particularly with the swift advancement of Artificial Intelligence.

Future work could focus on improving our LM-based architecture by using advanced models such as attention mechanisms [39], or even Large Language Models [7]. These models' augmented comprehension of context and semantics could considerably refine the procedure of predicting web application structures and directories.

Furthermore, since directory predictions can be limited by the language in which

a web application is developed, there is potential for exploring the use of pre-trained and cross-lingual embeddings [40, 44] to extrapolate the context and predict directories without having more language restrictions.

Another possible research direction that could provide a significant contribution is developing a language model explicitly trained on directories and files typically associated with vulnerabilities. By fine-tuning the model on these vulnerabilities, the model could aid in preemptively identifying potential security risks, thereby contributing to more proactive cybersecurity measures to secure web applications.

These areas of future work hold the potential to significantly influence the development of more secure web applications and present new security challenges to the usage of language models.

7.1.1. Seq2Seq Approach

After presenting some possible research directions that can follow up on the research in this project, it can also be helpful to report some other experiments we have conducted with other approaches.

Specifically, we experimented with using another neural architecture instead of the language model called Seq2Seq [37]. This model consists of two main components:

- Encoder: a RNN layer (in our case specifically a LSTM) that processes input sequence and returns the internal state representing the context.
- Decoder: another RNN layer (LSTM also in this case for our test case) that uses the internal state provided by the encoder and predicts the following sequences to the input sequence as output.

We chose this type of architecture because it suited our goal. Both models would be trained on the prior knowledge collected. During an attack, the encoder would be responsible for extrapolating the context from the URL given to it as input, while the decoder would use the extrapolated context to predict possible directories that would form new URLs.

The intuition behind this idea is simple: suppose we have a web application that has "*login*" and "*/user/register*" as valid directories. The Seq2Seq model gets fed with multiple paths, and the encoder is responsible for extrapolating the context from these paths. Then, the decoder can use this context to understand what possible directories may be in the web application, considering the context extrapolated so far. In this case, if we successfully discover "*login*", the context will likely lead the decoder to predict that also "*register*" is a valid directory in the web application, even if it is located in different areas or depths in the application filesystem. This ability to adapt the context feeding the valid paths discovered during the attack can be valuable to accurately predict directories and paths that are unseen in prior knowledge.

However, the initial dissatisfying results we obtained using it and the idea of employing a language model that would have integrated seamlessly with the algorithm already implemented for the probabilistic approach led us to discard this architecture to focus on the language model approach.

References

[1] Ayham Alomari et al. “Deep reinforcement and transfer learning for abstractive text summarization: A review”. In: *Computer Speech & Language* 71 (2022), p. 101276.

[2] Yazan Ahmad Alsariera et al. “Ai meta-learners and extra-trees algorithm for the detection of phishing websites”. In: *IEEE access* 8 (2020), pp. 142532–142542.

[3] Diego Antonelli et al. *Leveraging AI to optimize website structure discovery during Penetration Testing*. 2021. arXiv: `2101.07223 [cs.CR]`.

[4] Daniel Arp et al. “Dos and don’ts of machine learning in computer security”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3971–3988.

[5] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. ” O’Reilly Media, Inc.”, 2009.

[6] Philip Bontrager et al. “Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution”. In: *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*. IEEE. 2018, pp. 1–9.

[7] Yupeng Chang et al. “A survey on evaluation of large language models”. In: *ACM Transactions on Intelligent Systems and Technology* (2023).

[8] Kyunghyun Cho et al. “On the properties of neural machine translation: Encoder-decoder approaches”. In: *arXiv preprint arXiv:1409.1259* (2014).

[9] Michael Chui et al. “The state of AI in 2023: Generative AI’s breakout year”. In: (2023).

[10] P Kingma Diederik. “Adam: A method for stochastic optimization”. In: *(No Title)* (2014).

[11] Gerard Verweij Dr. Anand S. Rao. *Sizing the prize. What’s the real value of AI for your business and how can you capitalise?* Tech. rep. PricewaterhouseCoopers (PwC), 2020.

[12] Peng Gao et al. “Enabling efficient cyber threat hunting with cyber threat intelligence”. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. 2021, pp. 193–204.

[13] Abdulrahman Al-Hababi and Sezer C Tokgoz. “Man-in-the-middle attacks to detect and identify services in encrypted network flows using machine learning”. In: *2020 3rd International Conference on Advanced Communication Technologies and Networking (CommNet)*. IEEE. 2020, pp. 1–5.

