Offensive AI

Enhancing Directory Brute-forcing Attack with the Use of Language Models

Castagnaro, Alberto; Conti, Mauro; Pajola, Luca

DOI
10.1145/3689932.3694770

Publication date
2024

Document Version
Final published version

Published in
AISec '24

Offensive AI: Enhancing Directory Brute-forcing Attack with the Use of Language Models

Alberto Castagnaro
Delft University of Technology
Delft, The Netherlands
A.Castagnaro@student.tudelft.nl

Mauro Conti*†
University of Padua
Padua, Italy
mauro.conti@unipd.it

Luca Pajola†
University of Padua
Padua, Italy
luca.pajola@unipd.it

Abstract

Web Vulnerability Assessment and Penetration Testing (Web VAPT) is a comprehensive cybersecurity process that uncovers a range of vulnerabilities which, if exploited, could compromise the integrity of web applications. In a VAPT, it is common to perform a *Directory brute-forcing Attack*, aiming at the identification of accessible directories of a target website. Current commercial solutions are inefficient as they are based on brute-forcing strategies that use wordlists, resulting in enormous quantities of trials for a small amount of success.

Offensive AI is a recent paradigm that integrates AI-based technologies in cyber attacks. In this work, we explore whether AI can enhance the directory enumeration process and propose a novel Language Model-based framework. Our experiments – conducted in a testbed consisting of 1 million URLs from different web application domains (universities, hospitals, government, companies) – demonstrate the superiority of the LM-based attack, with an average performance increase of 969%.

CCS Concepts

• **Security and privacy** → **Penetration testing**; • **Computing methodologies** → *Natural language generation.*

Keywords

Offensive AI, Language Model, Web Security, Penetration Test

ACM Reference Format:
Alberto Castagnaro, Mauro Conti, and Luca Pajola. 2024. Offensive AI: Enhancing Directory Brute-forcing Attack with the Use of Language Models. In *Proceedings of the 2024 Workshop on Artificial Intelligence and Security (AISec '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3689932.3694770

1 Introduction

In its most general sense, hacking refers to modifying or manipulating a system's features to achieve a goal outside of the creator's original purpose. While often associated with illegal cyber activities, hacking can also be performed ethically, with permission, to improve system security and uncover vulnerabilities that can be fixed before malicious actors exploit them. Directory enumeration is a critical component of security assessments. It involves identifying accessible directories, files, and web paths in a web application. Effective discovery attacks, such as directory brute-forcing, can uncover hidden directories and files that may contain sensitive data or critical functionalities.

Offensive AI uses artificial intelligence technologies to conduct or enhance cyber attacks [10, 13]. This emerging field combines AI's adaptability and learning capabilities with traditional attack vectors, creating more sophisticated and automated threats. Offensive AI can rapidly analyze vast amounts of data, adapt to defensive measures, and execute attacks with increased speed and complexity.

Contribution. This paper contributes to the field by presenting a novel approach that leverages Language Models (LMs) to enhance the efficiency and effectiveness of directory brute-forcing attacks. Our method builds on prior knowledge retrieved by different web applications and then exploits embeddings to extrapolate the context of different words that form web paths and language models to generate new possible URL (Uniform Resource Locator) paths that can be used to send requests. Our contributions are summarized below:

(1) We designed a novel dataset containing 4 distinct types of applications that are often the targets for attacks, i.e., commercial, government, hospital, and universities, for a total of 1 million of URLs.
(2) We propose two novel directory brute-forcing attacks that leverage prior knowledge: a probabilistic and a Language Model-based approach.
(3) A systematic evaluation highlights the superiority of prior knowledge approaches compared to baselines. LM-based attacks outperform all 8 proposed baselines, with an average performance increase of 969%. On the other hand, probabilistic approaches show high performance when the budget of spendable requests is limited and, therefore, optimal for stealthier attacks.

We provide the code to replicate our experiments at: https://github.com/spritzmatterorg/LM-Directory-Bruteforcing.

Ethical Disclaimer. The techniques and methods discussed in this paper 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

*Also with Delft University of Technology.
†Also with Spritz Matter Srl.

AISec '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1228-9/24/10
https://doi.org/10.1145/3689932.3694770

dataset and code. Researchers willing to reproduce our experiment are invited to contact the authors.

2 Background

This section describes the theory behind Language Models [16–18]. Language Models (LMs) are statistical models that learn the *probability distribution* of sequences of words in a language. Their objective is to predict the likelihood of a word given the context of preceding words. Formally, given a sequence of words $\mathbf{x} = (x^{(1)}, \ldots, x^{(t)})$, LMs computes the probability distribution of the next word $x^{(t+1)}$:

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

where $x^{(t+1)} \in V = w_1, \ldots, w_{|V|}$, and V is a fixed vocabulary. Given a sentence, the goal of a LM is to estimate the probability of this sequence $P(\mathbf{x})$, which is obtained through the chain rule of probability:

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

Modern LMs utilize neural networks to learn complex relationships between words and context. Recurrent Neural Networks (RNN) are ideal for the task, as they model the generative process of a sequence data (*e.g.,* time series, natural language). Unlike other types of NN like feedforward NN, RNNs integrate feedback connections that allow them to retain information from previous time steps (*hidden state*), and then generate a new sample according to a probability distribution given the hidden state. At each time step, a RNN computes an output y^t based on the current input x^t and the hidden state h^{t-1} calculated at the previous step.

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}). \tag{3}$$

LMs are trained on large text corpora, and their parameters are learned to maximize the likelihood of observed sequences (maximum likelihood estimation). The loss function at step t is the cross-entropy between predicted probability distribution $\hat{y}_t$ and the true next word y_t:

$$J^{(t)}(\theta) = CE(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = -\sum_{w \in V} \mathbf{y}_w^{(t)} \log \hat{\mathbf{y}}_w^{(t)} = -\log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}. \tag{4}$$

By averaging the previous on the entire training set, we obtain the following overall loss:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^t(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\mathbf{y}}_{\mathbf{x}_{t=1}}^{(t)}. \tag{5}$$

Embedding representations play a crucial role in LMs. An embedding is a numerical representation of words, phrases, sentences, or even entire documents. These representations are typically high-dimensional vectors that capture the semantic meaning of the text. The importance of embeddings lies in their ability to transform text into a format that machine learning models can understand and process. Therefore, embeddings capture the nuanced meanings of words based on their context, which is essential for tasks like sentiment analysis, translation, and summarization.

