Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Sinan Sakaoğlu

KARTAL: Web Application Vulnerability Hunting Using Large Language Models

Novel method for detecting logical vulnerabilities in web applications with finetuned Large Language Models

Master's Thesis
Espoo, June 28, 2023

Supervisors:	Professor Antti Ylä-Jääski, Aalto University
	Docent Ben Slimane, KTH Royal Institute of Technology
Advisor:	Docent Ki Won Sung, KTH Royal Institute of Technology
	Marika Mörsky M.Sc. (Tech.), Netlight Consulting Oy

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

ABSTRACT OF MASTER'S THESIS

Author:	Sinan Sakaoğlu		
Title: KARTAL: Web Application Vulnerability Hunting Using Large Language Models Novel method for detecting logical vulnerabilities in web applications with finetuned Large Language Models			
Date:	June 28, 2023	**Pages:**	93
Major:	Computer Science	**Code:**	SCI3113
Supervisors:	Professor Antti Ylä-Jääski Docent Ben Slimane		
Advisor:	Docent Ki Won Sung, KTH Royal Institute of Technology Marika Mörsky M.Sc. (Tech.), Netlight Consulting Oy		

Broken Access Control is the most serious web application security risk as published by Open Worldwide Application Security Project (OWASP). This category has highly complex vulnerabilities such as Broken Object Level Authorization (BOLA) and Exposure of Sensitive Information. Finding such critical vulnerabilities in large software systems requires intelligent and automated tools. State-of-the-art (SOTA) research including hybrid application security testing tools, algorithmic bruteforcers, and artificial intelligence has shown great promise in detection. Nevertheless, there exists a gap in research for reliably identifying logical and context-dependant Broken Access Control vulnerabilities. We propose *KARTAL*, a novel method for web application vulnerability detection using a Large Language Model (LLM). It consists of 3 components: *Fuzzer, Prompter,* and *Detector*. The *Fuzzer* is responsible for methodically collecting application behaviour. The *Prompter* processes the data from the *Fuzzer* and formulates a prompt. The *Detector* uses an LLM which we have finetuned for detecting vulnerabilities. In the study, we investigate the performance, key factors, and limitations of the proposed method. We experiment with finetuning three types of decoder-only pre-trained transformers for detecting two sophisticated vulnerabilities. Our best model attained an accuracy of 87.19%, with an F1 score of 0.82. By using hardware acceleration on a consumer-grade laptop, our fastest model can make up to 539 predictions per second. The experiments on varying the training sample size demonstrated the great learning capabilities of our model. Every 400 samples added to training resulted in an average MCC score improvement of 19.58%. Furthermore, the dynamic properties of *KARTAL* enable inference-time adaption to the application domain, resulting in reduced false positives.

Keywords: Broken Access Control, Vulnerability, Large Language Models, Web Application, API, Detection, Scanner, DAST, Application Security

Language: English

Acknowledgements

I wish to thank both of my advisors and supervisors for their contributions to this thesis. I also express my gratitude to my mentor Lasse I., delivery coach Tuomas Y., and thesis coordinator Anna R. at Netlight Consulting for their guidance.

I would also like to thank my friends and my girlfriend for their support throughout the thesis. Finally, I'd like to express my heartfelt gratitude and appreciation to my mother, father and sisters, to whom I dedicate this thesis work.

To my father, thank you for your wisdom, support, and belief in me. You are my role model who has given me a strong sense of integrity, perseverance, and the courage to chase my dreams. Your track record of excellence in academia, work and fatherhood provided me with the confidence to take on any obstacle that comes my way. You inspire me to be my best self and have a meaningful life.

To my mother, thank you for your boundless love, continuous sacrifices, and nurturing spirit. You are the strongest woman in the world who never gives up against anything life throws in her way. To me, home is wherever you are. You have taught me the value of kindness, compassion, and the importance of always staying true to myself. Your unconditional love has been and will continue to be a constant source of comfort and encouragement in my life.

Espoo, June 28, 2023

Sinan Sakaoğlu

Abbreviations and Acronyms

AI	Artificial Intelligence
AMA	Ask Me Anything Prompting
API	Application Programming Interface
APR	Automatic Program Repair
BART	Bidirectional and Auto-Regressive Transformers
BERT	Bidirectional Encoder Representations from Transformers
BiLSTM	Bidirectional Long Short-Term Memory
BLEU	Bilingual Evaluation Understudy
BOLA	Broken Object Level Authentication
CD	Continuous Development
CI	Continuous Integration
CNN	Convolutional Neural Network
CORS	Cross-Origin Resource Sharing
CoT	Chain of Thought
CPU	Central Processing Unit
CSV	Comma Separated Values
CWE	Common Weaknesses Enumeration
DAST	Dynamic Application Security Testing
DNN	Deep Neural Network
EMPT	Exploratory Manual Penetration Testing
GLUE	General Language Understanding Evaluation
GPT	Generative Pre-trained Transformer
GPU	Graphical Processing Unit
GRU	Gated Recurrent Unit

HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IAST	Interactive Application Security Testing
ICL	In-Context Learning
IDE	Integrated Development Environment
IDOR	Insecure Direct Object Reference
LLM	Large Language Model
LMP	Language Model Programming
LMQL	Language Model Query Language
LSTM	Long Short-Term Memory
MAST	Manual Application Security Testing
MCC	Matthews Correlation Coefficient
ML	Machine Learning
NLP	Natural Language Processing
NN	Neural Network
OWASP	Open Web Application Security Project
PEFT	Parameter-Efficient Fine-Tuning
PET	Pattern Exploiting Training
PII	Personally Identifiable Information
RCE	Remote Code Execution
RNN	Recurrent Neural Network
SaaS	Software-as-a-Service
SAST	Static Application Security Testing
SDG	Sustainable Development Goal
SDLC	Software Development LifeCycle
SMPT	Systematic Manual Penetration Testing
SOTA	State-of-the-art
SPA	Single Page Application
SQL	Structured Query Language
TPU	Tensor Processing Unit

URL	Uniform Resource Locator
VpH	Vulnerabilities per Hour

Contents

Chapter 1

Introduction

Web applications have become essential services in our daily lives. As Software-as-a-Service (SaaS) market reached $167 billion in 2022 [1], web applications have become a lucrative target for hackers. By utilizing unpatched vulnerabilities and zero-day exploits, adversaries can get into systems and steal information such as payment details, passwords, and personal data. As a result, the significance of finding vulnerabilities before the code reaches the production environment and is exposed to the Internet has become increasingly crucial. There are a variety of tools available to detect flaws in applications. These tools mainly use a pre-written list of rules, vulnerability databases, or a custom-trained Artificial Intelligence (AI) model to scan the target application. However, they are not able to detect higher-level weaknesses that require logic and reasoning. Thus, there is a need for a new method for identifying difficult-to-detect vulnerabilities in the web application domain. Multi-tasking and few-shot learning capabilities of Large Language Models (LLMs) make them great candidates for solving this problem.

1.1 Background

Current application security testing methods are competent at finding vulnerabilities, however, each has unique drawbacks. Static Application Security Testing (SAST) tools are known for a high false positive rate and lack end-to-end testing, Dynamic Application Security Testing (DAST) tools tend to have low true positive rate and limited coverage, and Interactive Application Security Testing (IAST) tools are language-specific and require interaction [2–4]. Building on top of existing methods, novel solutions such as hybrid [5] and combined [4] were suggested to mitigate the drawbacks, which have shown improvements over using either method. Nevertheless, all of the methods lack the ability to detect vulnerabilities that require an understanding of context and reasoning. *Broken access control* is an example of such a group of vulnerabilities, which was the most frequently occurring set of flaws in web applications [6].

LLMs are AI models that have gained immense popularity over recent years. With the emergence of commercial and open source pre-trained LLMs with multi-billion parameters available to the general public, applying state-of-the-art AI models to under-investigated domains such as vulnerability detection is now cheaper, easier, and faster than ever before. In essence, these models are built to predict the next word in the given sequence of text, also called a *prompt*. The ingenuity of their design allows them to be agile, succeeding in a variety of Natural Language Processing (NLP) tasks they were not specifically trained for [7]. State-of-the-art research has explored various areas of cybersecurity applications such as Automatic Program Repair (APR). However, scanning web applications for potential vulnerabilities has not been scientifically explored yet.

1.2 Problem

Ever-increasing scale of modern web applications requires automated tools to detect flaws. Existing automated vulnerability scanning methods cannot reliably detect the Broken Access Control category of web application vulnerabilities as they do not possess the ability to logic or adapt. We specifically target complex and context-dependant Broken Object Level Authorization (BOLA) [8] and Exposure of Sensitive Information [9] vulnerabilities. BOLA is a security flaw that allows an attacker to alter object references and gain access to or conduct operations on unauthorized resources. When an application fails to enforce adequate permission checks depending on the user's privileges, a vulnerability occurs. The second security flaw occurs when messages created by an application mistakenly reveal sensitive information such as personal data, accounts, database queries, or application component details that an attacker can utilize. Exploiting these vulnerabilities can result in unauthorized data access, and privilege escalation, posing a severe danger to the security of the application.

1.2.1 Original Problem and Definition

This research presents a novel LLM-based web application vulnerability detection method, which we term KARTAL (“Eagle” in Turkish), and plans to answer the following research questions:

1. What is the performance of the proposed method in identifying selected class of web application vulnerabilities?

2. What factors contribute to the performance of the proposed method?

3. What are the challenges and limitations of the proposed method?

1.3 Purpose

The results of the finished work have the potential to reduce the cybersecurity incident risk of any organization that develops web applications. This can save the organization a large amount of money depending on its size and provide a competitive advantage. The people that will most likely benefit from reading the thesis are chief technology officers, chief information security officers, security engineers and analysts, tech leads, and software developers. Additionally, research in LLMs will extend to another domain where they can be applied.

The project aims to make contributions to the United Nations’ Sustainable Development Goals 8 (Sustainable economic growth), 9 (Resilient infrastructure), and 10 (Reduced inequality) [10] and will address potential unethical uses of the research. From a resource efficiency point of view, LLMs requires large amounts of resources to train from scratch. However, the project focuses on utilizing pre-trained LLMs, that are commercial or open source.

The host organization of the research is Netlight Consulting Oy [11]. Netlight is an innovative consulting company that provides genuine consulting services for leaders in the digital industry. They implement effective information technology solutions covering cybersecurity, e-commerce, finance, games, and the industrial Internet of Things.

1.4 Goals

The goal of this project is to automatically detect vulnerabilities that otherwise require manual expert review. This has been divided into the following three sub-goals:

1. Evaluate the performance and feasibility of the proposed method.
2. Identify the key factors that affect the performance of the proposed method.
3. Determine the challenges and limitations of the proposed method from technical, practical, and usability perspectives.

1.5 Research Methodology

In this study, we conduct experimental quantitative research. The research process consists of exploration, experimentation, and evaluation. We undertake a thorough literature review followed by data collection and dataset creation. Next, we define three different experiments. Firstly, a pretrained model selection experiment where we compare and contrast different base models for finetuning. Another experiment focuses on the inference performance, testing the latency of different base models running on various hardware settings. The last experiment tests the effects of training dataset size on the model performance. Using a selection of sequence classification metrics, we assess each experiment, analyze, and discuss them.

1.6 Delimitations

Building and training a LLM from scratch is out-of-scope as it would require a large set of resources and time which are not within the limits of this master's thesis. The research uses the commercially available GPT-3 by OpenAI for dataset generation and pre-trained MPNet, DistillRoBerTa, and MiniLM models for finetuning. A detailed comparison of algorithms behind the LLMs used in the paper will not be included. Implementation for components other than the Detector has been left out of scope as they do not contribute to answering the research questions.

1.7 Structure of the Thesis

Chapter 2 introduces relevant background information about vulnerability detection methods and large language models. Chapter 3 discusses the research methodology used in the paper. Chapter 4 presents our proposed novel method. Chapter 5 provides test results and analysis of the proposed method. Chapter 6 reflects on the analysis, evaluating results in terms of research questions and objectives. Chapter 7 concludes the paper with remarks and reflections of the author, in addition to offering recommendations for future work.

Chapter 2

Background

This chapter provides background information about Large Language Models and the foundational AI concepts on which it is built on. Additionally, this chapter describes web application vulnerabilities and their detection. The chapter also describes related work in LLM applications in advanced techniques in vulnerability detection.

2.1 Large Language Models

LLMs are pre-trained transformers that are trained on a large corpus of text [12]. They are the state-of-the-art AI models that are built on neural network architecture, namely deep learning [13, 14]. Although they are most successful at NLP tasks, they can solve tasks in a variety of domains such as Machine Translation [15], Dialog Systems [16], Medical Diagnosis [17], Computational Chemistry [18], and Data Augmentation [19]. Based on the text they are trained on, they can be monolingual or multilingual [14], understand programming languages [20], or even chess notation [21]. To achieve higher accuracy, the model can also be finetuned for a specific task. Furthermore, using In-Context Learning (ICL) the model adapts to new tasks on the fly, lowering the barrier to entry for organizations and businesses of smaller sizes to utilize LLMs. Understanding the importance of LLMs requires background knowledge of its predecessors and their shortfalls. In this section, we will examine the building blocks of LLMs and the reasons for their supremacy in AI.

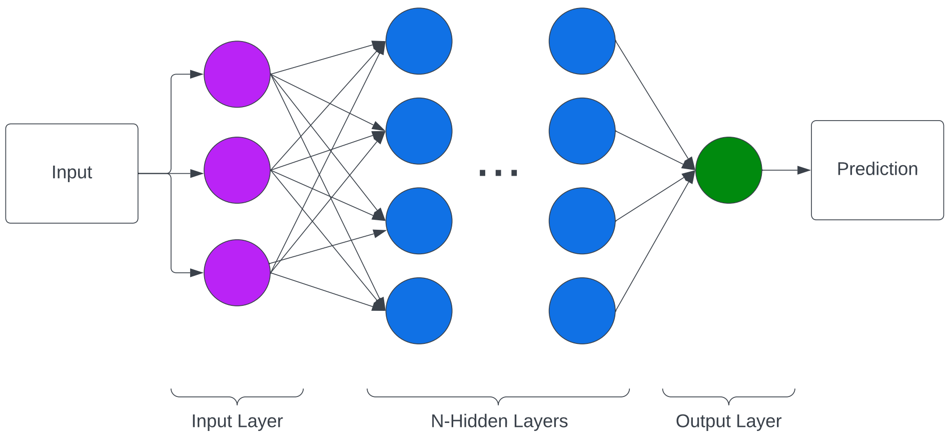


Figure 2.1: Deep Neural network architecture with input, hidden, and output layers. The prediction is a quantitative answer that can be a classification of an object, the probability of an event, or the next token in a sequence.

2.1.1 Deep Neural Networks

The first revolutionary AI models are Deep Neural Networks (DNNs). The main difference compared to the classical machine learning models is the depth and complexity of the networks [22]. DNNs use multiple interconnected hidden layers as can be seen in Figure 2.1. Deep learning techniques leverage the additional layers to allow the network to learn to solve an increasingly complicated set of problems, performing better on a wide range of tasks. Increasing the number of hidden layers allows the network to learn more abstract features, thus, improving the accuracy of the model. However, as the count of hidden layers grows, the computational complexity of the model will scale, putting greater pressure on the hardware.

Another side effect is vanishing or exploding gradients. This occurs during *the backpropagation* stage which updates the parameters of the network using the computed gradients of the loss function. When the gradients become too small (vanishing) or too large (exploding), the total learning rate of the network decreases. Finally, due to the diminishing rate of returns, at a certain point adding new layers will only provide negligible performance. However, there are various deep learning models that mitigate some of the mentioned issues and outperform a plain implementation of DNNs.

