Deep Learning x Malicious URL Detection

** I've lost the Bi-LSTM/char tokenizer model checkpoint that achieved an accuracy score of 90%.

Introduction

Malicious URLs pose serious problems for people, businesses, institutions, and organizations connected to the web. The urls of mass phishing campaigns sent from VPN-cloaked servers, impersonate corporate identities to steal personally identifiable information or access critical data of the target. They're used to hijack and deface company or government websites, rendering critical data inaccessible and inoperable. Malware could slip into software download from an untrusted website, providing another point of access for hackers to break into a network. These urls if ever clicked or visited even for a second could do grave damage and irreparable harm to anyone online.

The current standard methods for detecting malicious urls (i.e malware, phishing, and defacement) rely on blacklists — databases of confirmed bad urls — and heuristics — manually formulated rules to identify bad urls based on an analysis of confirmed bad urls. But, these techniques for detecting malicious urls. Detecting malicious urls based on a pre-defined set of characteristics and a database of collected bad urls proves unreliably effective against attackers because they constantly adjust the patterns of their urls and bypass heuristic rules.

Cybersecurity and AI researchers have been exploring new and more robust methods in the area of machine learning and deep learning that predict malicious urls more reliably and rapidly. Deep learning models still rely on large and frequently updated datasets, but the trade off for enhanced malicious url detection capabilities could better support corporations, organizations, and governments working around the clock to respond to increasingly challenging and evolving cybersecurity threats.

Method & Materials

Datasets

All data used in this project was sourced from public datadumps, API feeds, and Python libraries, then consolidated and formatted into Comma Separated Value (CSV) files for uniform processing. The final compiled dataset comprises both benign and malicious URLs, detailed as follows:

Deep Learning Model Architecture

The detection system is modeled on Recurrent Neural Network (RNN) architecture. It uses either a Bidirectional Long Short-Term Memory (Bi-LSTM) or Gated Recurrent Unit (GRU) layer.

Bi-LSTM and GRU architectures lend themselves well to sequential data analysis, as a URL is a sequence of characters.

Simplicity and Efficiency: Selecting one of these layers represents the simplest viable option for sequential text classification, offering an excellent balance between computational efficiency and detection capability.

Mitigation of Overfitting: A conscious decision was made to avoid more complex models (e.g., deep stacked RNNs or complex attention mechanisms) to prevent potential overfitting to the dataset's specific URL patterns. This ensures the model generalizes better to new, unseen malicious URLs.

Tokenizer Strategy

A standard Character-Level Tokenization scheme was employed for processing the URLs.

This approach was prioritized over using complex word-level tokenizers such as a Bidirectional Encoder Representations from Transformers (BERT) Tokenizer for the following reasons:

Feature Extraction: URLs are highly structure-dependent (e.g., domain name, path, query parameters). Character-level encoding preserves the structural integrity and allows the model to learn malicious patterns based on character sequences, special symbols, and obfuscation techniques, which is far more relevant to URL analysis.

URL Structure vs. Natural Language: BERT tokenizers are trained on the syntactic and semantic relationships inherent in large corpora of natural English text. URLs, conversely, are typically not composed of conventional words and do not adhere to grammatical rules.

Experiment & Results

I ran tests pairing the char or word level tokenizer with either bi-lstm or bi-gru for a total of 4 model configurations for testing: 1) bi-lstm + char, 2) bi-lstm + word, 3) bi-gru + char, 4) bi-gru + word. For the first round of testing or the first four runs, I held the epoch count, learning rate, batch size, and drop out probability constant. The bi-lstm x char tokenizer out performed all other model x tokenizer configurations, but I've not been able to replicate that performance in later testing rounds. The next round of testing didn't show any significant improvements in performance for any model x tokenizer configuration.

Gemini recommended that I use 256 for the maximum length of the url (i.e max_len) because hackers typically hide attack payloads at the end of long strings to evade detections by humans or simple scanners. It's also a good design choice for the character-level tokenization granularity because 90-95% of urls reach as long as 200 characters. max_len = 256 will ensure that I cover 98% of the data without truncating too much and losing important url features.