3 Threat model

Attack Description. A directory enumeration brute-force attack is a method that checks for and attempts to access directories and files on a web server that are not referenced by the application but are still accessible. This type of attack is performed by generating a large number of requests associated with different URLs sent to the server. The attack is commonly based on a wordlist, a list of words used to construct the URLs starting from the base one the attacker selects.

The primary goal of a directory enumeration attack is to uncover hidden files, directories, backup files, or administrative interfaces that may contain sensitive information or configuration data. If these resources are not adequately secured, they can be exploited to gain unauthorized access, escalate privileges, or launch further attacks. Vulnerabilities typically exploited by such attacks include misconfigured permissions, default installations with sample files, and outdated or unnecessary files left accessible on the server.

Directory enumeration brute-force attacks are often employed during the *reconnaissance phase* of a penetration test. A penetration test, or pentest, is an authorized simulated cyberattack on a computer system performed to evaluate its security. However, this type of attack may also be performed by malicious actors, so it is essential to be aware of this type of attack and to test web application security against it properly.

Automated tools. Several commercial and open-source tools are commonly used to perform directory brute-force attacks. These tools can be specific to this type of attack or be more broad-based to provide other functionalities; additionally, they also often come with default wordlists. Popular tools are:

- *Dirbuster*, a Java-based, multi-threaded tool specifically designed to brute force directories and file names on web or application servers developed by OWASP[1]. It has nine different default wordlists. The tool is freely available at: www.kali.org/tools/dirbuster/.
- *Wfuzz*, an open-source security tool designed to launch brute-force attacks against web applications by fuzzing input parameters and assisting penetration testers in identifying vulnerabilities. It is designed to perform several attacks, such as brute-forcing, fuzzing, and injection attacks. It also comes with several wordlists covering a variety of contexts. The tool is freely available at: wfuzz.readthedocs.io.
- *Burpsuite*, a commercial platform that provides a graphical tool for conducting security testing on online applications. It supports the entire testing process, from initial mapping and analysis of an application's attack surface to the discovery and exploitation of security flaws. Among these, Burpsuite can perform brute-force attacks to enumerate directories, given a target and a wordlist. The tool is available under different licences at: portswigger.net/burp.

Wordlists. Wordlists are essentially a set of directories utilized in a brute-force attack. Therefore, they play a vital choice when using these tools. A proper choice of wordlist can greatly impact the results, potentially uncovering more vulnerabilities.

[1] https://owasp.org/

In the context of directory brute-forcing attacks, there is a range of wordlist categories that fit different needs: from general-purpose wordlists to backup-file wordlists, CMS-specific (Content management system) wordlists, and even more.

Various automated tools are provided by default with various wordlists. However, many other user-created wordlists can be found on the Internet, and users may also create ad-hoc wordlists that satisfy their needs. In the scope of this research, we selected four general-purpose wordlists to assess:

- `big_wfuzz` [**BW**][2]: a Wfuzz default general-purpose wordlist that contains 3024 words.
- `top_10k_github` [**GH**][3]: a user-created wordlist in GitHub containing 10000 words, created selecting the most common words found in ten million URLs.
- `megabeast_wfuzz` [**MW**][4]: another Wfuzz default general-purpose wordlist that contains 45459 words.
- `directory-list_dirbuster` [**DB**][5]: a Dirbuster default wordlist containing 141835 words.

4 Methodology

Overview. Traditional attacks are essentially inefficient, as they are based on brute-forcing mechanisms. In this work, we explore two different approaches that might improve the attack: one based on probabilities and one using a Language Model for path generation. Given the scope of this research, which aims to be general and not to focus on specific technologies or sensitive information, both approaches aim to highlight the feasibility of implementing more efficient attacks and aim to exploit two features not used by the traditional wordlist-based brute-forcing approach:

- ***Prior Knowledge***. Web applications that belong to similar categories might have a similar structure. Given a target website, using knowledge retrieved from similar websites to decide what HTTP requests to send to the target website may positively impact the results.
- ***Adaptive decision-making***. During a directory brute-force attack, having the ability to dynamically decide which URLs to generate and which requests to send might improve the hit rate of successful responses and reduce ineffective requests.

Tree reconstruction. Before we discuss how traditional tools and our proposed approaches work, it is helpful to understand how HTTP responses allow us to reconstruct the filesystem of a web application. Since a filesystem has a hierarchical tree structure, the paths of each web application can be used to reconstruct it. In particular, we used the `AnyTree` class in Python to reconstruct the filesystems of each web application, considering as root the starting URL usually referred to as the target. This strategy allows us to perform depth-level analysis and simulations of offline brute-force attacks so that we do not perform actual attacks on online web applications, thus maintaining an ethical posture that still allows us to obtain meaningful results.

[2]https://github.com/xmendez/wfuzz/blob/master/wordlist/general/big.txt
[3]https://github.com/xajkep/wordlists/blob/master/discovery/top-10k-web-directories_from_10M_urlteam_links.txt
[4]https://github.com/xmendez/wfuzz/blob/master/wordlist/general/megabeast.txt
[5]https://github.com/3ndG4me/KaliLists/blob/master/dirbuster/directory-list-1.0.txt

For example, considering the paths `"/news"`, `"/home"`, `"/register"`, `"/news/2024"`, `"/news/today"` and `"/news/weather"` as the paths extracted from the crawl of a web application, we can visualize the corresponding reconstructed tree in Figure 1.

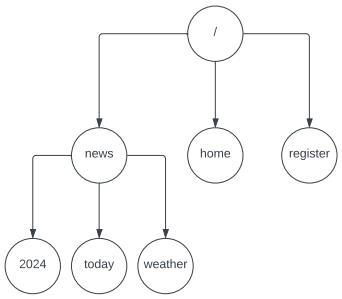


Figure 1: Visualization of a reconstructed tree.

4.1 Standard approach

The standard wordlist-based approach that we will use to compare the results with our proposed approaches is based on two main strategies: `Depth-First` and `Breadth-First`.