2.1.1.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of DNNs built to handle sequential data. They outperform plain Neural Network (NN) implementations in tasks such as NLP, time series prediction, and speech recognition [23]. The recurrent neurons in the hidden layers have cyclic connections to maintain a hidden state which acts as a memory of past states. The recurrent neurons produce output and update the hidden state at each time step. During the training phase of the network, this state is used as a secondary input to the neurons along the features vector. As a result, RNNs are able to model deeper relationships and dependencies within the input data, resulting in higher accuracy predictions. However, they suffer from the vanishing gradients problem. As the gradients are backpropagated from the output layer to the input, they become too small to effectively update the weights. This is caused by the weight of the hidden state which is shared across all time steps, creating a challenging problem for RNNs to efficiently learn long-term dependencies in the input [24].

2.1.1.2 Long Short-Term Memory Networks

As a response to the vanishing gradient problem, Long Short-Term Memory (LSTM) networks were developed. These networks use LSTM neurons which can selectively keep or discard information in the hidden state of the neurons over time steps [25]. These neurons contain three types of gates that work in order: *input, output, and forget gate.* The forget gate decides whether a piece of information from the last hidden state should be discarded/forgotten. In contrast, the input gate is responsible for choosing which information from the input should be retained and updated in the state. Finally, the neuron decides on what to output, using the output gate to filter the state. LSTM networks outperform RNNs in modeling long-term relationships within sequential data. In the domain of NLP, they have higher accuracy in tasks such as next-word prediction and translation. The main disadvantage of LSTM networks is the higher computational cost due to the additional logic introduced into the neurons.

An alternative approach to LSTM networks is Gated Recurrent Unit (GRU) networks. These networks also utilize multiple gates to read and update the hidden state, however, they are simplified. GRU networks essentially reorganize the functionality of the input, output, and forget gates from LSTM into a new set of gates: *update and reset* [26]. These gates are responsible for controlling the information flow in and out of the hidden state and to the output. With fewer gates and complexity, GRU networks are faster to train, have similar performance, and are less prone to overfitting compared to LSTM networks. However, they are inferior at modeling long-term dependencies.

2.1.1.3 Attention Mechanisms

A newer and revolutionary approach to the deep learning field is the *attention mechanism*, which has become an indispensable technique in solving difficult tasks including NLP and computer vision tasks [27]. It was originally developed within the neural machine translation domain to overcome the fixed-length vector problem in encoder-decoder based DNN models [28]. Inspired by human eyesight and cognitive psychology, they selectively ignore or focus on important parts of the input features. DNNs that utilizes the attention mechanism is substantially more efficient and accurate. Their superb understanding of long-term dependencies within the input sequence makes them great candidates for solving tasks that require deeper context and logical reasoning.

There are a few common types of attention mechanisms. *Soft attention* uses a softmax function to compute the distribution of attention scores of the input features. Meanwhile, the scoring function of the *hard attention* makes discrete binary selections. *Self-attention* scores the attention of the input features by comparing with themselves, enabling learning of context-aware representations. They are also a foundational component of *transformers*. Lastly, *multi-head attention* deploys multiple attention mechanisms in parallel, allowing the model to simultaneously focus on various parts of the input [12].

The disadvantage of employing attention mechanisms in DNNs is the increased computational complexity. Particularly with self-attention, the amount of required processing power scales quadratically with the input sequences, causing significantly longer training times and higher costs. Addressing computational and memory constraints early on is crucial for applications with large data sets.

2.1.2 Transformers

The most powerful LLMs today are based on the *Transformer* architecture, which was developed and published in 2017 [12]. Transformer-based LLMs such as PaLM and GPT-3 have been named *foundational models* as they are trained on a broad range of data and demonstrate great ability to adapt to many different downstream tasks [13]. This neural network architecture learns the context, importance, and relationship between words from natural

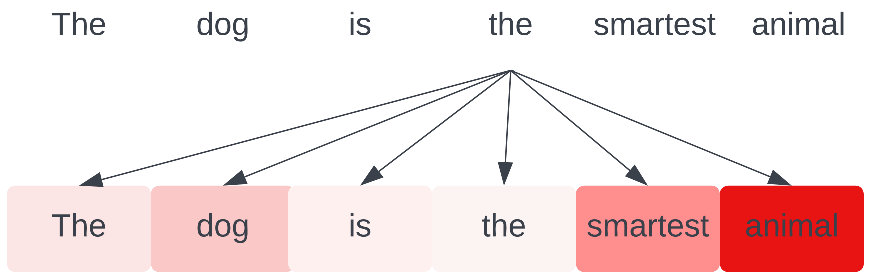


Figure 2.2: An example of self-attention scoring. Each token of the input is scored compared to other tokens. In this example, the token “animal” has the highest score relative to the token “the”. However, “animal” has a lower score relative to the token “the” at the beginning of the sentence.

language training data using the *self* and *multi-head* attention mechanisms. Their main innovation is, by design, they are highly parallelizable and perform well with unsupervised learning, and can be finetuned for a specific task with a relatively small amount of labeled data. The general architecture of transformers is demonstrated in Figure 2.3.

Transformers use a method called *Positional Encoding* to empower their rapid and parallel token processing capability. It is a crucial component of transformers, since without it, training the model with billions of points of data would take multiple magnitudes more computing power and time. Unlike RNN techniques such as LSTM or GRU, transformers do not inherently have information about the order of tokens in the input sequence. Positional encoding bridges this gap by generating a unique vector for each token and adding it to their embeddings. This method allows the network to learn and utilize patterns of token order while processing the tokens in parallel, maximizing speed without sacrificing accuracy.

Interpretability in DNN applications has been a great topic of discussion. Due to the vast number of hidden layers and their interconnectivity, mapping the decision-making process of DNN models is challenging. Transformer-based models also struggle with the same problem. As the models get larger and more complex, understanding their inner workings of it becomes increasingly difficult. This is an unfortunate disadvantage of DNNs as governance of the models from the safety and ethics point of view is low. Based on the

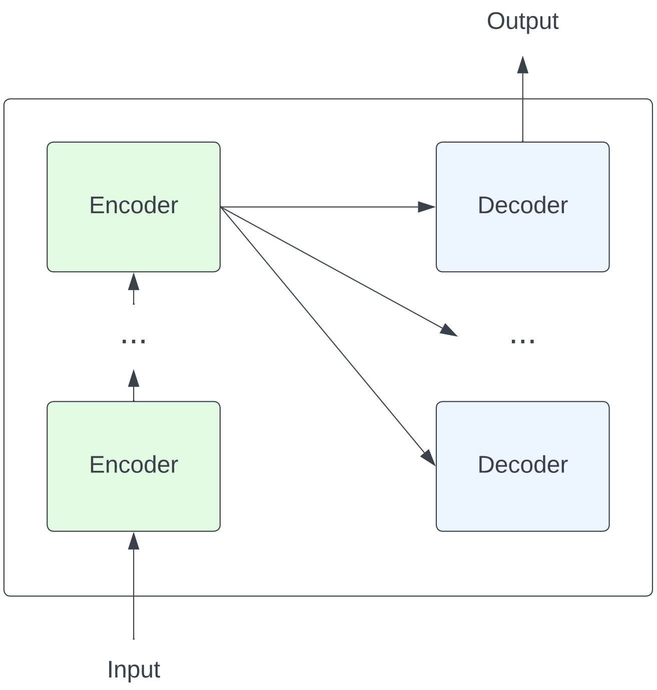


Figure 2.3: Simplified diagram of a Transformer architecture. It is made up of an Encoder-Decoder stack. The input is fed into the stack of n-encoders as token embeddings and encoded positions. Once the processing is done, the results are passed into each decoder, which feed-forward their results. Once the last decoder is done, predictions are ranked based on probability, and the highest-ranked token is served as the output.

training data, the models may develop a bias against certain ethnic groups, genders, or political leaning [29]. Fundamentally, the creator of any AI model should be able to confidently know and be able to demonstrate its safety measures for human usage. Additionally, making sure the model follows the ethical guidelines of its domain is of the utmost importance since abusing powerful AI models can lead to unfortunate consequences for its consumers.

Apart from NLP, transformers have been successfully applied in various fields. In computer vision research, *DALL-E* is an AI model that can generate images from natural language text descriptions [30]. Another application in computer vision uses transformer architecture to perform image recognition [31]. In the speech analysis domain, *Conformer* is a model based on transformers that are trained to recognize speech [32]. Medical applications such as drug research have also utilized transformers to predict molecule interactions and discover new therapeutic drugs [33]. Although all of these applications use the transformer architecture, the overall model structure varies in implementation. Depending on the application, transformer architectures can be used within encoder-only, decoder-only, or encoder-decoder networks.

2.1.2.1 Encoder-only Transformers

Encoding refers to mapping input data to a lower-dimensional representation. Encoder-only transformers do not have a decoder layer. They are known to perform well in unsupervised and transfer learning. Bidirectional Encoder Representations from Transformers (BERT) is an LLM that uses encoder-only transformer architecture [14]. It is trained using masked language modeling, which randomly hides certain tokens from the input. The goal of the pre-training is to predict the masked token. As a result, BERT can utilize right and left contexts to understand complex relationships of tokens. The success of BERT inspired many LLMs based on itself. RoBERTa uses dynamic masking of the input and trains with larger batches for better accuracy [34]. ALBERT is another LLM that aims to reduce the memory consumption of the model [35]. Another set of LLMs from the BERT family, DistilBERT and MiniLM uses *knowledge distillation* technique to compress the model run on lower-end hardware while performing similarly [36, 37]. A different approach that combines masked and permutated language modeling, MPNet, achieved better benchmark results compared to the BERT-like models [38].

Built on top of encoder transformers, *Sentence Transformers* are used to convert sentences into high-dimensional vector representations [39]. Sentence transformers are capable of capturing semantic linkages and contextual information inside sentences by utilizing cutting-edge models of transformers and pre-training on big corpora. These models are excellent at tasks such as semantic similarity, paraphrase identification, and information retrieval. Furthermore, the properties of sentence transformers make them great at classification applications where the class label of the input depends on the input itself.

2.1.2.2 Decoder-only Transformers

In contrast, decoder-only transformers ingest lower-dimensional representation and generate an output of the same dimension as the unencoded data. They are generally used in generative DNNs for objectives such as generating text, images, or audio. Generative Pre-trained Transformer (GPT)-2 and GPT-3 both utilize decoder-only transformer architecture [7, 40]. They have

1.5 and 175 billion parameters, respectively. These models are pre-trained by feeding in a large corpus of text while the tokens to the right end of the input window are masked. A downside of this approach is that, unlike BERT, GPT-2 is not bidirectional. The training objective is to predict the next word which makes GPT-2 great at generating coherent and contextually relevant text. However, when scaled, these models perform excellently in a variety of NLP tasks.

2.1.2.3 Encoder-Decoder Transformers

A combination of both models is encoder-decoder transformers. This architecture demonstrates great performance in mapping deep and complex relations in the input sequence and solving challenging NLP tasks such as machine translation. Bidirectional and Auto-Regressive Transformers (BART) is an example of an LLM that employs this architecture [41]. Instead of masking tokens or predicting the next token, BART uses *text infilling* corruption which replaces some text spans with a single mask token. The pre-training objective is to predict the original uncorrupted text. T5 takes a whole other approach, it uses supervised and self-supervised training [42]. The model is trained on data sets such as General Language Understanding Evaluation (GLUE), a multi-task benchmark containing 9 diverse natural language understanding and processing tasks [43]. By casting all NLP tasks as a text-to-text problem with special prefixes in the training examples, it achieves state-of-the-art performance.

2.1.3 Pre-training

After preprocessing the training data and setting up the model, pre-training is the next step. Although the prefix “pre” hints at a later and heavier training stage, this is not the case. Pre-training is the process of training an LLM on a large and general dataset before finetuning it for a specific set of tasks. This allows the model to understand the fundamentals and patterns of the language(s). The pre-trained model can later be adapted to a variety of downstream tasks with ease. The process starts by collecting a large dataset that consists of diverse text from sources such as websites, wikis, books, and magazines. This dataset should cover all patterns of the target domain. For example, if the target domain is NLP, the dataset should contain a large corpus of writing, possibly in multiple languages. In contrast, if the domain is music then the LLM be trained with numerous compositions in musical notation format.

Next, the architecture of the LLM should be modeled according to the chosen transformer variant and model. The final step before training is to define the pre-training objective. This should be chosen based on the target domain. The most common objectives are guessing the masked tokens (Masked Language Modeling), guessing the next token (Autoregressive Language Modeling), and uncorrupting the input (Denoising Autoencoder). Then, the model is ready to start pre-training, which is the most resource and time-consuming phase of building an LLM. To prevent data loss and enable easy sharing of the model, checkpoints in time can be saved and backed up. These checkpoints store the entire weight matrix of the network and can be used to continue pre-training or move to the finetuning phase. Table 2.1 compares different ways of training LLMs.

Stage	**Pre-training**	**Finetuning**	**In-context Learning**
Goal	General language understanding	Adapt to specific tasks	Learn from user input
Data	Unlabeled, Millions-Billions data points, general	Labeled, Hundreds-Thousands data points, task-specific	Labeled, 5-10 data points, task-specific
Phase	Initial	After pre-training	During inference
Advantages	Broad knowledge, transfer learning	Great accuracy on specific tasks, reduced training time	Good accuracy on a specific task, cheap, no training required
Limitations	Limited domain-specific knowledge, Highest cost, slowest to train	Overfitting, medium cost, slow to train	Limited context, temporary learning

Table 2.1: Comparison of pre-training, finetuning, and in-context learning for LLMs

2.1.4 Finetuning

Finetuning is the process of further training a pre-trained AI model to receive the best performance at a specialized task [44]. The pre-trained model is adapted to downstream tasks by using a labeled dataset. Finetuning utilizes the general language capabilities of the existing model, allowing to use of a fraction of the original dataset to achieve superior performance. First, the data should be prepared by a process of curation and labeling. Depending on the task, this can be a set of sentence pairs for machine translation or question & answer samples. Next, the model is initialized using the pre-trained weights. An earlier checkpoint might be a better option to start with if the data is overfitted at the end of pre-training. Finally, the model is evaluated against the test set of data. In the context of pre-trained LLMs, finetuned models can effortlessly outperform the base model in *zero-shot* settings with only a training dataset few hundred prompt and response examples [13].

With finetuning, the learning from the training samples is preserved. This requires the entire model to be loaded and initialized as weights of each layer in the neural network may be affected. The computational requirements for finetuning are less than the pre-training phase and can be further reduced with efficient finetuning techniques. Since finetuning is a stateful process, the resulting model no longer has the same properties as the foundational model. When trained to fit a specific task the finetuned model may lose its multitasking capabilities, achieving subpar results for general tasks [45]. Furthermore, finetuning a model can reduce its generalizing ability, preventing the use of ICL techniques such as *few-shot* [46].

2.1.4.1 Efficient Finetuning Techniques

Recently, efficient finetuning strategies have evolved to further reduce the computational and time requirements of finetuning. One such technique is Parameter-Efficient Fine-Tuning (PEFT). These methods aim to lower the number of trainable parameters in order to improve the model training speed. One of the approaches uses *Adapter Modules* that are inserted in-between layers. By freezing the rest of the pretrained model, finetuning can be accelerated by multiple magnitudes with comparable performance to fully finetuned models [47]. More advanced techniques of PEFT such as *(IA)*^{3} have been shown to finetune models that outperform the human baseline on certain benchmarks [48].