[14] Raza Hasan et al. “Artificial intelligence based model for incident response”. In: *2011 International Conference on Information Management, Innovation Management and Industrial Engineering*. Vol. 3. IEEE. 2011, pp. 91–93.

[15] Ying He et al. “AI Based Directory Discovery Attack and Prevention of the Medical Systems”. In: *2022 Computing in Cardiology (CinC)*. Vol. 498. IEEE. 2022, pp. 1–4.

[16] Sepp Hochreiter. “The vanishing gradient problem during learning recurrent neural nets and problem solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[17] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[18] Mohammad Mehdi Hosseini and Masood Parvania. “Artificial intelligence for resilience enhancement of power distribution systems”. In: *The Electricity Journal* 34.1 (2021), p. 106880.

[19] Armand Joulin et al. “Bag of tricks for efficient text classification”. In: *arXiv preprint arXiv:1607.01759* (2016).

[20] Nektaria Kaloudi and Jingyue Li. “The ai-based cyber threat landscape: A survey”. In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–34.

[21] Nabeel Sabir Khan, Adnan Abid, and Kamran Abid. “A novel natural language processing (NLP)–based machine translation model for English to Pakistan sign language translation”. In: *Cognitive Computation* 12 (2020), pp. 748–765.

[22] Yuanzhang Li et al. “A feature-vector generative adversarial network for evading PDF malware classifiers”. In: *Information Sciences* 523 (2020), pp. 38–48.

[23] Yunji Liang et al. “Behavioral biometrics for continuous authentication in the internet-of-things era: An artificial intelligence perspective”. In: *IEEE Internet of Things Journal* 7.9 (2020), pp. 9128–9143.

[24] Qian Liu et al. “You impress me: Dialogue generation via mutual persona perception”. In: *arXiv preprint arXiv:2004.05388* (2020).

[25] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[26] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[27] Yisroel Mirsky et al. “The threat of offensive ai to organizations”. In: *Computers & Security* 124 (2023), p. 103006.

[28] Edmilson Morais et al. “Speech emotion recognition using self-supervised features”. In: *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2022, pp. 6922–6926.

[29] Steve Morgan. *Cybercrime To Cost The World 8 Trillion Annually In 2023 — cybersecurityventures.com*. `https://cybersecurityventures.com/cybercrime-to-cost-the-world-8-trillion-annually-in-2023/`. [Accessed 07-05-2024]. 2022.

[30] Sungyup Nam et al. “Recurrent gans password cracker for iot password security enhancement”. In: *Sensors* 20.11 (2020), p. 3106.

[31] Adam Pauls and Dan Klein. “Faster and smaller n-gram language models”. In: *Proceedings of the 49th annual meeting of the Association for Computational Linguistics: Human Language Technologies*. 2011, pp. 258–267.

[32] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[33] Dan Petro and Ben Morris. “Weaponizing machine learning: Humanity was overrated anyway”. In: *DEF CON* 25 (2017).

[34] Fabio Petroni et al. “Language Models as Knowledge Bases?” In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 2463–2473. doi: 10.18653/v1/D19-1250. url: https://aclanthology.org/D19-1250.

[35] Robin M Schmidt. “Recurrent neural networks (rnns): A gentle introduction and overview. arXiv 2019”. In: *arXiv preprint arXiv:1912.05911* (1912).

[36] Tobias Schnabel et al. “Evaluation methods for unsupervised word embeddings”. In: *Proceedings of the 2015 conference on empirical methods in natural language processing*. 2015, pp. 298–307.

[37] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems* 27 (2014).

[38] Khoa Trieu and Yi Yang. “Artificial intelligence-based password brute force attacks”. In: (2018).

[39] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[40] Ivan Vulić and Marie-Francine Moens. “Monolingual and cross-lingual information retrieval models based on (bilingual) word embeddings”. In: *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*. 2015, pp. 363–372.

[41] Kevin Williams. *’Cyber-physical attacks’ fueled by AI are a growing threat, experts say — cnbc.com*. https://www.cnbc.com/2024/03/03/cyber-physical-attacks-fueled-by-ai-are-a-growing-threat-experts-say.html. [Accessed 07-05-2024]. 2024.

[42] Dali Zhu et al. “DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data”. In: *2017 IEEE symposium on computers and communications (ISCC)*. IEEE. 2017, pp. 438–443.

[43] Jinhua Zhu et al. “Incorporating bert into neural machine translation”. In: *arXiv preprint arXiv:2002.06823* (2020).

[44] Will Y Zou et al. “Bilingual word embeddings for phrase-based machine translation”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1393–1398.