Depth-First. In a directory brute-force attack, the Depth-First approach prioritizes the exploration of subdirectories within a discovered directory before moving on to other directories at the same level. The algorithm initiates by sequentially sending HTTP requests using the entries in a wordlist. Upon receiving a positive response, which indicates the construction of a valid URL and, hence, the discovery of a valid directory, the algorithm shifts its focus to brute-forcing the subdirectories of this newly discovered directory. It exhaustively searches within these subdirectories before it resumes brute-forcing other directories at the same depth as the previously validated one. This approach ensures a comprehensive search within each directory before moving on to the next, thereby maximizing the chances of uncovering valuable information nested deep within the directory structure. An algorithmic representation of this approach can be visualized in Algorithm 1, where `constructURL(URL, word)` is a function to generate a valid URL appending a word to the path of the URL and `isValid(response)` is a function that 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
```

Breadth-First. In contrast to the previous approach, the Breadth-First approach prioritizes the exploration of directories at the same level before delving into their subdirectories. The algorithm begins by sending HTTP requests sequentially using the wordlist entries. When it receives a positive response, indicating the formation of a valid URL and, hence, the discovery of a valid directory, it continues to brute-force the remaining directories at the same depth. Only after it has exhausted all directories at the current level does it proceed to brute-force the subdirectories of the discovered directories. This method ensures a thorough search across each level of directories before descending deeper into the directory structure, thereby maximizing the chances of uncovering valuable information distributed across the directories. The majority of commercial tools implement this approach. We also present a pseudocode implementation in Algorithm 2, using a queue to store and retrieve the URLs used during the process.

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
```

4.2 Probability-based approach

Approach. Prior knowledge might be essential to improve the attack performance. The intuition is straightforward: if the majority of websites contain paths like `/login` and `/register`, it is likely that the tested website contains such directories as well. An algorithm that, therefore, prioritizes directories based on prior knowledge can be effective.

The first strategy we present optimizes the depth and breadth-first strategies described in Section 4.1, where the wordlist is ordered according to prior knowledge (e.g., gathered from web applications similar to the victim). This adds dynamic decisions on how to go about generating the following HTTP request to maximize the number of positive requests while minimizing the number of unlikely and incorrect requests. The prior knowledge, or training dataset, contains crawls of paths of various web applications, possibly of the same category as the target where the attack will be performed. The prior knowledge can be then infused into the algorithms in two possible manners:

(1) Constructing a **Weighted Training Tree**. Using the same reasoning with which we described how it is possible to reconstruct a filesystem of a web application from the crawl of its paths, we will proceed to construct a single filesystem tree that unites all the paths that contain our training dataset indistinctly from the web application. Furthermore, this tree will be weighted: for each new node in the tree (corresponding to a directory), we would maintain a counter indicating how many times that particular node is repeated. An example of this is reported in Figure 2.

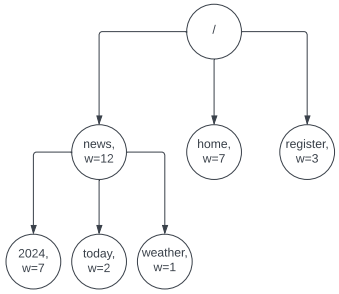


Figure 2: Visualization of a Weighted Training Tree, obtained merging paths from a Training Dataset.

(2) Constructing a **Weighted Wordlist Tree**. A weighted tree using only the words from the wordlist. Starting from a general wordlist, we construct a weighted tree similar to the Weighted Training Tree. However, this tree only includes words from a pre-designed wordlist (e.g., big_wfuzz). The weight of each node (directory) in this tree is determined based on the training set. For example, if we consider the wordlist ["*news*", "*home*", "*2024*", "*today*", "*about*"] and the Weighted training tree shown in Figure 2, the corresponding Wordlist Weighted Tree can be visualized in Figure 3. Note that, in this case, folders such as `register` and `weather` are not included in the tree, since they are not contained in the original wordlist.

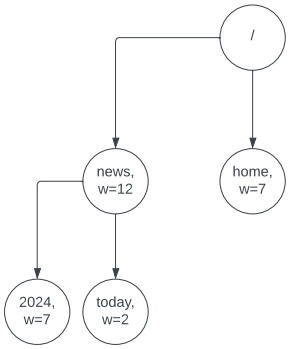


Figure 3: Visualization of a Wordlist Weighted Tree, based on a wordlist and a Weighted training tree.

In this way, we can make the best use of parent-child relational information between directories and subdirectories and give weight to words in the wordlist critical for the adaptive selection of requests

to be made. Furthermore, a pruning process can be applied to those branches whose w is lower than a threshold. The pruning of unlikely words helps, consequently, to minimize the number of less probable requests. To give an example of how pruning works, considering Figure 2 and Figure 3, we can see that the word "about", which was initially in the wordlist from which the tree is created, is not part of the Wordlist weighted tree (so it has a weight of 0 as a possible subdirectory for every directory), and "news" has a weight of 12 as a subdirectory of the base root, but is pruned from being a child-node of "home".

Algorithm. The probabilistic approach employs a max heap (a data structure that keeps the maximum element of a given property on top of it) with tuples of base URLs, a word, and the weight assigned to it. The max heap keeps the tuples ordered by probability, so we can always pop the highest one to construct and send a request. The probability of a word being a valid subdirectory of a directory is computed dynamically by dividing the weight of the possible subdirectory by the sum of the weights of every possible subdirectory to that directory.

In the beginning, given a target base URL, the algorithm will push in the max heap all the possible subdirectories of the root directory "/" retrieved from the weighted tree with the corresponding probabilities. Whenever we receive a successful response, the algorithm pushes all the possible subdirectories to the response URL with the probabilities into the heap. This mechanism allows us to implement an adaptive decision-making strategy to consistently send the most likely HTTP request away as new subdirectories are discovered.

An algorithmic exemplification of this approach is presented in the algorithm 3, where `getWordswithWeights()` returns the word-weight pairs taken from the specified URL in the weighted tree, and `getProbability()` calculates the probability of a word as described before.