In contrast, Pattern Exploiting Training (PET) is a semi-supervised training method that improves model performance when the labeled training dataset size is small [49]. PET rephrases the training samples into a cloze test format by modifying them into one of the premade patterns. A verbalizer mapping is made between possible classes and words in vocabulary to describe each of them. For each pattern, an LLM is finetuned using a small dataset. Next, the set of finetuned LLMs are combined to soft label a larger dataset. Finally, a classifier is trained on the new augmented dataset.

SetFit is another efficient finetuning method that shares the goal of increasing performance under low-resource settings [50]. Similar to PET, it first finetunes a pretrained transformer on a small set of text pairs in a contrastive Siamese form. Next, a classifier is trained using the rich text embeddings from the finetuned model. SetFit is an order of magnitudes faster to finetune while achieving comparable results to PET and PEFT. As an additional advantage, SetFit does not use prompts or verbalizers, resulting in a simpler and swifter finetuning process.

2.1.5 Prompting

Beginning as a means to interact with a LLM, prompting has turned into a science. A prompt is the query of input text passed to the model as token embeddings. Since most LLMs are trained on massive amounts of literature, they demonstrate a great understanding of natural human language. The prompt can directly query the model and does not require a specific format. For example, the "Is an apple a fruit?" prompt gives a task to the model. First, the model must interpret this question as a classification task. To answer this question the model should be able to understand what constitutes a fruit and how an apple relates to it. Once the model "decides" on the answer, it responds to the user with the answer in natural language, i.e. "Yes, an apple is a fruit.".

Prompts can go beyond simple questions. A variety of NLP tasks can be done out of the box after the pre-training phase such as summarization, sentiment analysis, paraphrasing, open-ended questions, and similarity detection. These tasks constitute the core capabilities of human understanding and are commonly used in testing the performance of the LLMs. The "thought" process of the LLM can also be queried, resulting in a detailed Chain of Thought (CoT) which can be used for determining knowledge gaps in the model. Depending on the prompt, the model may require additional knowledge and context to reply. For example, a model that was trained using data from 2021 cannot give accurate answers about an event that happens in 2022. In such cases, information unknown to the model can be prepended in the prompt to be used as an *oracle* for the model. This method can be further developed to teach new tasks to the model with the concept named ICL, examples of which can be seen in Figure 2.4.

A major limitation with prompting is the length of the input. As mentioned in 2.1.1.3, the amount of computation required when using the self-attention mechanism scales quadratically with the token count. Thus, the amount of external knowledge insertable into the model during inference is limited. Furthermore, the token count restrictions apply to the reply of the LLM as well, effectively doubling token/compute costs. There are some approaches to optimize for accuracy per token. BatchPrompting merges multiple questions with the same context into a combined prompt, while achieving comparable or even better results [51]. MultiStagePrompt dismantles a prompt into a consecutive set of smaller prompts results of which are fed into each in a chain [52]. Assessed in language translation tasks using Bilingual Evaluation Understudy (BLEU) [53], this method performs better than single-stage prompts. A different method involves using a novel method named Language Model Query Language (LMQL) [54]. It uses the Language Model Programming (LMP) concept to optimize prompts with scripting, increasing the accuracy of responses while minimizing token counts.

2.1.5.1 In-context Learning

As a result of their general understanding capabilities, LLMs can learn new tasks from the prompt during inference. In essence, the model can temporarily learn how to solve a problem or generate similar examples based on the context of the input with ICL [40]. At first, the user provides a description of the target task and possibly some examples of how to accomplish it. Next, the user appends the task in question to the prompt. Once enough contextual information is in the prompt, the model can be queried. Depending on the response, the user can use an iterative refinement process to provide better context and send a new query to the model, repeating until the desired outcome is reached. The greatest advantage of ICL is flexibility. Users can achieve performances comparable to finetuning without spending a large amount of computing power or time. Additionally, the need for labeled data is further reduced, all the model needs is the task description and optionally some examples. The downsides of ICL are its inconsistency and limited context. Due to the token limitations, it is not possible to fit an entire subject in a prompt, reducing the number of applications being able to utilize it.

2.1.5.2 Zero-shot Learning

A subgenra of ICL is zero-shot learning. Depending on the task and the model, it can easily outperform pre-trained transformers [55]. The term “zero-shot” refers to the lack of examples in the training data and the prompt

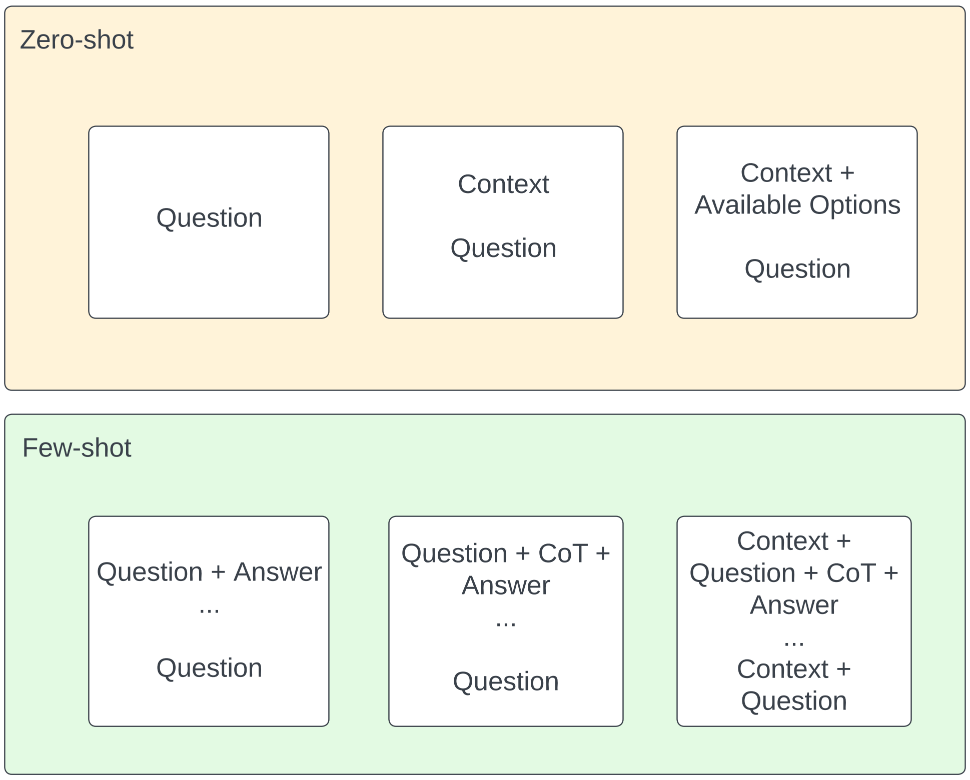


Figure 2.4: Example formats for Zero-shot and Few-shot prompts. Zero-shot prompts consist of questions, which may require context or data. Additionally, a prompt can give a list of available options to answer from. In few-shot prompts, the question is supported by a set of examples and answers. Advanced techniques can also include CoT and the context for increased performance.

context. With zero-shot learning, the model relies solely on their reasoning, CoT, and the context of the prompt, raising the importance of writing clear and in-depth queries. The main advantage of this approach is that it does not require any labeled data. As demonstrated in Figure 2.4, the prompt only includes the problem and context. Zero-shot prompts also remove input bias and sensitivity, improving output quality and originality. The disadvantage is lower performance relative to finetuned models. Without any pre-existing knowledge of a completely new task, the LLM suffers in the accuracy of its predictions. Furthermore, the answers can be rather inconsistent due to the low confidence the model has in its predictions, outputting increasingly random and non-relevant information.

Techniques such as NPPrompt can improve the zero-shot performance of models by engineering a clear prompt and augmenting the results of the LLM [56]. However, the greatest factor in the performance of zero-shot prompts remains to be the data fed into the pre-training phase [57]. Another method for better zero-shot accuracy is finetuning. When finetuned using multiple NLP datasets in natural language instruction format, the resulting model outperformed its base model in zero-shot prompts [44]. Thus, combining multiple techniques to optimize for the learning capabilities of the model is a recommended way to create a LLM model that is great at multitasking.

2.1.5.3 Few-shot Learning

Few-shot learning is a technique used in the post-training stage where the prompt has prepended a set of examples for the task in question. During inference, the model learns how to accomplish the task from the prompt itself and responds with an appropriate answer. Few-shot prompting demonstrates enhanced learning capabilities compared to zero-shot, improving the performance of the model in new tasks. This technique can increase the accuracy of the model by orders of magnitude and the performance scales with the number of examples provided in the prompt [40]. Figure 2.4 presents multiple examples for few-shot prompting. The simplest method is to provide a few sets of question-answer pairs, followed by the question the LLM will answer. Increasing the number of examples and their similarity to the final question will improve the correctness of the output. Additionally, the context required to answer the question can be prepended to the prompt as well. The context can help to steer the model in the right direction. For example, if the question is asking “How many shoes are there in the world?” providing the count at an earlier year or the up-to-date human population will improve the quality of the reply.

Another technique is to add the CoT process to the example pairs. In cases where the LLM cannot map the relationship between the question-answer examples, the accuracy of the model can suffer. Providing the CoT to acquire the answer allows the model to walk through the path to the solution, achieving superb performance and surpassing even finetuned models [58]. A different approach uses the generational capabilities of the LLM to augment the prompt. Ask Me Anything Prompting (AMA) method generates a series of answers and questions about a given context [59]. Next, the generated few-shot prompt is queried multiple times to get a distribution of replies from the model. Finally, using weak supervision, the answers are combined into a single final prediction. With AMA, a small model can outperform models 30x their size, without finetuning or specialized pretraining.

The downside of few-shot prompting is that learning is not retained as the weights in the model are frozen. Essentially, every prompt has to include the necessary knowledge and examples to solve the next problem. As a result of the token count restrictions mentioned in 2.1.5, the few-shot prompt has fewer tokens reserved for the target question and output. This limits the complexity of the questions the LLM can solve. Moreover, few-shot prompts are affected by the input sensitivity of LLMs. Depending on their training, the model can respond completely differently due to a slight change in the phrasing or choice of examples in the prompt [60]. Thus, when engineering few-shot prompts, wording, amount of examples, order of the prompt, and the question format should be first experimented with, then selected carefully.

2.2 Web Application Vulnerabilities

Web applications are substantially more vulnerable to hacking than other forms of software. Due to their accessibility over the Internet and dependency on several technologies, frameworks, and protocols, web applications are by their very nature more vulnerable to cyberattacks. The number and type of vulnerabilities used to perform these attacks is large, which can make it difficult to understand and defend against them. In addition, some vulnerabilities are more common and pose a higher risk than others. Application developers must navigate the vast landscape of vulnerabilities and build their applications with safety precautions while battling project deadlines. Hence, picking their battles and focusing on the top vulnerabilities is the most effective strategy. The Open Web Application Security Project (OWASP) routinely produces a list of the top ten security risks, known as the OWASP Top Ten, to help businesses become more aware of and help secure their online applications [6]. The list includes a description, examples, mitigations, and a list of mapped Common Weaknesses Enumeration (CWE) [61] identifiers for each category.

2.2.1 Broken Access Control

The most frequently occurring and complex vulnerability category in web applications is Broken Access Control, which poses a serious danger to web applications. Access control policies aim to keep users operating within the confines of their assigned permissions in order to prevent failures. When access is provided to anybody rather than being limited to certain skills, roles, or users, it violates the principle of least privilege, which is a frequent weakness in access control. This can potentially result in unauthorized access to sensitive data and system features. Attackers can also get through access control checks by altering the Uniform Resource Locator (URL), internal application state, HyperText Markup Language (HTML) page, or by employing an attack tool to change Application Programming Interface (API) requests, leading to illegal activities and data modification. Moreover, by permitting API access from untrusted sources and allowing users to visit authenticated or privileged pages without sufficient authorization, respectively, Cross-Origin Resource Sharing (CORS) misconfiguration and force browsing can also assist in creating a variety of vulnerabilities.

2.2.1.1 CWE-639: Authorization Bypass Through User-Controlled Key

Commonly, web applications retrieve a user record using a key value that the user controls, such as a sequence of numbers. This key would be used to search for an object that the user is authorized to provide that record to the user. If the application has the Authorization Bypass Through User-Controlled Key weakness, the authorization procedure would not correctly check the data access operation to confirm that the authenticated user conducting the action has adequate permissions to complete the request. Thereby, evading any further permission checks in the system. This can cause sensitive data leaks, unauthorized data manipulation, and privilege escalation. Also named Insecure Direct Object Reference (IDOR) or Broken Object Level Authentication (BOLA), are difficult to detect. Modern web applications have dozens of different entities with various access control rules, which may also be nested within each other. This creates a great challenge for existing tools as the search space and complexity for BOLA weakness grows exponentially with the number of entities and endpoints. Additionally, the access control rules can differ from one application to another, requiring considerable custom configuration for each automated tool. The mitigation of such weaknesses includes adopting secure and random key generation techniques, establishing robust authorization procedures with strict checks, and implementing strong input validation to guarantee the correct formatting and validation of user-controlled keys.

2.2.1.2 CWE-209: Generation of Error Message Containing Sensitive Information

A subset of the Exposure of Sensitive Information to an Unauthorized Actor (CWE-200) weakness is the Generation of Error Message Containing Sensitive Information. By giving detailed information about faults that happened inside an application, error messages are often created to help users and developers address problems. However, these messages could unintentionally reveal sensetive information that can compromise the security of the application such as debug logs, database queries, system configurations, and business logic. Usernames, passwords, and other Personally Identifiable Information (PII) may also fall under this category. Sensitive information may be leaked in error messages for a number of reasons, such as bad coding techniques, broken error handling, or inadequate input and output validation. For instance, a password reset form can include the full name and email of the user without authentication. Detection of this weakness consists of an end-to-end manual and automated analysis. By comparing the application response against a database of patterns and samples, automated tools can detect common variations of CWE-209 such as SQL server messages or program stack traces. Nevertheless, in more complex cases such as the reset password example, an understanding of the language, business logic, and application domain is required. Implementing effective error handling, verifying and sanitizing user input, and anonymizing or sanitizing sensitive data inside error messages are the fundamental mitigations in reducing the likeliness of CWE-209.

2.3 Vulnerability Detection

Finding vulnerabilities in software is a difficult task. Understanding the relationships between different components and the potential attack vectors becomes more and more challenging as software applications continue to expand in size and complexity. The effects of undiscovered vulnerabilities can be severe given the growing reliance on software systems in many facets of our life, from company operations to essential infrastructure. A web of dependencies is created by the reliance on third-party libraries, services, and APIs by modern systems. Vulnerability detection is made more difficult by the fact that these dependencies may introduce new vulnerabilities or exacerbate already existing ones. Furthermore, Agile and DevOps approaches have become more popular, which has sped up development cycles and increased the frequency of releasing new features and upgrades. This could lead to a situation where security flaws are unintentionally introduced or missed in the quest for quicker releases.

The strategies used by cyber attackers have evolved, as they nowadays use cutting-edge approaches to exploit weaknesses and avoid detection. In order to keep ahead of new threats, vulnerability detection systems must continually adapt and grow. To maintain the security of applications, challenges in detection have a number of crucial elements that must be addressed. As the applications scale, the time and computing requirements increase with it, underscoring the need to quickly and precisely identify flaws in complex systems. Additionally, when optimizing for coverage it can be difficult to strike a balance between true positives and false positives. Turning up the sensitivity of scanning tools will result in increasing both metrics and overwhelming security teams with warnings. Lowering the sensitivity and power will create the opposite effect, granting a false sense of security to the application. Lastly, the attacks adversaries use have started to exploit multiple weaknesses in a system, either in parallel or in a chain. These are much harder to detect, and a few low-risk weaknesses by themselves might not trigger alarms in any system. However, when combined, these weaknesses could be exploited to create a high-risk vulnerability such as a Remote Code Execution (RCE).