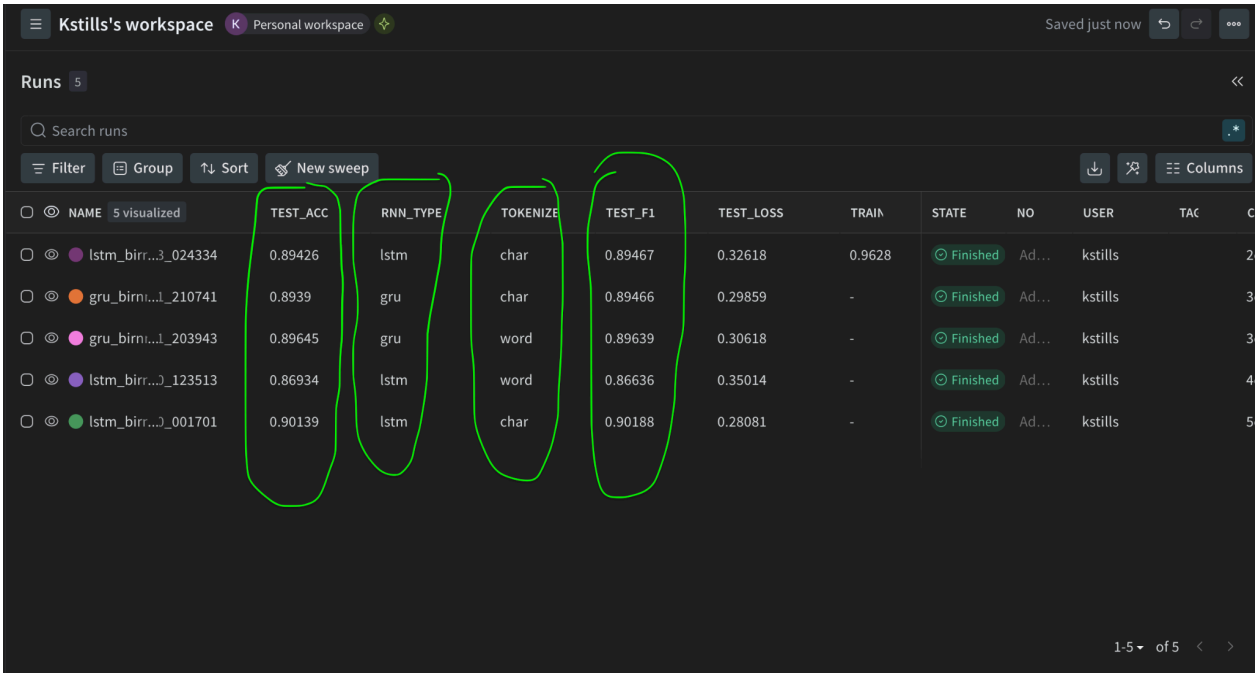

Figure 1: test accuracy and loss, tokenizer type, rnn type, and test f1-score. **there's an additional test simply for supplying a checkpoint used in inference.py