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
```

Because of aggressive pruning on the wordlist tree or due to a training dataset that does not contain as much data, the possible requests provided by the weighted tree will be exhausted quickly, even before reaching any set request budget. In light of this, when all potential requests derived from the existing knowledge within the weighted tree have been explored, it is advisable to employ a conventional breadth-first strategy. This approach takes into account the previously successful responses, thereby mitigating the need to reissue redundant HTTP requests.

4.3 Language-Model based approach

Approach. Following the intuition of the probabilistic approach described in Section 4.2, we design a neural network mechanism leveraging Language Models (see Section 2) to generate probable subdirectories to a given path. Given a URL, we can consider its path a sequence of words separated by "/." This sequence of words can be fed as input to the neural network mechanism, which will output the words that most likely follow the input sequence. Those words can be used to construct new URLs and send new HTTP requests.

With this method, we aim to leverage the power of customized embeddings (i.e., embeddings trained on the corpus), and overcome the limitations of the probabilistic approach. In particular, the probabilistic approach calculates relationships among directories that appear in the prior knowledge. On the other hand, with the embedding, the model learns the context, and therefore directories appearing in a similar context will be used to generalize the attack. For instance, suppose that in our prior knowledge, we have URLs such as `"/account/setting/info"`, `"/account/setting/password"`, `"/account/setting/logout"`, `"/profile/setting/ password"` and `"/profile/setting/info"`: the directories `account` and `profile` are utilized in a similar context, and therefore their embedding will be close. At inference time, a LM might infer the URL `"/profile/setting/logout"` even though this information was not available in our prior knowledge. In this example, a probability approach would have assigned 0 to this association.

Model architecture. Leveraging language models in our architecture involves using a crucial component of them: the vocabulary. This component maps words in our sequences (i.e., directories) to unique indexes. The vocabulary allows the translation of words into integer indices that the neural network architecture can process. To reduce the dimension of the vocabulary, only words more frequent than a certain threshold are considered. Additionally, vocabulary helps the architecture understand the structure of the sentences while handling unknown words and variable-size sequences with special tokens. These tokens are: `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).

Our designed neural network architecture is primarily based on the Long-Short-Term Memory (LSTM) network [9], a type of Recurrent Neural Network. Our proposed LM architecture consists of several key components:

(1) **Embedding Layer**. The first layer of the model is an embedding layer, which transforms the input words into dense vectors of fixed size, defined as embedded size. Embedding representations are learned at training time.
(2) **LSTM Layer**. It follows a LSTM layer, crucial for the learning of patterns in sequential data. This layer takes the embeddings of the input words and returns its own hidden states and cell states.

(3) **Dropout Layer**. To prevent overfitting, dropout layers are used after the Embedding and LSTM layers. *Dropout* is a regularization technique that randomly sets a fraction of input units to 0 with a specific frequency of rate at each step during the training.
(4) **Fully Connected Layer**: The LSTM outputs (hidden states) are then passed through a fully connected (linear) layer to transform them into the desired output shape, which is the size of the vocabulary.
(5) **Softmax Function**: A softmax function is applied to transform the output of the fully connected layer to probabilities assigned to each vocabulary word.

Figure 4 shows an overview of the architecture. Here, we can see how the URL path is split into tokens and mapped into integers using vocabulary. Then, after the sequence of integers is given as input to the model, we observe how the softmax function takes the output of the fully connected layer and assigns the probability that it is the next in the sequence to each number. At this point, the most likely word is chosen to form a new path, but the choice can also be made on any other word.

Training and Validation. Our architecture's training process involves feeding it with paths and having it predict the following directory in the path. The model's predictions are compared to the actual following directories in the path, and a loss function is used to quantify the difference between the predictions and the truth. This loss is minimized using an optimization algorithm, which adjusts the model's parameters to make its predictions more accurate.

During the training phase, the model performance is periodically evaluated on the validation set to prevent overfitting on training data. This strategy allows us to monitor the model's generalization ability to unseen data. We utilize an *early stopping* mechanism to stop the training when the model's performance on the validation set starts to deteriorate (a phenomenon known as overfitting) or does not improve for p epochs.

Algorithm. The algorithm incorporating the proposed neural network architecture (Algorithm 4) uses a strategy similar to Algorithm 3, where a max heap is used to construct the most probable URL. Instead of using the wordlist weighted tree, the language model in the function `predict()` is used to return a number of word-probability pairs with a higher probability specified in the `topPredicts` hyper-parameter.

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, newProb in newTuples do
14:                 maxHeap.push((url, word, newProb))
15:             end for
16:         end if
17:     end while
18: end procedure
```

5 Dataset

In this section, we present the datasets collected for our experiments. In particular, Section 5.1 describes the data collection process. It follows Section 5.2, presenting an in-depth analysis of our data.

5.1 Description

Source of data. The data for this research is obtained from CommonCrawl [6], a non-profit organization that crawls the web and freely provides its archives and datasets. CommonCrawl was selected due to its comprehensive repository of web crawls that are updated regularly; we utilized `CC-MAIN-2023-40` crawl version. This approach not only streamlines the data collection process but also aligns with ethical considerations, avoiding manual spidering and crawling of websites that may be categorized as brute-force attacks and not overloading web servers with severe amounts of requests but instead using historical data. Given the scope of the search and the multitude of data in the CommonCrawl corpus, we collected the URLs from the HTTP responses. We then extracted the `domain`, `path`, and `response code` necessary to identify invalid responses.

Datasets. Considering the increasing number of cyber attacks and the wide variety of possible targets, we decided to consider four different datasets representing some most common categories of organizations at risk of cyber attacks [7, 19]. In addition, given the scope of the search, we maintained only websites written in English. We now list the four distinct datasets we collected:

- *Universities dataset* [**UNI**]: consisting of the HTTP responses from 100 English-based web applications of the top universities listed in the QS 2023 World University Rankings [7]. Furthermore, we did not consider universities without an English version of their web application.
- *Hospitals dataset* [**HOS**]: consisting of the HTTP responses belonging to the web applications of the first USA 100 hospitals listed in "Ranking Web of World Hospitals" [8].
- *Companies dataset* [**COM**]: consisting of the HTTP responses from 100 corporate web applications of companies in the S&P 500 [9], choosing in order of highest capitalization (January 2024) and avoiding companies that had e-commerce as their main web application.
- *Government dataset* [**GOV**]: consisting of the HTTP responses from 336 different USA government web applications [10].

In our experiments, we will report the attack performance when considering the datasets separately and together.