2.3.1 Manual Application Security Testing (MAST)

Security experts use manual application security testing as a technique to manually examine and evaluate the security of software applications. The tester manually finds possible security flaws, assesses their severity, and makes mitigation suggestions. From the beginning of computer programming, manual application security testing has been an essential step in the software development process. As systems got more sophisticated and the Internet evolved, attackers had more opportunities to take advantage of security flaws in software applications. In order to keep up with the constantly shifting threat landscape, manual application security testing has developed throughout time, encompassing diverse approaches and procedures.

The amount of skill and knowledge involved in manual application security testing is one of its main benefits. Particularly in the context of business logic errors, access control problems, and other sophisticated security threats. Security experts may spot complex vulnerabilities and attack routes that automated technologies can overlook [62]. Testers are able to investigate numerous user interactions and examine the software's response under various circumstances. As a result, manual testing offers a more extensive and nuanced knowledge of the behavior of the program. Manual testing consists of penetration testing, threat modeling, code review, and security-focused design reviews. With the simulation of actual attacks on the application, penetration testing allows testers to identify vulnerabilities and gauge the application's resistance to such attacks. Early detection of possible threats and vulnerabilities using threat modeling enables developers to address these risks before they show up in the finished product. Next, security professionals study the application's source code in a process called *code review* to find vulnerable spots and potential attack vectors. Lastly, security-focused design reviews assess the software's entire design and architecture to make sure security issues are properly addressed at all levels.

Effective manual testing might be difficult to carry out due to the growing complexity of modern web applications and the rapid speed of software development. Testers must also possess a broad and constantly rising skill set due to the use of a growing variety of programming languages, frameworks, and technologies. When multiple testers are involved, the human component of the manual testing method might result in variations in the process' quality and completeness.

2.3.2 Static Application Security Testing (SAST)

SAST is a white box approach where the source code is parsed and analyzed to automatically detect vulnerabilities. Figure 2.5 demonstrates the relationship between different security testing methodologies. By identifying problems early in the Software Development LifeCycle (SDLC), SAST attempts to provide developers the opportunity to fix them before they affect the production environment. By examining the code without executing it, SAST can provide insights into the application's behavior and possible security risks. The need for more secure applications became obvious as the Internet's expansion exposed software to a wider range of dangers, and enterprises began to notice the costs associated with security breaches. Early SAST techniques were primarily concerned with spotting frequent code faults like input validation problems and buffer overflows.

The key advantage of SAST is the capacity to uncover vulnerabilities early in the SDLC, decreasing the cost and time necessary to address them [2]. Moreover, SAST can precisely pinpoint the position and characteristics of the vulnerability by analyzing the code directly, making it simpler for developers to fix the problem. They consistently achieve better code coverage, finding vulnerabilities other methods miss. State-of-the-art SAST tools use cutting-edge analysis methods like data flow analysis, control flow analysis, and taint analysis to find numerous code vulnerabilities. Data flow analysis follows the path taken by data as it moves through the program to spot problems such as improper data processing or storage. In order to identify potential security risks associated with the application's logic, control flow analysis examines the application's execution paths. To find potential injection vulnerabilities, such as Structured Query Language (SQL) injection or cross-site scripting, taint analysis monitors the flow of unauthorized user input through the application.

One of the main drawbacks of the SAST tools is the potential to produce false positives when the tool alerts users to a problem that is not actually a vulnerability [63]. Developers may squander time and money looking into and trying to fix problems that do not exist. Moreover, SAST might not be able to detect all vulnerabilities, especially those that depend on the behavior of the program during runtime, like some access control or business logic problems [62]. SAST is becoming increasingly difficult and time-consuming due to the complexity of modern applications, especially the growing usage of third-party libraries and frameworks. Additionally, SAST tools are unable to fully comprehend the environment in which the application operates, which could leave gaps in the analysis. Finally, SAST tools use parsers, de-obfuscators, and syntax tree builders which are language and platform dependant. Thus, the types of projects they can be used on are limited.

2.3.3 Dynamic Application Security Testing (DAST)

As a black-box method, DAST, tests the apps without having access to the source code. It examines an active application to find potential security holes and vulnerabilities. DAST simulates attacks on the application by interacting with its exposed interfaces, such as web pages or APIs, to assess the program's replies and behavior under various scenarios. This type of testing aids in identifying vulnerabilities brought on by the configuration or runtime environment of the program. Early DAST technologies mostly targeted online applications, namely SQL injection, cross-site scripting, and weak authentication. DAST tools now handle a variety of application types, including APIs and mobile apps, and have widened their coverage to detect a larger range of vulnerabilities.

The advantage of a DAST is its ability to find vulnerabilities that static analysis might miss, like configuration errors, access control problems, or vulnerabilities resulting from the interaction of application components [3]. Additionally, it can aid in locating weaknesses connected to third-party com-

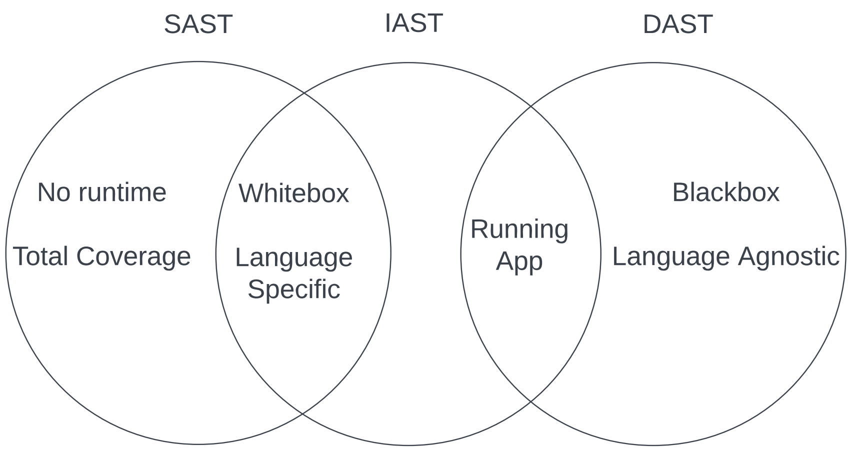


Figure 2.5: Methodology diagram of SAST, DAST, and IAST

ponents or services which may only surface during runtime. The effectiveness of runtime security measures such as web application firewalls or intrusion detection systems can also be verified by DAST. Most recent solutions employ a variety of sophisticated techniques. Sending erroneous or unexpected input to an application is known as "fuzzing," and it is used to find flaws and strange behavior. In combination, by utilizing automated crawling, DAST tools easily find and communicate with application components like web pages or API endpoints, that may otherwise be hidden. Moreover, DAST tools assisted by machine learning algorithms reduce false positives and negatives of anomalies suggestive of vulnerabilities or attacks.

However, DAST may become more difficult to utilize and less successful in some circumstances due to the growing usage of complex client-side technologies [63]. The fact that DAST requires the application to be running and frequently includes interacting with it in real-time means that it can occasionally be slower and more resource-intensive than SAST. Moreover, DAST might not be able to spot vulnerabilities that only appear under particular circumstances, including concurrency problems or race conditions. In addition, because DAST examines the behavior of the program rather than its code, it often offers less specific information than static analysis on the location and kind of vulnerabilities [62]. Finally, DAST may need substantial configuration and modification to test an application properly, particularly for those with complicated and non-standard communication protocols.

2.3.4 Interactive Application Security Testing (IAST)

IAST is a hybrid method where the testing tool connects to the application via an *agent* and analyzes the application during runtime. As a result, IAST is able to identify vulnerabilities by watching how the application's parts communicate with one another, how its data flows, and how it interacts with its runtime environment. By combining approaches from both methods, it inherits the advantages of both SAST and DAST. The main advantage of IAST is its ability to provide precise and actionable information about vulnerabilities, as it can pinpoint the exact location in the code and the conditions under which the vulnerability can be exploited [4]. This level of detail helps developers remediate vulnerabilities more efficiently than with DAST alone. Additionally, IAST minimizes the rate of false positives and false negatives compared to SAST and DAST by integrating static code analysis with dynamic runtime analysis.

IAST also enables real-time vulnerability identification, enabling developers to identify and address security vulnerabilities as they emerge throughout the development process. With the use of these AI technologies, IAST tools are better able to comprehend the organization of the program, spot patterns that point to vulnerabilities, and further minimize false positives and negatives. Integrated Development Environment (IDE), build systems, and continuous integration and continuous delivery (Continuous Integration (CI)/Continuous Development (CD)) pipelines are a few examples of popular development and DevOps tools that modern IAST tools integrate with. This makes it simple to incorporate security testing into the software development lifecycle.

Nonetheless, there are areas that this technique may perform poorly. IAST tools might not discover vulnerabilities that emerge from certain hardware or operating system settings. Additionally, IAST's dependency on instrumentation might present issues when testing applications created using languages that do not have mature or suitable instrumentation support. Depending on the application framework, IAST tools might cause compatibility or performance concerns. They may also not be appropriate for some applications, including those that primarily rely on client-side technology such as Single Page Application (SPA). When working with third-party components, notably, IAST additionally needs access to the application's source code or binaries, which is not always possible.

2.4 Related works

In this section, we will provide an overview of state-of-the-art web application vulnerability detection tools and methods.

2.4.1 Web Vulnerability Detection Tools Comparison

Amankwah et al. [64] compared eight commercial and open-source DAST tools. The study used the OWASP DWVA and WebGoat applications to benchmark the tools. These applications are purposefully vulnerable for testing and training reasons. They have documented list of vulnerabilities from each OWASP Top Ten category, which is great for researching new tools and techniques. The result of the testing showed that none of the tools were able to detect authentication flaws. Furthermore, only three of the tools were able to find access control flaws. These flaws are ranked seventh and first on the OWASP Top Ten list, respectively. This result greatly highlights the need for an open-source tool that can detect such vulnerabilities. Additionally, open-source tools failed to detect high-severity vulnerabilities while reporting a high number of false positives. Commercial closed-source performed better, however, the highest precision among them was 68%, which is less than some of the open-source counterparts.

Mateo Tudela et al. [4] analyzed the effectiveness of combining multiple vulnerability detection tools. SAST, DAST, and IAST use different techniques and are each better at detecting certain types of vulnerabilities that the other tools might not be strong at. The study compares all permutations of combining two SAST, two DAST, and two IAST tools. Half of the tools are open-source. The tests showed IAST tools consistently outperforming SAST and DAST in almost all metrics. One IAST tool had 100% accuracy and recall. The best-performing permutation was the IAST+IAST+DAST tool combination. Combinations including SAST tools had the highest recall, however, they also received high false positive rates. Similar to Amankwah et al. [64], open-source tools had a lower rate of finding high-severity vulnerabilities. The dataset for the study was the OWASP Benchmark project, which consists of a number of documented vulnerabilities across all categories of the OWASP Top Ten. However, the study did not identify any flaws in access control or authentication categories during the testing. Albahar et al. [65] also tested multiple open and closed source scanners on the OWASP Benchmark project. The results showed a maximum of 70% OWASP Top Ten coverage by a commercial DAST tool.

A grand study from Elder et al. [62] scanned a real-world production application using SAST, DAST, and two different methods. A systematic approach where the analyst designs and documents the security objectives before the testing starts is Systematic Manual Penetration Testing (SMPT). The unstructured version of SMPT is Exploratory Manual Penetration Testing (EMPT). The system under testing was OpenMRS, a large open-source medical record management system. In total, two DAST and three SAST tools were tested. Out of all methods, EMPT was the most effective in terms of Vulnerabilities per Hour (VpH), the most severe vulnerabilities found and the count of OWASP Top Ten categories covered, followed by SMPT, SAST, and DAST. Manual methods outperformed the automatic testing tools in Identification and Authentication Failures, and Security Logging and Monitoring Failures. Additionally, in all categories, manual methods had higher high-severity vulnerability rates. Overall, SAST found the most weaknesses, of which more than half were low severity. Although manual techniques demonstrate the best results, their success and required effort depend on the analyst. Additionally, manual testing does not scale with larger projects.

2.4.2 State-of-the-art Vulnerability Detection Tools

Our research in the literature showed a lack of vulnerability detection methods for authentication and authorization categories. We believe this is mainly due to the difficulty in detecting such weaknesses in web applications as their logic is complex and dynamic. Until the rise of LLMs, training an AI model powerful and capable enough to solve this task was out of reach for most researchers including ourselves. SOTA tools appear every day in the research scene. However, not many of them document and open-source their source code. As a result, it is not straightforward to find vulnerability detection tools that can be run and compared. Although we prefer such tools, we will be reviewing tools that do not have publicly accessible code as well.

Guo et al. [5] proposed *HyVulnDetect*, a hybrid method that uses a code property graph together with a Bidirectional Long Short-Term Memory (BiLSTM) model, the results of which are used by the classifier to determine if the code snippet is vulnerable. The major downside of this approach is that it requires multiple steps of preprocessing to get data to the algorithm. Specifically, creating the code property graph uses the Joern tool which works with various programming languages. Meanwhile, the Bidirectional LSTM model, utilizes a pre-trained word2vec model. Even though it is a white-box detection method, it cannot pinpoint where the vulnerability is in the given code snippet. Nonetheless, it achieves great accuracy in detection and provides better performance compared to existing rule-based vulnerability mining tools including Flawfinder and Cppcheck.

DEKANT, an AI model that treats code as natural language, was introduced by Medeiros et al. [66]. Traditional static analysis tools generally use the approach of parsing the code or extracting it to an abstract data structure. In contrast, DEKANT processes the code using the NLP method hidden Markov model. Using this model, the annotated large code corpus is used for training it, in the case of this paper, it is in PHP programming language. To improve the detection capabilities of the model, the authors first converted the large code segments into slices similar to code snippets in HyVulnDetect [5]. Next, the snippets are translated into an intermediate language which removes irrelevant lines of code and simplifies the rest using taint analysis vocabulary. As a result, the model has fewer tokens to classify and can be used cross-platform, as long as there are converters for its custom intermediate language. In the study, DEKANT found 4143 zero-day vulnerabilities across several open-source PHP plugins with great accuracy.

Thapa et al. [67] created a novel method for detecting vulnerabilities in C/C++ codebases using pre-trained transformers. Initially, they generate *Code Gadgets* from the target application. This process involves removing all comments and non-ASCII characters, normalizing the codebase, and destructuring functions with classes into a single serial execution. Library functions are unrolled in a similar way. The task of the model is a simple classification of whether the code gadget is vulnerable or not. The authors test the network using different transformers such as GPT, BERT, and RoBERTa. Additionally, they test using RNN based models including BiLSTM. The models pre-trained on natural language and source code are then finetuned. During finetuning, the models receive code gadgets and a classification head which allows for improved supervised learning. The study used two sets of vulnerabilities: buffer overflow and resource management. The paper reports the transformer-based models to outperform Convolutional Neural Network (CNN) models in false positive and false negative rates as well as the F1-score. Moreover, the results show that the F1-score was positively correlated with the size of the models.

Chapter 3

Methodology

The purpose of this chapter is to provide an overview of the research method used in this thesis. Section 3.1 describes the research process. Section 3.2 focuses on the data collection techniques used for this research. Section 3.3 depicts the experimental design. Section 3.4 explains the techniques used to evaluate the reliability and validity of the data collected. Section 3.5 demonstrates the method used for the data analysis. Finally, Section 3.6 defines the framework selected to evaluate the performance of the proposed method.