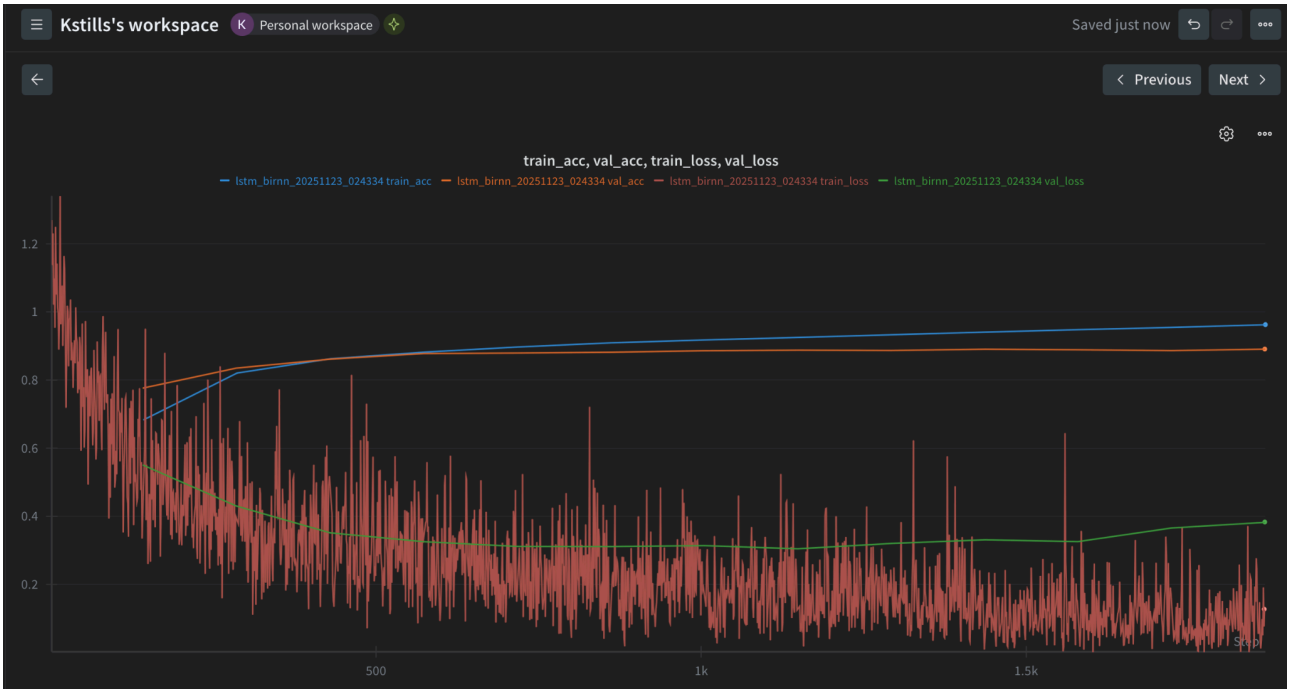

Figure 2: the validation and training loss and accuracy for 4 experimental runs (excludes the 5th run)

lstm_birnn stores the checkpoints.
best checkpoint: lstm_birnn/lstm_birnn-epoch=09-val_f1=0.8915.ckpt

---

Conclusion

The Character-level Bi-LSTM model won out because of the fundamental difference between Uniform Resource Locators (URLs) and natural human language.

URLs Don't Mimic Natural Language
Conventional word-level tokenization, as well as complex, pre-trained models like BERT Tokenizers, are sub-optimal for this task.

Absence of Semantic Meaning: Models like BERT are trained on human-generated text to learn syntactic and semantic relationships—the grammar and meaning behind words. URLs, conversely, are highly structural sequences that often intentionally employ meaningless "word salad," character substitutions, or concatenations of non-standard terms for obfuscation.

Irrelevant Features: Applying a word-level or BERT-based approach forces the model to search for linguistic patterns that are virtually non-existent or irrelevant in a URL string, reducing signal recognition and missing critical patterns in the data.

Advantages of Character-Level Encoding

The Character-Level Tokenization strategy, coupled with the sequence-processing power of the Bi-LSTM, proved more effective because it leverages the URL's inherent structure:

Preservation of Structural Integrity: By treating the URL as a strict sequence of individual characters, the model is able to learn patterns defined by special symbols (e.g., . ? / :), unusual domain lengths, and the specific arrangement of these elements that signal malicious intent.

Detection of Obfuscation: Attackers frequently employ obfuscation techniques such as homoglyphs, excessive sub-domains, or unusual encoding. The Bi-LSTM, processing this character stream bidirectionally, excels at identifying these subtle, sequential anomalies which would otherwise be absorbed and lost within a larger "word" token.

Optimal Sequence Learning: The Bi-LSTM is ideally suited for this character-level output, as it effectively captures long-range dependencies across the entire sequence (from domain to path to query parameters). This architecture learns structural features relevant to maliciousness, which are purely sequential and positional, rather than semantic.

In summary, the character-level bi-lstm achieves higher F1 scores because it accurately models the URL as a complex, non-linguistic sequence, focusing on structural patterns and obfuscation tactics that are key indicators of a cyber threat. But, considering that I used a different dataset for the second round of testing, it's possible that the character-level bi-lstm happened to perform extraordinarily well on the dataset used in round one.

Citations

Pytorch Framework
Lighting Module
https://lightning.ai/docs/pytorch/stable/common/lightning_module.html

Lightning DataModule
https://lightning.ai/docs/pytorch/stable/data/datamodule.html

Google Colab
general use
https://colab.research.google.com/#scrollTo=-Rh3-Vt9Nev9
google.colab files
https://colab.research.google.com/notebooks/io.ipynb

Public Datasets
URLHaus Database Dumps
https://urlhaus.abuse.ch/api/
OpenPhish Phishing Feeds
https://openphish.com/phishing_feeds.html
Kaggle Malicious URL Dataset
https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset
Tranco
https://pypi.org/project/tranco/

Libraries and Other Tools
Python shutil
https://docs.python.org/3/library/shutil.html
Python urllib.parse
https://docs.python.org/3/library/urllib.parse.html
Python re
https://www.w3schools.com/python/python_regex.asp
Python Requests
https://pypi.org/project/requests/
Python Zipfile
https://realpython.com/python-zipfile/
Kagglehub API
https://www.kaggle.com/docs/api#getting-started-installation-&-authentication
Scikit-learn StratifiedShuffleShift
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html
Torch packed_padded_sequence
https://docs.pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pack_padded_sequence.html

Torch pad_packed_sequence

https://docs.pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pad_packed_sequence.html#torch-nn-utils-rnn-pad-packed-sequence

Pathlib

https://docs.python.org/3/library/pathlib.html