Preprocessing. After extracting the domain, path, and status code from each HTTP response for each dataset, we performed additional preprocessing steps on the data to reconstruct the hierarchical structure of the file system of the Web applications, guaranteeing

[6] https://commoncrawl.org/

[7] https://www.topuniversities.com/world-university-rankings/2023
[8] https://hospitals.webometrics.info/en/americas/usa
[9] https://www.slickcharts.com/sp500
[10] https://www.usa.gov/agency-index

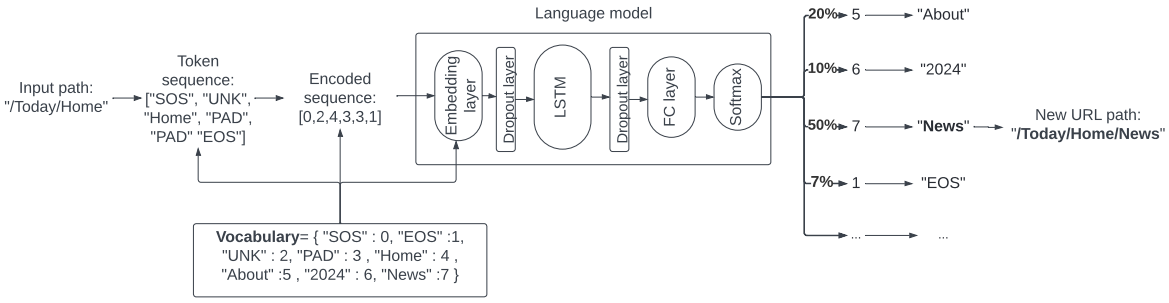


Figure 4: Prediction of the next directory from our LM-based architecture.

their integrity and enabling more consistent analysis in the datasets. Firstly, we applied initial filtering to the HTTP responses, preserving only the responses with a status code 200, representing most of the crawled responses. The HTTP status code 200 indicates that the client request has succeeded. Secondly, eventual queries or files that were part of the path were removed. This choice was mainly made to maintain a manageable scope of analysis. URLs can often contain queries or files with different extensions that introduce a significant degree of variability and are highly dependent on the technology with which the web application was developed, thus going against our intended general approach. Additionally, it is useful to define the *depth* of a path: considering a path as a sequence of words (where each word corresponds to the name of a directory), we define the *depth* as the number of words that the path consists of. The order matters since each directory in that path will be at a given depth (e.g.: `"/news/2023"` will have depth = 2, where `"news"` is at depth 1, and `"2023"` is at depth 2).

5.2 Datasets Analyses

Overview. Following the data preprocessing, we analyzed the characteristics of each dataset and the similarities and differences between them, which allowed us to get a general overview of the datasets and highlight what direction the test results might take.

We conduct the following analyses:

- *Dataset description*, where we describe datasets' properties (e.g., quantities, distributions) and the nature of their URLs.
- *Wordlist Coverage Ratio Analysis*, where we describe how standard wordlists can cover the retrieved URLs.
- *Stemming Analysis*, where we attempt to understand the impact of small name variations in the directories (e.g., `books` and `book`) and wordlists coverage.
- *Dataset Similarity Analysis*, where we describe the degree of similarity between directories in the four collected datasets (e.g., how universities and hospital URLs differ).

We report below only the dataset description, while in-depth analyses can be found in Appendix A.

5.2.1 Dataset description. We now describe the four distinct datasets we retrieved. For each dataset, we analyzed the following information:

- Number of domains (`# Domains`).
- Number of paths in each dataset (`# Paths`). For instance, suppose a dataset contains two web apps, each with one domain (e.g., `"domain1/login"` and `"domain2/login"`), the number of paths is two, i.e., `["/login", "/login"]`).
- The average number (and standard deviation) of paths of the web apps contained in a given dataset (`# paths AVG` and # paths STD). For instance, given two web apps containing each 1 URLs, the average is equal to 1, and the standard deviation to 0.
- The number of unique paths in the dataset (`# U-Paths`). For instance, suppose the dataset contains the following samples `" domain1/account/info "` and `" domain2/account/ info"`, the unique paths are defined as the unique path `["/account/info"]`.
- The number of directories (`# Dir`) contained in the dataset. For instance, given `"domain1/account/info"` and `"domain2 /account/settings"`, the dataset contains four directories `["account", "info", "account", "settings"]`.
- The number of unique directories (`# U-Dir`) contained in the dataset. For instance, given `"domain1/account/info"` and `"domain2/account/settings"`, the dataset contains three unique directories `["account", "info", "settings"]`.
- We analyze the depth of URLs in terms of average and standard deviation (`Depth AVG` and `Depth STD`). For instance, the URL `"domain1/account/info"` has a depth equal to two. Hypothesizing that a directory with greater depth is likely to be more specialized and, consequently, less common, analyzing depth distribution might provide insight into a web application's granularity and structure.
- The average depth and standard deviation of URLs similarities in a given dataset (`Sim AVG` and *Sim STD*). In more detail, this metric computes the similarity for each pair of websites in each dataset. By defining a website as a set of its own directories (computed by the retrieved URLs), the similarity between two websites can be computed using the Jaccard Similarity, defined in Equation 6. This metric is defined between 0 and 1, where 0 means that two sets are distinct, while one means identical.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{6}$$

Table 1 shows the statistics for each dataset. We highlight that not all scraped domains contained the HTTP status code 200, and therefore they are not present. This justify the discrepancy between

the number of scraped domains and the actual domain (e.g., the dataset contains URLs from 88 universities and not 100).

	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
# U-Paths	201768	205587	143067	502693
# Dir	203613	209945	143620	512595
# U-Dir	171215	173394	106097	462812
Depth AVG	4.11	3.31	4.43	3.40
Depth STD	1.69	1.72	2.23	1.66
Sim AVG	0.022	0.019	0.016	0.016
Sim STD	0.031	0.017	0.023	0.024

Table 1: Statistics for the datasets: universities [UNI], hospitals [HOS], companies [COM], government [GOV].

Discussions. It is interesting to observe that websites have different structures in terms of number of pages. For instance, universities and hospitals tend to have a bigger number of pages (`# Path AVG`) compared to companies and governments. From a depth perspective, university and company websites tend to have deeper web app structures compared to hospitals and government. Last, the similarity analysis clearly shows that websites in the same dataset tend to have different types of directories, and the average Jaccard similarity tends to zero. By considering these statistics together, we can clearly remark why directory enumeration is not trivial, and it might require an enormous amount of requests for a small number of hits (i.e., discovered directories).