3.1 Research Process

This study follows the common research process that consists of exploration, experimentation, and evaluation. To begin, a thorough literature review was undertaken at the start of the project and was kept up to date throughout. During the process, cutting-edge transformer models and training datasets were evaluated for the proposed technique. Next, the experiment scope and methodology were designed based on the gaps found in the literature review. This process includes selecting the pre-trained model, setting up the training environment, and creating the experiment pipeline. Finally, the best-performing models are analyzed and compared using the assessment approach mentioned in Section 3.6. Post-analysis, we discuss the results of the experiment, and share our research learnings in the study, followed by future research directions. The overall process can be viewed in Figure 3.1

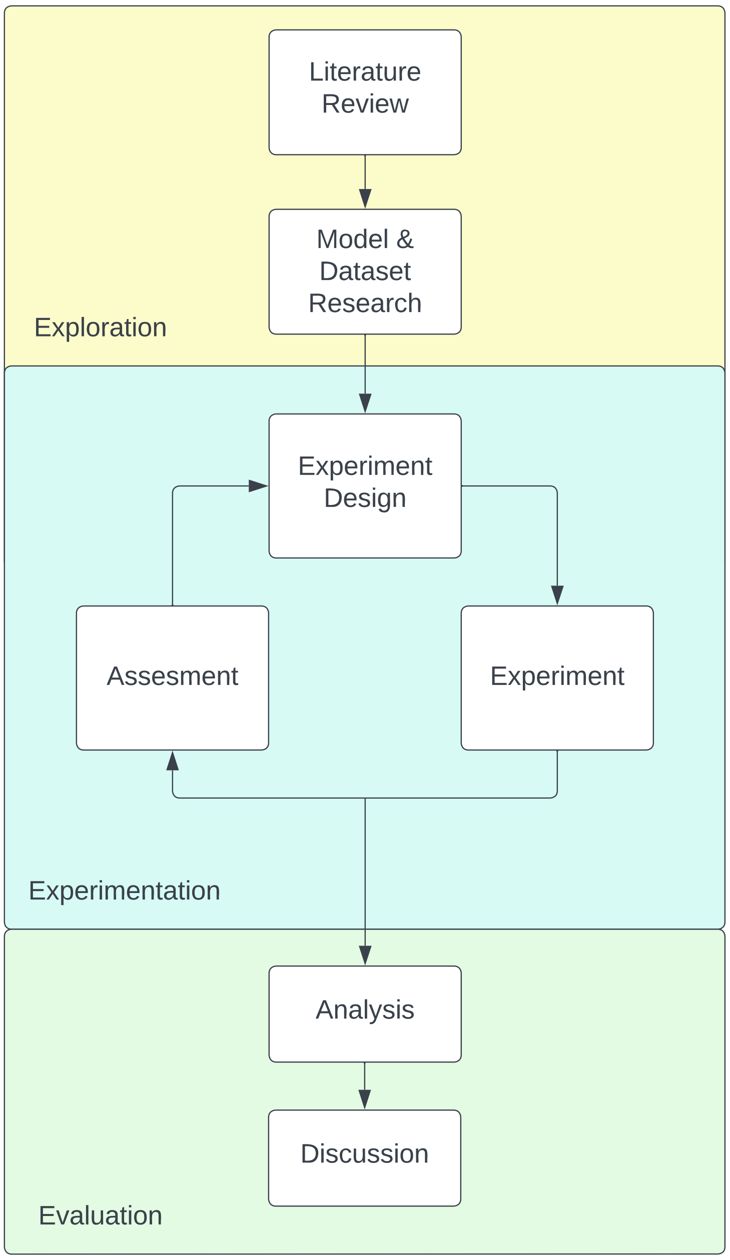


Figure 3.1: Research Process

3.2 Data Collection

Our research found a few public datasets of HyperText Transfer Protocol (HTTP) requests that were labeled with vulnerability identifiers, such as TORPEDA [68] and ECML/PKDD 2007 Discovery Challenge [69] datasets. However, after careful consideration, we concluded that our research would not benefit from them. These datasets consist of request data from network and application firewalls which are labeled as malicious/non-malicious or in some cases with CWE identifiers. Nevertheless, they do not include the HTTP responses to the request. With only request data, classification methods can only identify payload-based attacks. Vulnerabilities, such as CWE-684, require information from the response for identification. Lastly, none of them have labeled samples on our target vulnerabilities, making them candidates only for negative examples.

To rapidly create a new labeled dataset, we used a few different techniques. At first, we gathered vulnerability samples from OWASP and MITRE resources. These included demonstrative examples as well as documented real-world application vulnerabilities. We converted the request and response pairs to our minimized text format and prepended contextual information to fit our template. We also created negative samples based on the vulnerable examples with small changes to make the examples non-vulnerable. Next, we crafted a few-shot prompt that includes a task description, a template for the response, and a list of labeled examples for ICL. Using this text, we prompted the gpt3.5-turbo model by OpenAI. The underlying model is a GPT3-based LLM, finetuned for conversational prompting and optimized for latency. Trained with over multi-hundred billions of tokens of text, it received an 85 F1 score in a few-shot setting in CoQA benchmark [40]. In our data generation method, we used few-shot prompting combined with advanced prompt engineering techniques that contribute to ICL. The maximum sequence length allowed for input and output combined is 4096 tokens. This enables us to leverage longer prompts with increased context.

The quality of the prompt has a direct effect on the performance of the generative models. Thus, we followed an evolutionary approach in our prompting process. At each iteration of the instruction prompt, we evaluated the quality of the results and augmented the prompt until the desired quality was achieved. For the purpose of accelerating the process, we instructed the model to generate 10 samples. At first, our prompt included both negative and positive samples. After examining 500 generated samples, we could only identify 10 positive labels. In an attempt to increase the number, we separated the positive sample generation prompt, removing the negative label description and samples. With this prompt, the true positive sample generation rate doubled.

The next prompt included a short task description, a template for the response, and few-shot samples. It had 10% accuracy in generating vulnerable samples as it was instructed. Additionally, the generated samples often did not comply with the provided sample template. In the next iteration of the prompt, we extended the description of the task, highlighting the restrictions and improving the vulnerability definition. These changes yielded great improvement in accuracy and reduced the likeliness of malformatted samples.

A few more iterations of the instruct prompt were experimented with, altering variables including few-shot example count and grammatical changes to the task description. Evaluation of the quality of the output demonstrated minimal improvements. In the final version of the prompt, we included CoT for each few-shot sample, further increasing the accuracy of the output. This prompt can be examined in detail in Appendix B. The maximum achieved accuracy was 30% for the correctly generated samples. Unfortunately, the low rate of accuracy nullified the original plan of using the generated samples as is in the dataset. Consequently, we decided to manually review and relabel each sample that was generated to ensure the validity of our dataset and experiment.

The process of manually labeling was a slow task due to its complexity. It took on average 30 seconds to label one sample. To improve the efficiency of labeling, each sample was formatted with a script for easy-to-read multi-line paragraphs with keywords such as ”Request” highlighted. Since most of the examples the model provided were false positives, we decided to only review samples generated to be positive by the model. This approach lowered the class imbalance in the labeled dataset. We ended the labeling process once we had at least 200 labeled samples from each class. In total, 1780 samples were manually labeled.

The abnormal examples the model generated were missing context, request, or response, were malformatted, and in rare cases, it replied back with the prompt without changes. At this stage, we cleaned the data by removing the abnormal examples, concatenating data into a single file, and reformatting it to Comma Separated Values (CSV) format. We also fix examples that are out of order, spread across multiple lines, or have more tokens than the model can handle without truncation. The latter was difficult to correct since the token count would change based on the tokenizer. Thus, we set a sensible maximum length and filter unfitting samples. Finally, we split the dataset into smaller chunks for observing the effect of sample size on model performance.

3.2.1 Sampling

The dataset is sampled into multiple smaller sub-datasets using stratified sampling, except for the “XLarge” dataset as can be seen in Table 3.1. The datasets were designed for cross-validation, each fold will have an equal amount of classes if split into 10 folds. The “XLarge” dataset was created using the entire labeled dataset which had mostly negative samples. This dataset is also designed for 10-fold splits and each fold has an equal amount of labels.

Name	Total	Not Vulnerable	CWE-639	CWE-209
Small	150	50	50	50
Base	300	100	100	100
Large	600	200	200	200
XLarge	1780	1340	200	200

Table 3.1: Datasets and their properties

3.2.2 Target Population

The target population is a pair of vulnerabilities in the Broken Access Control category from the OWASP Top Ten list. CWE-639 Authorization Bypass Through User-Controlled Key is labeled as “1”. CWE-209: Generation of Error Message Containing Sensitive Information is labeled as “2”. Samples containing no vulnerability are labeled as “0”. No other vulnerability is sampled or classified.

3.3 Experimental Design

The experimental design is separated into two phases: data preparation, and model testing. The purpose of the first phase is to create the training dataset which includes all steps regarding data, including data gathering, cleaning, generation, and preparation. The second phase utilizes the datasets created from the previous stage and trains a transformer model for classification, which will be used for evaluation.

1. Collect vulnerability samples

2. Create few-shot prompts from samples

3. Use the prompts with GPT3 for dataset generation
4. Clean, label and prepare the training dataset
5. Testing and Experimenting finetuning LLMs using SetFit
6. Analysis

3.3.1 Test Environment Setup

The project repository includes the datasets, code for finetuning, and experiment scripts [1]. We provide a Dockerfile in Appendix A to set up the environment for the experiment and the finetuning of the LLM. Experiments can be run with the respective Python scripts. We also include scripts for generating the graphs used in the analysis.

3.3.2 Hardware and Software to be used

The study will use two separate environments for the experiments. A training environment running on the Google Colab [70] service is used for finetuning the models. This environment has 3 Graphical Processing Unit (GPU) options: NVIDIA T4, V100, and A100 which have 15GB, 15GB, and 40GB VRAM respectively. The second environment is used for inference experiments, which runs on a consumer-grade laptop with an RTX 3060 GPU with 6GB of VRAM, and an AMD Ryzen R7 5800H 8-core Central Processing Unit (CPU) with 16GB of RAM. As software, Python is used as the scripting language, and HuggingFace SetFit [50] library for finetuning and running the LLM model. SetFit library requires PyTorch deep learning library underneath, which should be installed before it [71]. The project uses open-source pre-trained transformer models hosted on the HuggingFace Hub [72]. At the beginning of finetuning scripts, this model will be downloaded to the local device. Thus, an Internet connection is required to begin the experiments.

3.4 Assessing Reliability and Validity of the Data Collected

3.4.1 Reliability of Method

The entire experiment can be launched within a containerized environment using Docker [73]. Thus, any researcher with an interest in repeating the

[1] `https://github.com/e1337us3r/KARTAL`

study can simply run the Dockerfile on supported platforms. Running the experiment in a container creates an identical environment to the one used in the paper, down to the OS level. Moreover, using the Dockerfile image will automatically install the required software within the container using the same version numbers as the experiments, resulting in a greatly consistent testing environment.

3.4.2 Reliability of Data

Although the finetuning process has a certain amount of inherent randomness, the results of the experiments we share in the paper are consistent and can be reliably recreated. At first, in order to have reliable and consistent results with each experiment trial, we remove all possible sources of randomness and noise. Our datasets are prepared for folds and shuffled in advance for finetuning. This ensures future experiments will use the same training and validation splits with an equal amount of classes. Another precaution we took was to preselect a seed value for when randomness is needed. Most pre-trained models come without a classification head which has to be initialized with random weights. Initializing model head weights in each experiment. We use the preselected seed value each time a new head has to be initialized for increased reliability in results.

3.4.3 Data Validity

Measurements of metrics for evaluation are done using the open-source "evaluate" library from HuggingFace. It is a well-established Python library that can take highly accurate measurements during the model evaluation step. The measurements are returned with high-precision floating point numbers, which we then round up for presentation. Furthermore, we use k-fold cross-validation and take the mean of k-folds for each individual metric which conveys a holistic view of performance, improving the validity of results.

3.5 Planned Experiments & Data Analysis

In order to assist further research, we aim to demonstrate the performance of the proposed technique with various parameter configurations. Demonstrating the effect of each variable will allow future experiments to focus only on the ones that matter for the overall performance of the technique. The variables we test variations of are the pre-trained model and dataset size. We also experiment with hyperparameters of the training model, however, analysis of them is out of scope for this study. Nevertheless, we share the settings we used for the training process for the repeatability and validity of our results.

3.5.1 Hyperparameters

Hyperparameters are the high-level variables that configure the model training process. They control the performance and behavior of the model. Selecting the best parameters is crucial for the research as suboptimal parameters can mislead the end results. Hyperparameters cannot be calculated and require experimentation to find the optimal configuration. Iterating over a large pool of parameter configurations and testing them for the best performance can be referred to in the literature as "hyperparameter tuning" or "hyperparameter optimization" [74]. In our study, we optimize for both the convergence speed and performance of the model. Due to limited resources, we had to limit the search space of the optimal hyperparameters. Additionally, we set the total experiment to count to 20 which saves time for other parts of the study. The final parameters that were used in the study can be seen in Table 3.2.

Parameter	Value	Description
Learning Rate	2e-6	Step size at which the model adjusts its parameters
Epoch	3	Number of times the entire dataset is passed through the model
Iteration Count	20	Number of text pairs to generate
Seed	25	Randomizer seed value to ensure reproducibility of the results
Warmup proportion	0.1	Proportion of training steps dedicated to gradually increasing the learning rate
Batch Size	32	Number of training examples processed in a single iteration

Table 3.2: Hyperparameter configuration used for training

3.5.2 Pre-trained Models

The performance of the finetuned model has a direct relationship with the performance of the pre-trained model. Model authors generally publish the performance of the model on various benchmarks making choosing the pre-trained model with the best performance relatively simple. Hardware requirements of finetuning and running inference changes with each model. The model properties that affect the VRAM requirements are sequence length, number of layers, and vocabulary size. The overall scale of the model also affects the inference latency. In order to find the optimum model with maximum performance and minimum latency, we experiment with the top 3 best-performing sentence transformers that are available on the HuggingFace platform.

Model Name	Performance	Encoding Speed	Seq. Length
all-mpnet-base-v2	63.30	2800	384
all-distilroberta-v1	59.84	4000	512
all-MiniLM-L12-v2	59.76	7500	256

Table 3.3: Pre-trained model properties

We highlight three important features of pre-trained models in Table 3.3. Performance is the average score of 14 distinct tasks from various disciplines in encoding sentences, and semantic search [75]. The encoding speed is the number of sentences an NVIDIA V100 GPU can encode per second. Lastly, the sequence length is the maximum number of tokens the model can process, including the number of tokens in the output. The all-mpnet-base-v2 model has the best overall score, however, it is also the slowest-to-run model. In contrast, all-MiniLM-L12-v2 is the fastest model at the cost of performance and 1/3 shorter sequence length. Meanwhile, all-distilroberta-v1 has the all-round performer, positioning itself in the middle of both LLMs.

3.5.3 Training Dataset Size

The size of the dataset used in the training can have an impact on the performance of the model. The potential effect is amplified when the dataset is smaller in size. In order to demonstrate the effect of data size on the performance metrics of the trained model, we created 4 distinct datasets listed in Table 3.1. The first three datasets have an equal amount of data in each label category with only varying in overall size. The final and largest dataset is unbalanced, with most of the labels in the non-vulnerable data category. The purpose of creating this dataset was to test whether increasing labeled data in the non-vulnerable category would result in lower false positives in the classification phase. We conduct this experiment exclusively using the all-MiniLM-L12-v2 model as it the fastest model to finetune.

3.5.4 Inference Performance

The average time to make a prediction is an important metric for evaluating the performance of an LLM. In cases where the inference latency is excessive, synchronous applications of the model are severely reduced. A fast model could enable real-time detection of vulnerabilities during development which can greatly increase the security of the web application. We measure the latency of our finetuned LLM by running the prediction pipeline with 1000 randomized samples. The experiment is conducted on a laptop to test the feasibility of the real-time detection use case, the hardware specifications of which were detailed in Section 3.3.2. Additionally, we run the experiment with and without GPU acceleration to get a comprehensive benchmark of the performance of the model. The test is run 10 times and the average, minimum, and maximum inference latency is recorded.

3.6 Evaluation Framework

Since we formulate the problem as a classification, we construct a confusion matrix and evaluate the key metrics using it. In a classification issue, a confusion matrix is a table that indicates the number of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN) for each class. The columns of the matrix indicate the expected class labels, whereas the rows represent the actual class labels. The confusion matrix is an effective tool for assessing classification model performance. It enables us to examine not only how well the model performs overall, but also how well it performs for each class and where mistakes occur. We calculate numerous metrics by studying the values in the confusion matrix, which will help us understand the model's strengths and flaws and compare them to other models.

To evaluate the performance of the proposed method we use cross-validation. A popular method in ML for assessing the effectiveness of prediction models is the k-fold cross-validation. The data is divided into K folds of equal size using a random number generator before k-fold cross-validation is performed. The remaining K-1 folds are then utilized for training the model, with one of the K folds serving as the validation set in each iteration. On the training set, the model is developed, and the validation set is used to assess its performance. Each of the K folds is utilized precisely once as the validation set, and the procedure is repeated K times. Ultimately, an overall estimate of the model's performance is generated by averaging the outcomes of each iteration. The dataset size and required degree of precision for the performance estimate influence the choice of K, which in our study is 10.

Compared to conventional hold-out approaches, k-fold cross-validation has the main benefit of enabling a more thorough examination of the model's performance. It can give a more reliable assessment of the model's performance by training and testing it on various subsets of the data, lessening the influence of any peculiarities or random fluctuations in the data. By assessing the model's performance on the validation set in each iteration, k-fold cross-validation may also be used to finetune model hyperparameters such as the number of hidden layers in a neural network.

One of the key benefits of this validation method is the full utilization of the dataset when training and testing, which is crucial when using a smaller dataset. This method provides for a more complete assessment of the model's capacity to generalize to new data. Nevertheless, training and testing the model K times is a time and resource-intensive operation. Moreover, datasets with unbalanced classes or skewed distributions may not be acceptable for k-fold cross-validation, necessitating alternate assessment.

To overcome this issue we use stratified sampling. It is a sampling technique used in statistics and ML to ensure that the distribution of a target variable is well-represented in the sample data. The technique involves dividing the population into subgroups or strata based on the target variable, and then randomly sampling from each stratum in proportion to its size. This ensures that each stratum is represented in the sample and that the sample distribution is similar to the population distribution.

3.6.1 Evaluation Metrics

The evaluation metrics we utilize are frequently employed in binary classification issues and offer a thorough assessment of ML models. We use *macro* averaging strategy to jointly assess all label classes. In situations where incorrect positive predictions have a significant cost, *precision* estimates the fraction of accurate positive forecasts among all positive predictions. The fraction of true positives that are accurately detected is measured by *recall*, which is a crucial metric when false negative predictions are costly. Both of these metrics are intermediate values and do not represent well the overall performance of the model. While accuracy and recall are simple to perceive and comprehend, they might be deceptive if the dataset is unbalanced or the cost of false positives and false negatives is not equal. The F1-score is an excellent statistic to utilize in this case since it is the harmonic mean of accuracy and recall.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3.4}$$

$$MCC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}} \tag{3.5}$$

It is crucial to have a metric representing the overall performance of the model including the negative statistics, especially in situations when erroneous optimistic forecasts might have serious repercussions. Thus, we chose the Matthews Correlation Coefficient (MCC) as the metric to optimize for. MCC considers both true and false positives and negatives and calculates the correlation between predicted and real labels [76]. Unlike other assessment measures such as accuracy, which can be deceptive in unbalanced datasets, MCC considers the distribution of positive and negative cases in the dataset. Another advantage of utilizing MCC is that it produces a single scalar number that summarizes the model's performance, which makes comparing the performance of multiple models easier. Moreover, changes in the decision threshold have no effect on MCC, which might be a concern for assessment metrics like accuracy and recall.

Chapter 4

Implementation

In this chapter, we present the implementation details of each stage of the experiment.
Our novel approach KARTAL consists of multiple components that have different functions. The main components are the Fuzzer, Prompter, and LLM, which are employed in the respective order. The classification flow continually processes information from the target API until a prediction is made. The Fuzzer component sends various HTTP requests to the API with different levels of user privileges. The requests and response pairs are then compressed and stored. Next, the Prompter component chains the saved pairs into a single prompt together with the API context and prompts the finetuned LLM. Finally, the LLM makes a classification based on the information in the prompt and prints the predicted label.

Figure 4.1 illustrates the architecture of the proposed method. The components of KARTAL are yellow color coded. The dotted lines represent optional data flows. As can be seen from the diagram, the approach is simple and flexible. Additional components can be added to the system without requiring re-finetuning the model. Similarly, the model can be replaced with minimal changes to the system.

4.1 Fuzzer

The first component in the flow of data is the Fuzzer. In general, fuzzers are automated testing tools designed to analyze the security and resilience of software applications. Their function is to systematically produce and insert erroneous, unexpected, or malformed data into applications such as APIs. When done at scale, they can discover otherwise difficult-to-spot weaknesses in code. There are many types of fuzzers such as basic, intelligent, and

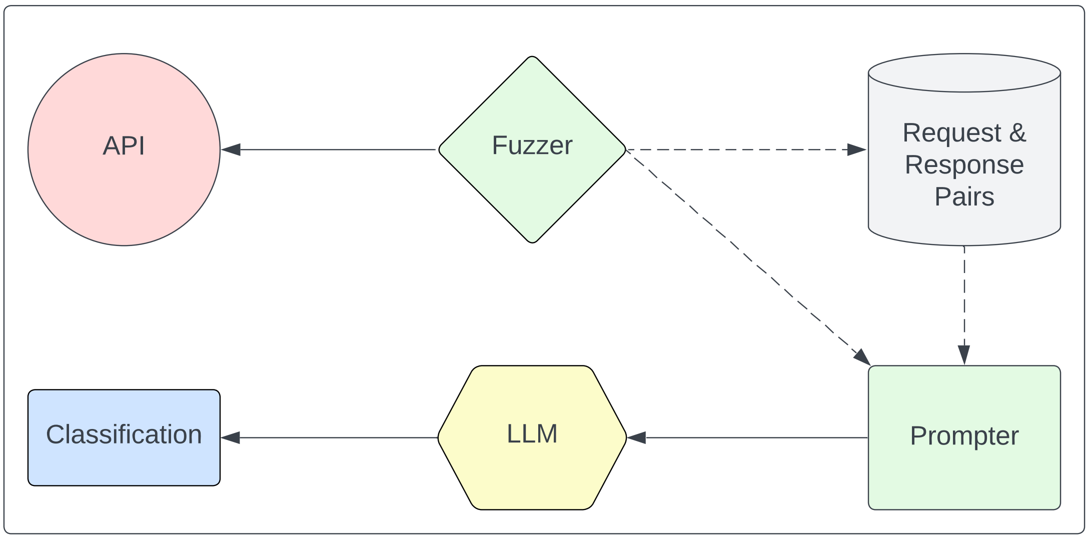


Figure 4.1: Proposed method architecture

protocol based. Figure 4.2 presents the process of the Fuzzer component in KARTAL.

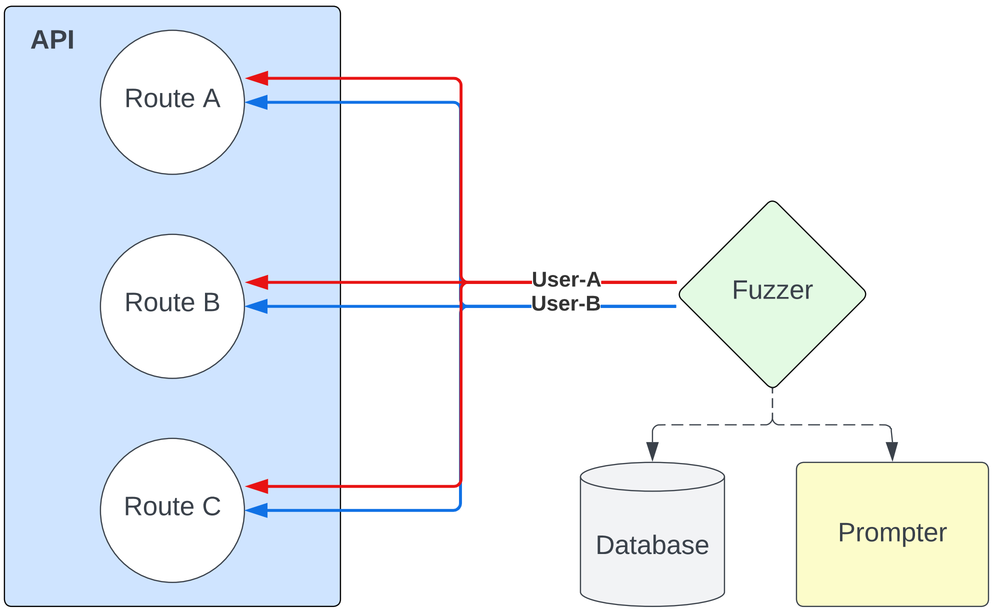


Figure 4.2: Fuzzer component

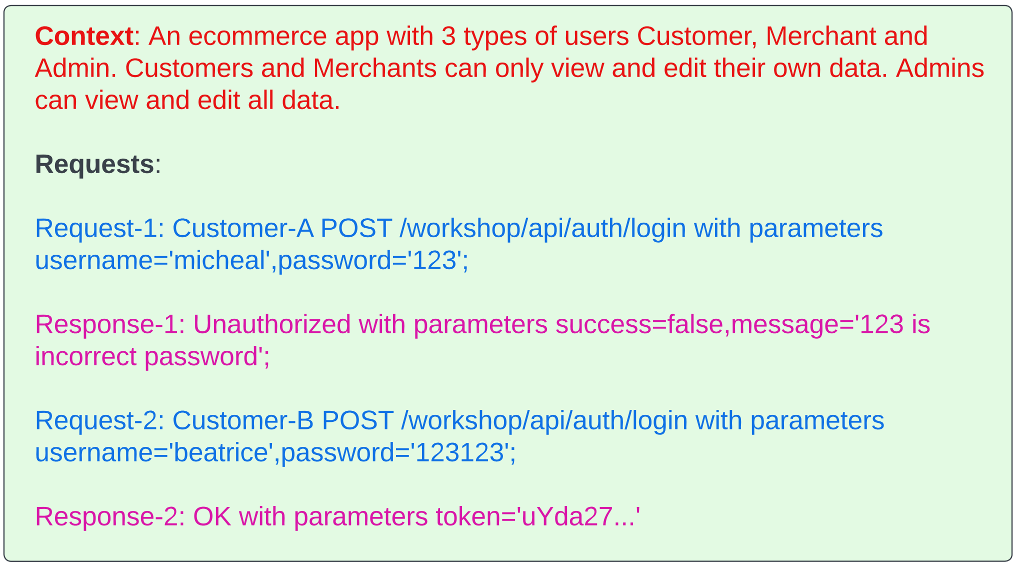
The ideal Fuzzer implementation for KARTAL combines generation and mutation-based approaches. Mutation-based fuzzers alter existing valid inputs using mutations such as bit flips, random insertions, deletions, or rearrangements. Meanwhile, generation-based fuzzers generate inputs from scratch while conforming to the API's structure and limitations. They can produce legitimate and semantically relevant inputs using grammar-based or model-based methodologies, which can be useful for testing complicated APIs. Broken access control flaws often require testing various levels of authorization. As such, the Fuzzer component must methodologically craft and mutate inputs with different user groups and privileges.

The responses from the requests that the Fuzzer sends are paired together. Next, the results from the Fuzzer are transferred. The dotted lines in Figure 4.2 represent alternative ways this transfer can be done. The first method is to directly send the resulting fuzz pairs in batches to the Prompter. The advantage of this approach is low complexity. However, this method suffers from lower chances of identifying vulnerabilities since the random sampling of batches may couple irrelevant data together. A more effective approach is to store the request and response pairs from which they can be intelligently grouped for the Prompter. For instance, the grouping can simply be done based on the URI path or resource. KARTAL does not include a Fuzzer, users are free to choose open-source source fuzzers or create a custom implementation.

4.2 Prompter

The purpose of the Prompter component is to combine the list of HTTP request and response pairs and application context into a single coherent text to prompt the LLM for classification. Figure 4.3 displays an example of a prompt created by the Prompter component. The structure of the prompt is comprised of 2 parts. The *Context* describes the target application, lists the names of all entities, and defines the roles of each of them. This information provides the LLM with the constraints of the application as well as insights into the domain. For instance, in a social media application, the users are expected to have read-only access to posts by other users. Unless specified otherwise, the model will not mark such interactions as vulnerable. This approach benefits from the ICL capabilities of the model. The *Context* provides dynamic properties to KARTAL, allowing it to adapt to different environments and restrictions during inference.

The second part of the prompt consists of a list of HTTP request and response pairs. The pairs can either directly be fed in from the Fuzzer or queried from storage, as can be viewed in Figure 4.1. Depending on the type


Context: An ecommerce app with 3 types of users Customer, Merchant and Admin. Customers and Merchants can only view and edit their own data. Admins can view and edit all data.

Requests:

Request-1: Customer-A POST /workshop/api/auth/login with parameters username='micheal',password='123';

Response-1: Unauthorized with parameters success=false,message='123 is incorrect password';

Request-2: Customer-B POST /workshop/api/auth/login with parameters username='beatrice',password='123123';

Response-2: OK with parameters token='uYda27...'

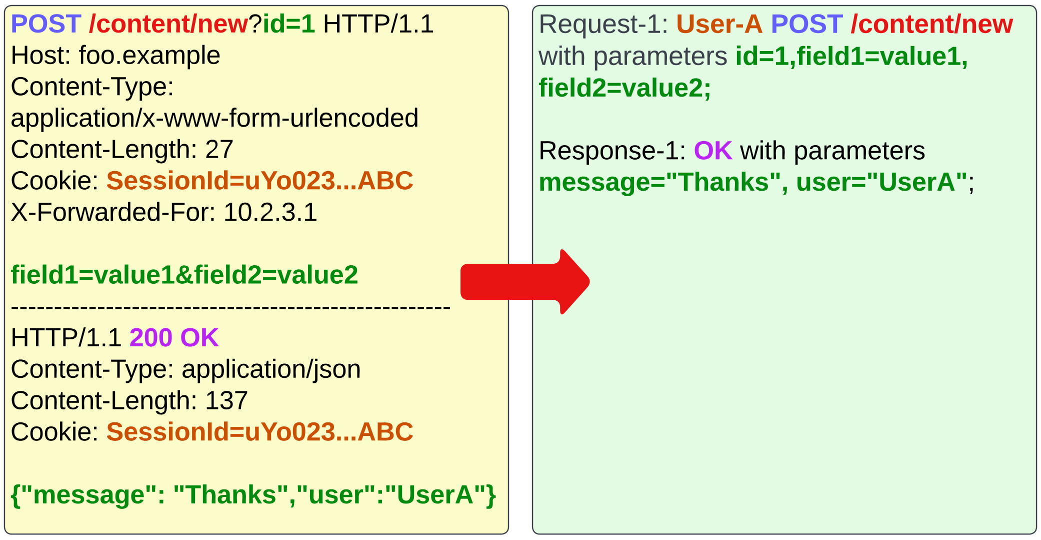
Figure 4.3: An example Prompt ready for classification

of vulnerability, the selection of the request and response pairs can be important. In such cases, querying the pairs based on the URI path, object or entity can improve the vulnerability detection rate of the model. Nevertheless, users who implement a custom query mechanism should pay attention not to over-constrain the search space of pairs, leading to impairment of the accuracy of the model.