6 Evaluation

6.1 Experimental Settings

Testbed. To set up our brute-force directory attack simulations, we partitioned the datasets into distinct sets for training, validation, and testing. We train our proposed approaches (i.e., probabilistic and LM-based) in a training set that merges the four presented in Section 5.1. Merging the four data sources for the training is essential to have a sufficient number of samples to train a LM with. We, therefore, merge the four datasets and divide them into training, validation, and testing sets with a 70-10-20 split ratio, as mentioned in Section 4.3. Note that the split is not random in terms of URLs, but from a domain perspective. In this way, all URLs of a given website will appear only in training, validation, or testing set. With this approach, we avoid any *data snooping* [3].

Regarding the testing environment, we opted to simulate brute-force attacks offline, utilizing the virtual filesystems reconstructed from the test applications. This approach permits executing multiple attack simulations using different strategies without actualizing real-time brute-force attacks on live web applications. Furthermore, this approach allows us to work with a high number of simulated requests without introducing latency due to the HTTP requests. Additionally, aligning with our objective to maximize successful responses while minimizing request volume, we established a maximum budget of 100,000 requests spendable by each simulated attack before termination.

LM validation. We utilize PyTorch [15] to design our architecture. LM uses the training set to learn meaningful association, while the validation set is utilized to select the best LM hyperparameters. For the model selection, we use a grid-search validation over the following hyper-parameters.

- *Data representation.* We identify hyper-parameters that controls the training input: the maximum length of the paths utilized in the training phase 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 values [5, 10], while for the latter, defined as `min_freq`, we chose [3, 5].
- *LM architecture.* `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].

An early stopping mechanism is set with patience equal to 10 epochs. The learning phase uses Adam [11] as optimizer, and CrossEntropy as loss function. Finally, the best model is chosen based on the lower loss at the validation set. In addition, we tested an additional parameter, namely the number of predictions to be considered whenever a positive response is received during the attack and new predictions are made. The parameter, defined as `topPredicts` in Algorithm 4, is tested with the values [100, 250, 500, 750, 1000, 2000, 5000, 10000].

Evaluation Metric. We utilize the following evaluation metrics:

- *Average Successful Response Rate*, consisting of averaging the total amount of successfully discovered directories for each tested website.
- *Bins efficiency*, consisting of averaging the total amount of successfully discovered directories for each tested website in a given range of requests. We evaluate the following bins: 0-100, 101-1000, 1001-10000, 10001-50000, and 50001-100000.

We did not consider execution times as they depend on various factors (such as the number of threads used in the attack and time between request and response variable for each application) and are not quantifiable with offline simulations.

6.2 Results

Overview. Table 2 shows the overall results obtained in our experiments, at the varying of the dataset, wordlists, and inference techniques. LM-based attack outperforms all the other baselines in all datasets, demonstrating the superiority of language models compared to naive techniques like brute-force or probabilistic approaches. Interestingly, LM's performances are not equally balanced on all datasets: for instance, LM struggles with university and company websites, while being robust with hospital and government ones. The probabilistic-based approach further shows a good improvement over brute-force attacks since, with a limited budget, the latter might not be able to fully assess all the possibilities. On the other hand, probabilistic approaches, by optimizing the priorities of the requests, are more efficient.

Bins efficiency. The efficiency analysis provides another interesting perspective on how different approaches perform. We analyze

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

the efficiency shown in Figure 5, calculated by considering simulations on the general test dataset (**[ALL]**) and on the wordlist `big_wfuzz`.

Although the mean results obtained using the `breadth`, `depth`, and `probabilistic` approaches are the same, the efficiency varies considerably. The probabilistic approach performs very well in initial requests and then declines as requests increase. Although the language model approach registers a 400% increase in average successful responses, it performs worse in initial requests than the probabilistic approach but is more efficient in the long run. This behaviour between the two is also observable in the other cases. Therefore, adopting a probabilistic approach might be ideal when the budget is more limited.

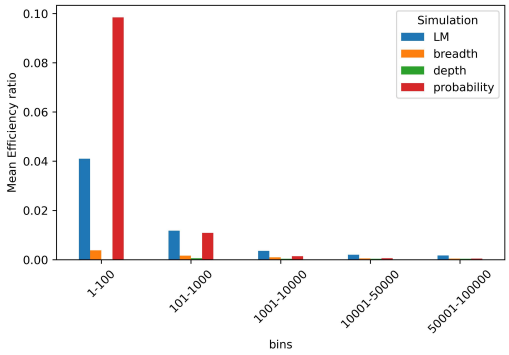


Figure 5: Mean Efficiency Ratio of the four approaches on different bins, considering wordlist = `big_wfuzz` and the general dataset.

6.3 Discussion

The impact of `topPredicts`. Algorithm 4 relies on many hyperparameters, such as `topPredicts`. It controls the number of most likely predictions to be considered in each new folder found. We analyze how the best LM found changes its performance at the varying of this setting. In particular, we explore the following: 100, 250, 500, 750, 1000, 2000, 5000, and 10000. As shown in Figure 6, as the number of predictions considered increases, the average number of successful responses received in the attack simulations decreases. In addition, although with a smaller `topPredicts` the initial average number of successful responses received is better, the simulations end earlier as they have no more new predictions with which to send new requests. It is, therefore, essential to consider a value for this parameter that maximizes the results and utilizes the entire predetermined request budget. Therefore, it might be ideal to set `topPredicts` to a small number if our budget is limited, as the model reaches the most successful responses in a short time. On the opposite, larger numbers like 500, 750, and 1000 are ideal when the budget allows an exhaustive search.