4.2.1 HTTP Request and Response Compression

A major limitation of transformer models is the sequence length. As mentioned in Section 2.1.1.3, the hardware requirements for running the model scale quadratically with the number of tokens in input and output. Table 3.3 displays the maximum sequence length allowed by each pre-trained model. Given this limitation, using the full HTTP request/response would reduce the amount of context that can fit in a single prompt. To address this issue, we use compression. Our method of compression has 4 stages. First, we omit fields that are not relevant to the classification such as the “Content-Length” header and the HTTP protocol version. Next, we deserialize the query and body parameters into a single stream of text, removing markup and additional syntax of formats, such as XML and JSON. The third stage trims long blocks of text such as the contents of a blog post. Additionally, we assign short identifiers to the randomly generated identifiers that objects may use in formats including UUIDv4.

Finally, we extract and convert the credentials in the HTTP request into the authorized entity. These credentials are commonly stored in the “Cookie” or “Authorization” header and sent with each request by the browser or the HTTP client. Mapping credentials to an entity can simply be done by assigning a random letter to each one. This approach will treat all users to have the same level of privilege. However, entity-privilege information is required to detect broken access control vulnerabilities that involve privilege escalation.



```
POST /content/new?id=1 HTTP/1.1
Host: foo.example
Content-Type:
application/x-www-form-urlencoded
Content-Length: 27
Cookie: SessionId=uYo023...ABC
X-Forwarded-For: 10.2.3.1

field1=value1&field2=value2
------------------------------------------------
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 137
Cookie: SessionId=uYo023...ABC

{"message": "Thanks","user":"UserA"}
```

```
Request-1: User-A POST /content/new
with parameters id=1,field1=value1,
field2=value2;

Response-1: OK with parameters
message="Thanks", user="UserA";
```

Figure 4.4: HTTP compression example

Utilizing compression significantly reduces the token count for each HTTP request and response. In the example shown in Figure 4.4, the token count drops from 130 to 48, a 63.1% reduction. The savings can reach 90% depending on the request/response headers or content, further increasing the amount of context that can be fit in a single prompt. It should be noted that our implementation of the HTTP compressor is customized for the specific vulnerabilities we aim to classify. Other types of vulnerabilities may require some of the fields we omit or truncate. Users should implement a compressor that is customized to their requirements based on the techniques we provide.

4.3 Model Finetuning

NLP and ML studies have become much more accessible in light of user-friendly ML libraries, such as SetFit by HuggingFace [50]. These libraries offer algorithms, tooling, and case-specific recipes that may be quickly applied to a variety of NLP applications by researchers, developers, and even non-technical users. In particular, HuggingFace's ecosystem of libraries stands out as a thorough and easy-to-use package that lower the barrier to creating cutting-edge AI models. HuggingFace hosts a variety of pre-trained models and offers an easy interface for optimizing these models for certain tasks. The open-source library and tooling together with hosted models save time, funds, and human resources, fundamentally enabling the research to be completed.

The SetFit library provides a consistent user interface for many models, users can transition between models without having to make significant code alterations. This feature allowed us to test our method on different LLM architectures within the time constraints of the study. The library is integrated with deep learning frameworks including PyTorch and TensorFlow [71, 77], enabling users to choose the ML backend and benefit from GPU or Tensor Processing Unit (TPU) acceleration for effective training and inference. SetFit provides simple interfaces for tokenization, model setup, hyperparameter optimization, and training.

The overall process of finetuning can be described in 3 sections: dataset preparation, initializing evaluation, and creating the training loop. First, we select and load the dataset from the CSV file. We use the Datasets [72] library for this purpose which comes with a few key benefits. It can load datasets locally or from the Internet, in multiple text formats, and simultaneously split the dataset into different parts. Using this feature, we create 10 folds and split the dataset into training and validation sets. The first fold of the validation set is the first 10% of the dataset, while the rest is part of the training set. Then the second 10% becomes the validation set with the rest becoming the training set. The process repeats until all 10 folds are created.

Next, we use Evaluate and Sklearn libraries to create our evaluation function. Evaluate offers out-of-the-box capabilities for calculating common metrics such as accuracy and recall. In order to generate the confusion matrix, we use the metrics module of the Sklearn library. Combining all together, we create a single function that takes model predictions and the ground truths and returns a dictionary object with the metrics we presented in Section 3.6. This function runs after the target number of epochs has been completed for each fold. We also create a separate function that calculates the mean of each metric for all folds.

In the last phase, we create the training and evaluation loop. For each fold, we load a new copy of the pre-trained model. The model is only downloaded for the first fold and will be retrieved from the cache for subsequent iterations. Next, we create our model trainer with the dataset of the fold, and the hyperparameters listed in Section 3.2. Finally, we start the training process. Initially, the library generates the contrastive training pairs and then begins to iterate over each. Batching assists in accelerating the iteration by processing multiple pairs together.

For maximum performance, we use end-to-end training. End-to-end training requires updating the weights of the model and classification head simultaneously while it is being trained. With this method, the model as a whole may incorporate both low-level and high-level properties to adapt to the particular job. Head-only training, on the other hand, freezes the model weights and updates only the classification head. This approach accelerates the training process significantly. However, the performance of head-only training depends heavily on the dataset of the pre-trained model. In some cases, this will lead to subpar performance if the trained task does not appear in the pre-trained model's dataset. While computationally more expensive, end-to-end training gives the model the chance to fine-tune all of its parameters, improving the learning performance of both previously seen and unexplored tasks.

Once the training is completed for a fold, we evaluate against the validation dataset and save the fine-tuned model to disk for future usage.

4.4 Inference

The main output of the fine-tuning stage is the trained model. The saving functionality provided by the library creates a folder on disk that includes updated model weights, configuration, and classification head. For inference, we load the fine-tuned model by providing the path to this folder, which can also be located on the Internet. Once loaded, we can prompt the model for classification. This can be done in two ways, either by providing a list of strings, each of which represents a prompt, directly to the model. The output will be a list containing predicted labels for each prompt in the same order as the input list. The second method uses the prediction probe of the model which outputs a list of probabilities for each class label instead of predicting a single label. The inference can be run on GPU or CPU, which can be switched at runtime. It should be noted, that the hardware requirements of inference are considerably lower than the training process.

Chapter 5

Results and Analysis

This chapter provides a comprehensive analysis of the experimental results obtained from our study. In Section 5.1, we present the evaluation of fine-tuning different pre-trained base models along with a comparative analysis of their performance. Next, in Section 5.2, we delve into the results obtained by varying the size of the training dataset and assess its impact on the performance of the model. In Section 5.2.1, where we analyze misclassifications made by the models. Lastly, in Section 5.3, we evaluate and present the efficiency of the model in terms of their inference time and resource requirements. Throughout this analysis, we refer to the dataset introduced in Section 3.2 and utilize the evaluation metrics discussed in Section 3.6.1 to report our findings.

5.1 Pre-trained Model

As described in Section 3.5.2, we experiment with finetuning different classes of pre-trained transformers to identify the most optimal base model. After successfully training all 3 models, we compile all of the evaluation metrics into a single table in Figure 5.1. The values in the table are the mean of the results of the 10-fold cross-validation. We sort the table based on descending MCC value.

At first glance, we can see that finetuned all-mpnet-base-v2 model outperforms other finetuned models in all metrics except for accuracy. The finetuned all-distilroberta-v1 achieves an accuracy of 87.53%. Its accuracy slightly surpasses the accuracy of finetuned all-mpnet-base-v2 by 0.39%, and all-MiniLM-L12-v2 by 1.77%. In recall evaluation, the finetuned all-mpnet-base-v2 model is again the best performer with 83.36% recall. Although the finetuned all-distilroberta-v1 had the best accuracy, it ranks second in recall with 80.86%, a 3.09% difference. By contrast, recall is the worst-performing metric for all-MiniLM-L12-v2. It only manages to achieve 75.98% recall, 6.42% less than the second best, and 9.71% than the best score.

In precision measurements, the finetuned all-mpnet-base-v2 once again exceeds the scores of other finetuned models. It scores 82.20% precision, a modest improvement of 0.72% compared to all-distilroberta-v1 which scored 81.61%. The precision of the finetuned all-MiniLM-L12-v2 model ranks last with 77%, a clear difference of 6.75% compared to all-mpnet-base-v2, and 5.99% compared to all-distilroberta-v1.

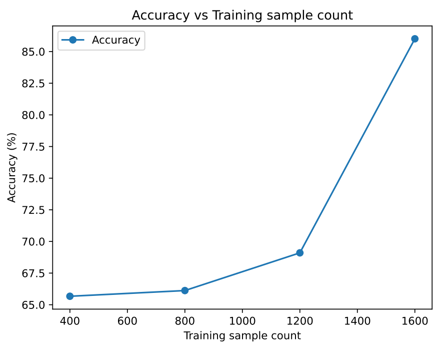
Table 5.1: Evaluation of the fine-tuned model with different base models

Base model	Accuracy	Recall	Precision	F1	MCC
all-mpnet-base-v2	87.19%	83.36%	82.20%	0.82	0.70
all-distilroberta-v1	87.53%	80.86%	81.61%	0.80	0.69
all-MiniLM-L12-v2	86.01%	75.98%	77.00%	0.76	0.63

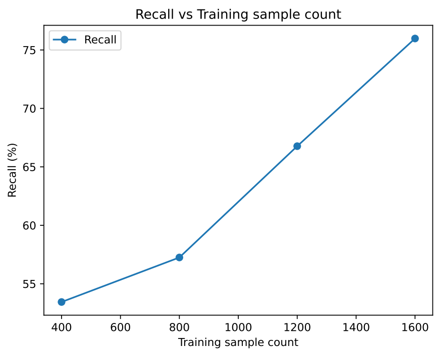
For well-rounded metrics including F1 and MCC, the rankings from other metrics do not change. Nevertheless, the differences between each score dwindle. The finetuned all-mpnet-base-v2 model scored 0.7 MCC, followed by 0.69 MCC of all-distilroberta-v1, and finally all-MiniLM-L12-v2 model with 0.63 MCC score. Finetuned all-mpnet-base-v2 demonstrates 1.45% improvement compared to the finetuned all-distilroberta-v1, and, 11.11% compared to all-MiniLM-L12-v2. The F1 scores are similarly close. The finetuned all-mpnet-base-v2 model scores 0.82 which is 2.5% higher than finetuned all-distilroberta-v1, and 7.29% higher than finetuned all-MiniLM-L12-v2.

5.2 Training Dataset Size

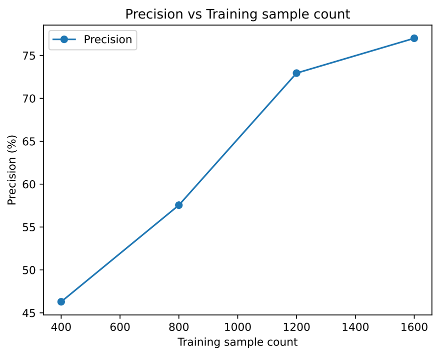
Generating and manually labeling data was one of the most time-consuming tasks in the study. In this Section, we examine the relationship between training sample size and evaluation metrics. Figure 5.1 illustrates the change in each metric at training sample sizes of 400, 800, 1200, and 1600. We perform k-fold cross-validation and take the mean of 10 folds. The samples are taken from the "XLarge" dataset and represent the first 25, 50, 75, and 100 percent of the samples in each fold. We selected all-MiniLM-L12-v2 as the pre-trained model to finetune for this experiment due to its smaller footprint and training speed.



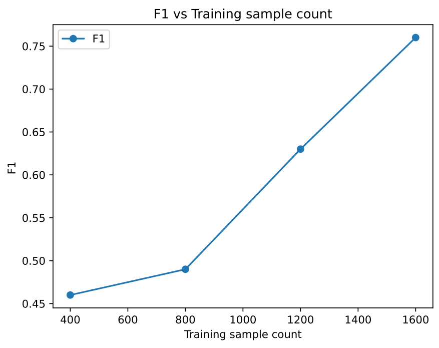
(a)



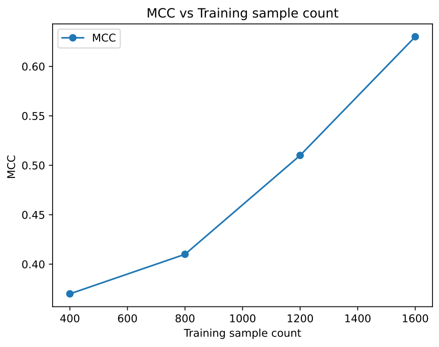
(b)



(c)



(d)



(e)

Figure 5.1: Evaluation Metrics vs Training Sample Count. (a) Accuracy, (b) Recall, (c) Precision, (d) F1, and (e) Matthews Correlation Coefficient (MCC) plotted against the training sample count. The evaluation metric for each graph shows how it changes as the quantity of training samples increases.

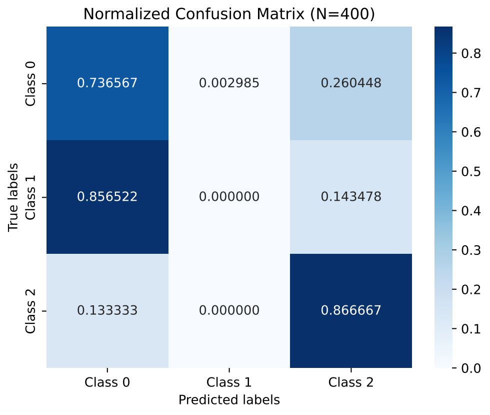
The accuracy of the model has a positive relation with training sample count as demonstrated in Figure 5.1a. However, the effect of sample size on accuracy is not evident at first. By doubling the training sample size from 400 to 800 samples, the accuracy increases by only 0.69%. Next, we increase the count by 50% from 800 to 1200 samples. During this training round, we see a 4.51% improvement in accuracy. Finally, we increase the training sample count by another 1/3 from 1200 to 1600. The boost in accuracy is a remarkable 24.64%, far surpassing previous gains from previous steps.

The recall values of the model exhibit intriguing tendencies in relation to the training sample count. As seen in Figure 5.1b, there is a significant rise in recall as the sample size grows. When the sample size is increased from 400 to 800, the recall improves by 3.81 points, which is a 7.13% improvement. Continuing the pattern, increasing the sample count by 50% to 1200 yields a recall value of 66.77, showing a significant 16.63% improvement over the previous step. Finally, a 1/3 increase to a sample size of 1600 causes the trend to lose steam, only improving recall by 13.79%.

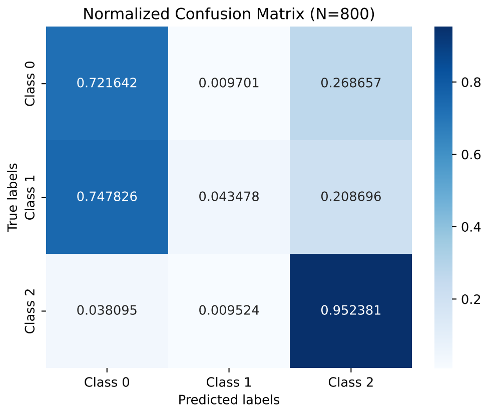
The precision of the model has a varied pattern with increasing sample size, as shown in Figure 5.1c. With a sample size of 400, the precision value is initially a modest 46.29%. When the sample count is increased to 800, the precision increases by 24.32%, culminating in a precision value of 57.55%. This increase reflects the favorable influence of a bigger training sample on model precision. Increasing the sample count by 50% to 1200 results in a precision value of 72.94%, a substantial improvement of 26.74%. However, when the sample size is further expanded by 1/3 to 1600, the precision growth slows, with an incremental gain of 5.57%, ending with a precision value of 77%.