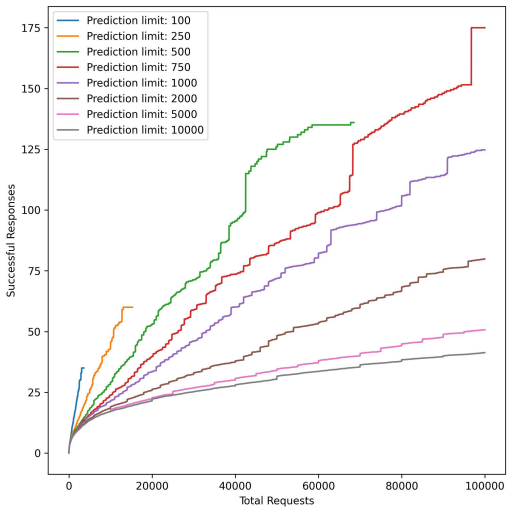


Figure 6: Evolution of average successful responses for different topPredicts values

Results analysis. The two proposed approaches show substantial improvements in both metrics examined. Of the two standard approaches, the `breadth-first` strategy (also implemented by commercial tools) emerges as the best.

The probabilistic approach improves the performance of the breadth-first approach in 65% of the cases considering the four default wordlists, while in the remaining, it obtains equal or slightly lower results. Depth-first approaches are outperformed in 100% of the cases. In particular, if we consider the breadth-first approach, the probability-based approach using the train-set wordlist has the following average improvements: University +141%, hospital +281%, companies +85%, government +78%, and **ALL** +159%. The strength of this approach is the efficiency of successful responses received using few requests, which outperforms all other approaches considerably. The Language-based approach outperforms the standard approach in 100% of the simulations. The LM-based model approach has the following average improvements over the breadth-first baselines: University +582%, hospital +1004%, companies +499%, government +639%, and **ALL** +969%.

Embeddings similarity. The ability of embeddings to extract context from web application paths and generalize is essential to predict valid directories and URLs. The results, especially in simulations on general test sets, highlight how the Language model approach successfully uses the context extrapolated from embeddings to achieve significantly better results than the other approaches.

We can observe this by reporting two examples: given two directories, we use the Cosine similarity to measure the top 10 words most similar directories, which should belong 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 cases, we can see that the words determined similarly by the embeddings represent the same word but slightly different, such as '`about`' with '`about-us`' or '`article`' with '`articles`'. In addition, we find other words that relate to the context created by the word under consideration, such as '`authors`' or '`stories`' for '`article`. This outcome confirms the superiority of LM in generalizing the observed pattern at training time.

Examples of LM Patterns. The high average number of successful responses obtained from the LM-based approach testifies to the model's ability to predict valid directories that follow recurring patterns. For example, let us examine the Language model's predictions on two different URLs:

(1) URL: */campus-life-events/calendar*. Among the top 10 directories predicted with this URL, we have ['05', '06', '08', '11', 'may', jun'], which refer to days or months of a calendar.
(2) URL: */media*. Among the top directories predicted with this URL, we have ['press-releases', 'news'] that are found in multiple paths in the training dataset and that refers to a similar context.

7 Related Work

The emergence of offensive AI in cybersecurity presents a new frontier where artificial intelligence (AI) is leveraged to create sophisticated and automated attacks and enhance the penetration testing process [10, 13]. These attacks represent a new landscape that poses significant challenges and opportunities in cybersecurity, especially with the raising of LLMs and generative AI.

The use of generative AI to enhance directory brute-forcing attacks has yet to be explored. The closest attempt is presented by He et al. [8], where the authors proposed an attack to medical systems by adopting semantic clustering of sentences. No much information are reported in terms of data, methodology, and results. Similarly, Antonelly et al. [2] presented an innovative approach using the Universal Sentence Encoder (USE) for semantic analysis. The K-means algorithm and the elbow method were used for clustering to optimize directory brute-forcing (dirbusting), with an improvement in the results of up to 50% on only eight web applications tested.

Several other studies have analyzed the threat that offensive AI poses to organizations in other types of attacks. Bontrager et al. [5] demonstrated the potential of AI-generated fingerprint deepfakes to compromise biometric systems through dictionary attacks, highlighting the vulnerability of such systems to sophisticated AI techniques. Al-Hababi et al. [1] investigated man-in-the-middle attacks leveraging machine learning to identify services in encrypted network flows. Li et al. [12] presented a generative adversarial network designed to evade PDF malware classifiers, illustrating the ease with which AI can bypass traditional cybersecurity defences. Nam et al. [14] developed a recurrent GANs-based password cracker aimed at enhancing IoT password security. While intended for defensive purposes, the study also signifies how AI can be repurposed for

8 Conclusions

Current directory brute-forcing attacks are notoriously inefficient since they rely on brute-forcing strategies, resulting in an enormous amount of queries for a few successful discoveries. In this work, we investigated whether the utilization of prior knowledge might result in more efficient attacks. We propose two distinct methods that rely on prior knowledge: a probabilistic model and a Language Model-based attack. We then experimented with our proposed methodology in a dataset containing more than 1 million URLs, spanning across distinct web app domains such as universities, hospitals, companies, and government. Our results show the superiority of the proposed method, with the LM-based approach outperforming brute-force-based approaches in all scenarios (an average performance increase of 969%). Furthermore, the simple probabilistic approach results effective when the budget of requests is limited (below 100, for stealthier attacks). The research presented in this paper lays the groundwork for several promising directions for future investigation. The use of Artificial Intelligence to create sophisticated attacks is a topic that is constantly evolving and growing in cybersecurity, especially with the fast development of Language models.

Future work could explore improvements of our LM-based architecture, such as attention mechanisms [20], or even Large Language Models [6]. These models' enhanced understanding of context and semantics could significantly refine the process of predicting web application structures. Additionally, a path that may be explored is the development of a language model trained explicitly on paths and files commonly associated with vulnerabilities. By focusing on these critical areas, the model could help preemptively identify potential security risks, thereby contributing to more proactive cybersecurity measures and showing the feasibility of such attacks.

These areas of future work offer the potential to significantly impact the development of more secure web environments and pose new security challenges to language model usage.

9 Acknowledgment

This work was supported by the European Commission under the Horizon Europe Programme, as part of the project LAZARUS (https://lazarus-he.eu/) (Grant Agreement no. 101070303). The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

References

[1] Abdulrahman Al-Hababi and Sezer C Tokgoz. 2020. 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, 1–5.

[2] Diego Antonelli, Roberta Cascella, Gaetano Perrone, Simon Pietro Romano, and Antonio Schiano. 2021. Leveraging AI to optimize website structure discovery during Penetration Testing. arXiv:2101.07223 [cs.CR]

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

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

[5] Philip Bontrager, Aditi Roy, Julian Togelius, Nasir Memon, and Arun Ross. 2018. Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution. In *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*. IEEE, 1–9.

[6] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* (2023).

[7] Abrael Delgado. 2023. Who is the Prime Target for Cyber Attacks? — compuquip.com. https://www.compuquip.com/blog/prime-target-for-cyber-attacks-and-to-look-out-for. [Accessed 10-05-2024].

[8] Ying He, Cunjin Luo, Jiyuan Zheng, Kuanquan Wang, and Henggui Zhang. 2022. AI Based Directory Discovery Attack and Prevention of the Medical Systems. In *2022 Computing in Cardiology (CinC)*, Vol. 498. IEEE, 1–4.

[9] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[10] Nektaria Kaloudi and Jingyue Li. 2020. The ai-based cyber threat landscape: A survey. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–34.

[11] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[12] Yuanzhang Li, Yaxiao Wang, Ye Wang, Lishan Ke, and Yu-an Tan. 2020. A feature-vector generative adversarial network for evading PDF malware classifiers. *Information Sciences* 523 (2020), 38–48.

[13] Yisroel Mirsky, Ambra Demontis, Jaidip Kotak, Ram Shankar, Deng Gelei, Liu Yang, Xiangyu Zhang, Maura Pintor, Wenke Lee, Yuval Elovici, et al. 2023. The threat of offensive ai to organizations. *Computers & Security* 124 (2023), 103006.

[14] Sungyup Nam, Seungho Jeon, Hongkyo Kim, and Jongsub Moon. 2020. Recurrent gans password cracker for iot password security enhancement. *Sensors* 20, 11 (2020), 3106.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library.* Curran Associates Inc.

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

[17] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. 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)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 2463–2473. https://doi.org/10.18653/v1/D19-1250

[18] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. 2015. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*. 298–307.

[19] The Constella Team. 2022. Top Common Targets for Hackers | How Do Hackers Choose Targets? — constella.ai. https://constella.ai/top-common-targets-for-hackers/. [Accessed 10-05-2024].

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

A Dataset Analyses

A.1 Wordlists Coverage Ratio Analysis

By analyzing the coverage of the different wordlists on the web applications in each dataset, shown in Figure 7, we can see that a low percentage of words are found even at low depths where we would expect them to be more common and thus present in the wordlist.

The `coverage ratio` across the different datasets shows considerable variance but overall low values, highlighting how directory brute-force attacks using these wordlists could potentially miss multiple valid requests.

Although the `coverage ratio` shows an upward trend as depth increases, it is essential to emphasize that although the higher-depth words might be more specific, their number is significantly reduced (as highlighted by the depth distribution analyzed earlier). In addition, the most critical point concerns the poor coverage of the initial words, which form the basis of most pathways: higher-depth directories will not be explored if antecedent ones are not explored.

A.2 Stemming Analysis

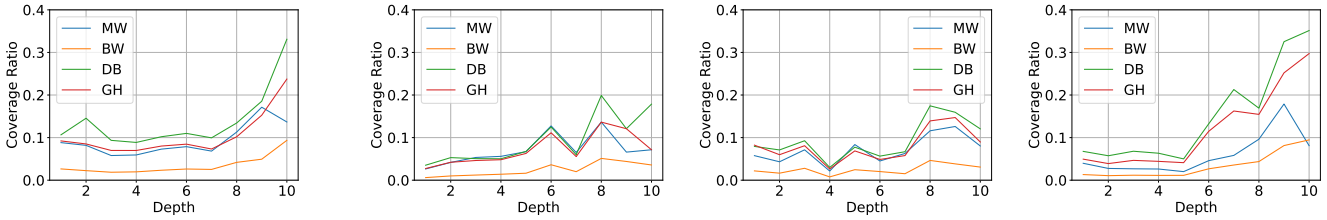
Stemming is a linguistic process that simplifies words to their base or root form, known as the stem, often by removing common prefixes or suffixes. For example, stemming removes plural (dogs → dog), -ing form (running → run), etc. In our study, we employed the Porter-Stemmer [4] algorithm to analyze the effect of stemming on the total number of unique words within our datasets. In our analysis, we found a few instances of how a root form represents minimal variations of the same word, highlighting different conventions or singular and plural forms. A pair of examples are:

(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%.

However, these represent only a minority of cases, as most root forms correspond to only one word, and the percentage reduction in the datasets remains marginal, as shown in Table 3. These statistics highlight how the words that make up our directory list differ (although some words have various declinations) and how that should be taken into account when designing new, improved approaches.

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

Table 3: Summary statistics after the STEMMING for the four datasets: universities [UNI], hospitals [HOS], companies [COM], and government [GOV].

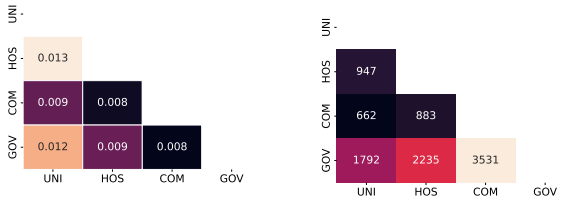


(a) University. (b) Hospital. (c) Company. (d) Government.

Figure 7: Coverage analysis at the varying of the four datasets and four wordlists. Datasets: universities [UNI], hospitals [HOS], companies [COM], and government [GOV]. Wordlists: `big_wfuzz` [BW], `top_10k_github` [GH], `megabeast_wfuzz` [MW], and `directory-list_dirbuster` [DB].

A.3 Similarity Analysis

Last, we measure the similarity between any pair of the collected dataset. We utilize two metrics: the Jaccard similarity of each dataset wordlist, and the number of paths in common (relative number) between the datasets. Figure 8 shows the results. The first clear outcome is highlighted by the low Jaccard similarities: each dataset contains different directories. In other words, how websites of universities have almost completely different structures compared to hospital ones. This reasoning can be applied to any pair of datasets we utilized. A second interesting outcome is given by the relative count of common directories among different datasets. For instance, government and company websites contain many common paths. Considering the number of unique directories shown in Table 1, it is clear that it is non trivial to design an effective directory enumeration brute-force attack.



(a) Jaccard similarities between datasets wordlist. (b) Number of common paths.

Figure 8: Similarity analysis for the four dataset: universities [UNI], hospitals [HOS], companies [COM], and government [GOV].