As seen in Figure 5.1d, increasing sample sizes result in a significant rise in F1 score similar to the recall curve in Figure 5.1b. The initial F1 value is 0.46 for a sample size of 400. However, when the sample size is increased to 800, the F1 score improves by 6.52%, resulting in an F1 value of 0.49. Continuing the pattern, increasing the sample count by 50% to 1200 results in a significant 28.57% rise in the F1 score, providing an F1 score of 0.63. This huge improvement emphasizes the need of having a large enough sample size to get better F1 scores. Finally, with a 1/3 increase in sample size to 1600, the F1 score rises at a slightly slower rate. The incremental increase in F1 value is 20.63%, yielding a final F1 score of 0.76.

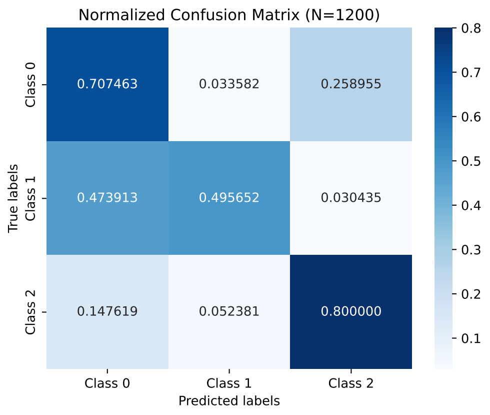
The MCC curve in Figure 5.1e is a more rounded version of the F1 curve in Figure 5.1d. This similarity is the result of both equations sharing the purpose of providing well-balanced evaluation metrics for classification. Upon doubling the initial sample count, the MCC score improves by 10.81%, resulting in an MCC value of 0.41. In a resembling trend, when the sample count is raised to 1200, the MCC score boosts by 24.39%, resulting in an MCC value of 0.51. This significant rise emphasizes the need for a sufficient sample size in generating better MCC scores, analogous to the results of other metrics. The upward trend begins to decelerate in the largest sample size, improvement rate descending to 23.53% and resulting in an end MCC score of 0.63.



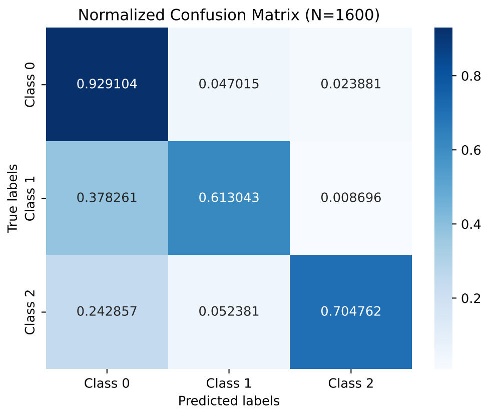
(a)



(b)



(c)



(d)

Figure 5.2: Normalized confusion matrices for four different sample sizes: (a) 400 samples, (b) 800 samples, (c) 1200 samples, and (d) 1600 samples. Each matrix shows how well a model performs classification, with the results adjusted to show the proportions of true positives, false positives, false negatives, and true negatives.

Examining the confusion matrix for each sample size provides a better understanding of differences in evaluation metrics. Figure 5.2 uses heatmaps to visualize the confusion matrices. Since we have 3 distinct classes, the resulting matrices are 3 by 3. Per the legend on the right-hand side of each matrix, darker shades of blue implicate higher density. For better interpretability, the matrices are normalized. Inspecting the matrices in ascending order of training sample size we can observe a clear trend. As the sample size increases together with TP and TN while FP and FN predictions decrease. This trend was visible from F1 and MCC scores in Figure 5.1, however, they did not show which classes were specifically under or overperforming. Figure 5.2a is the confusion matrix of the model that was finetuned with only 400 samples. From the values of the matrix, it is clear that classes 0 and 2 are classified with moderate accuracy.

In contrast, the model has failed to identify a single TP in class 1. As the sample size increases, the accuracy of identifying this class increases together. The major leap in accuracy occurs in Figure 5.2c from 800 to 1200 training samples, where the ratio of TP for class 1 improves by greater than 10 times. The classification performance for class 2 reaches its highest in Figure 5.2b with 800 training samples, then begins to diminish. Meanwhile, the rate of incorrectly classifying samples as class 2 steadily decreases. Throughout the experiment, class 0 sustains a slight decline in accuracy until Figure 5.2d with 1600 samples, where it outperforms all of the other classes.

5.2.1 Examples of Incorrect Classifications

In order to improve the accuracy of the model, the cases where it fails must be examined. On certain occasions, these can be edge cases that are unlikely to appear in real data. Furthermore, samples that are equidistant to two different classes can considerably increase regression loss and reduce convergence speed during training.

To exhibit incorrectly classified samples, we used the same finetuned model from Figure 5.2d and gathered examples. In Listings 5.1 and 5.2, we have samples from two different classes that have been falsely classified as negative. In the first case, the sample shows CWE-639 vulnerability because in Request-3 Driver-A successfully accepts a ride that does not belong to him which is against the rules defined in the context.

Listing 5.1: A false negative example for CWE-639 class, true label = 1, predicted label = 0

```
Context: A ride-sharing app where drivers can only
    accept their own ride requests.
Requests: [

Request-1: Driver-A GET /api/home with parameters
    userId='Driver-A';
Response-1: OK with parameters userId='Driver-A',
    rideRequests=['Ride-A'];
Request-2: Driver-A POST /api/rides/accept with
    parameters rideId='Ride-A';
Response-2: OK with parameters status='accepted',
    driverId='Driver-A';
Request-3: Driver-A POST /api/rides/accept with
    parameters rideId='Ride-B';
Response-3: OK with parameters status='accepted',
    driverId='Driver-A']
```

Listing 5.2: A false negative example for CWE-209 class, true label = 2, predicted label = 0

```
Context: A secure messaging app for financial companies
    , where senders can securely send sensitive data to
    verified receivers.
Requests: [

Request-1: Sender-A POST /secure-message with
    parameters receiver='receiver-email-123',message='
    sensitive client account information';
Response-1: Unauthorized with parameters msg='Receiver
    not verified, please ask them to create an account
    on the system']
```

The second false negative case is harder to identify. In response to Request-1 which tries to send a message to an unknown receiver, the app acknowledges the existence of the account with the email sent by the user as well as leaks the verification status of the account. This flaw can be abused by a malicious user to enumerate accounts of high-value customers which can be combined with another flaw to potentially lead to getting full account access. Meanwhile, Listing 5.3 is an example from our dataset that was incorrectly identified as a positive example from class 1. The app accepts all requests for objects by their respective owners while rejecting those who request resources that they are not authorized to access.

Listing 5.3: A false positive example for non-vulnerable class, true label = 0, predicted label = 1

```
Context: An online shopping app, where each customer
   can only access and modify their own cart.
Requests: [


Request-1: Customer-A GET /api/cart with parameters
   cartID='Customer-A';
Response-1: OK with parameters cartID='Customer-A',
   items=[{'name':'product-1','price':10},{'name':'
   product-2','price':20}];
Request-2: Customer-B GET /api/cart with parameters
   cartID='Customer-A';
Response-2: Unauthorized with parameters errorCode=401,
   errorMessage='You␣are␣not␣allowed␣to␣perform␣this␣
   action.',errorDetails='The␣cart␣id␣is␣wrong␣or␣not␣
   authorized␣to␣view.';
Request-3: Customer-A PUT /api/cart with parameters
   cartID='Customer-A',items=[{'name':'product-1','
   price':10},{'name':'product-2','price':20},{'name':'
   product-3','price':30}];
Response-3: OK with parameters cartID='Customer-A',
   items=[{'name':'product-1','price':10},{'name':'
   product-2','price':20},{'name':'product-3','price'
   :30}]
```

5.3 Inference Performance

The size, and complexity of the finetuned pre-trained model determine the inference performance. Per simple logic, the larger the model is, the slower it will run. We load the finetuned models from Section 5.1 and measure the latency for predicting 1000 labels. To simulate the environment of a researcher or developer, we run our experiments on a consumer-grade laptop with specifications listed in 3.3.2. We test both GPU and CPU inference performance which can be used to estimate the cloud computing costs of running the model. In order to improve the validity of our measurements, we repeat the test 10 times and calculate the mean, min, max, and std values for each model. The models in Table 5.2 are sorted on the avg column descendingly.

Table 5.2: System latency for predicting 1000 labels in seconds

	GPU			
Base model	avg	min	max	std
all-mpnet-base-v2	8.699	8.436	10.171	0.524
all-distilroberta-v1	3.968	3.766	5.615	0.579
all-MiniLM-L12-v2	1.854	1.556	4.399	0.894
	CPU			
Base model	avg	min	max	std
all-mpnet-base-v2	314.550	312.722	317.463	1.651
all-distilroberta-v1	144.266	143.530	144.790	0.394
all-MiniLM-L12-v2	60.624	59.001	64.686	1.829

In direct relationship with the sizes of the pre-trained models, the slowest-to-run model is all-mpnet-base-v2 followed by all-distilroberta-v1 and all-MiniLM-L12-v2. When GPU inference latencies are compared, the finetuned all-MiniLM-L12-v2 is 2.14 and 4.69 times faster than all-distilroberta-v1 and all-mpnet-base-v2 respectively. In CPU inference, the order of slowest to fastest pre-trained models does not change. However, all of the models are substantially slower than GPU inference. For instance, the finetuned all-mpnet-base-v2 model is 36.16 times, while all-MiniLM-L12-v2 is 32.7 times slower to classify 1000 samples. In contrast to GPU inference, all-MiniLM-L12-v2 is 2.38 and 5.19 times faster than all-distilroberta-v1 and all-mpnet-base-v2 respectively.

Chapter 6

Discussion

In this Section, we analyze the results from our study and discuss the research questions from Section 1.2.1. We also higlight secondary benefits of the proposed method in Section 6.4.

6.1 Overall Performance

One of the primary considerations when integrating an automated vulnerability scanner into a software project is performance. Thus, the first research question we examine is *What is the performance of the proposed method in identifying selected class of web application vulnerabilities?* In our study, we evaluated the performance of our proposed method from multiple perspectives. First, we measured the detection performance using the evaluation metrics from Section 3.6.1. In our experiments, the model with the best results had an accuracy of 87.19%, with F1 and MCC scores of 0.82 and 0.7 respectively. This is a great achievement when the difficulty of the task is considered. Detecting CWE-639 and CWE-209 vulnerabilities is time-consuming and complex. Additionally, it requires cybersecurity expertise. KARTAL automates this process, saving time and resources.

Another performance indicator is inference latency. Agile software development practices have normalized continuous deployments that substantially reduced the code to production duration. As a result, the latency expectations for vulnerability scanning tools have also been impacted. Unlike most detection tools, KARTAL is able to utilize hardware acceleration with GPUs. Depending on the base model, Figure 5.2 demonstrates that it can classify 114 to 539 samples per second running on a laptop GPU. At this inference speed, we can scan large-scale projects within minutes compared to up to a few days by manual review. The resource-efficient architecture of KARTAL enables it to be used on lower-end hardware, which democratizes the usage of cutting-edge technology for the masses.

Nevertheless, without hardware acceleration, the inference latency suffers greatly. On CPU, our method can predict 3 to 17 labels per second depending on the finetuned model. Although this is slower, it is still above the human baseline for labeling speed. The automated scanner can run 7/24 without pauses and work in parallel with other scanners to improve the overall efficiency of vulnerability detection. We will dive deeper into possible improvements in the next section.

6.2 Key Factors in Performance

In this section, we aim to answer the research question *What factors contribute to the performance of the proposed method?* Increasing the size of the training dataset is the most effective way to improve the performance of our method. We can see a clear trend in Figure 5.1a and Figure 5.1e, adding just 400 more samples can result in a significant boost in performance. This, on average, has been demonstrated to enhance the MCC score by 19.58%. Furthermore, from 1400 to 1600 samples, the rate of improvement in accuracy increases 5.5-fold. Figure 5.2 validates this by visualizing changes in predictions per class. From 800 to 1200 training samples, the false positive rate is reduced significantly across the confusion matrix. As a result of this experiment, we conclude that the model develops a better knowledge of language subtleties by including more varied and representative data, resulting in improved performance.

Another key factor in performance is the selection of the pre-trained transformer. Larger models, in general, outperform smaller models due to their enhanced ability to capture complicated language patterns. This is evident from the order of performance in Figure 5.1. The all-mpnet-base-v2 model is a pre-trained model based on the massive mpnet-base that is finetuned on over a billion sentences across 32 datasets. These datasets include tasks that improve different parts of language understanding of the model. It is worth mentioning that distilled versions of bigger models that have been compressed can also provide excellent outcomes. These distilled versions are smaller in size yet retain a considerable chunk of their bigger counterparts' performance characteristics [36, 37].

Both the all-distilroberta-v1 and all-MiniLM-L12-v2 models share the same datasets with all-mpnet-base-v2, with the key difference of being finetuned versions of distilled models. The former model is based on a distilled version of the RoBERTa-base model. The latter model finetunes a distilled bert-base model. The difference in MCC score between finetuned all-mpnet-base-v2 and all-MiniLM-L12-v2 models is 7.7 times higher compared to the difference between finetuned all-mpnet-base-v2 and all-distilroberta-v1 model. If speed is the only selection criterion for your business case, all-MiniLM-L12-v2 is a decent choice for finetuning. However, when the duration of the training, hardware requirement, inference latency, and evaluation performance is considered, the all-distilroberta-v1 model is the best all-around performer.

It is critical to consider the interaction between the pre-trained model selection and the inference hardware selection. Due to their increasing model size and complexity, larger pre-trained models, often demand more computing resources during inference. In such instances, it is beneficial to use strong GPUs or specialized accelerators such as TPUs to achieve efficient and real-time LLM predictions. Furthermore, distilled versions of bigger models offer a balance between performance and resource needs. When compared to their bigger counterparts, distilled models have a reduced memory footprint and lower computing needs. As a result, executing inference on CPUs or using less capable GPUs can still produce excellent results for distilled models.

Another method of improving the speed of the proposed method is to run on distributed cloud architecture. Large companies build software of grand sizes and quantities, and any change in vulnerability detection speed can have exponential results in secure software production. All components of KARTAL can function in parallel within their responsibility zones. Multiple instances of the Detector can run on different parts of the Fuzzer data. By parallelizing the detection unit, KARTAL can scale horizontally and substantially reduce the time to detect at a fraction of the cost.

6.3 Challenges and Limitations

Our third and final research question was *What are the challenges and limitations of the proposed method?* The most important limitation of our method is the limited sequence length. This limit is inherited from the pre-trained transformer of our choice. In this study, our best performer was a finetuned all-mpnet-base-v2 model. As viewed on Table 3.3, it can accept at most 384 tokens while all-distilroberta-v1 and all-MiniLM-L12-v2 can utilize 512 and 256 tokens respectively. This limit directly affects the number of requests that can be added to each prompt. As a result, deeply nested vulnerabilities that require the inspection of dozens of HTTP requests at the same time cannot be detected by our method.

Another limitation is pre-trained model availability. In this study, we finetuned free and open-source models. Our performance analysis showed a direct relationship between the size of the pre-trained model and the performance of the finetuned model. These models are released to the public as a by-product of academic research. Training a large language model can take multiple weeks and millions of dollars. Consequently, the size of open-source language models is restricted by the cost of training them. In contrast, commercially available models do not have such restrictions and can achieve SOTA performance. For this study, we preferred to utilize open-source models in order to keep our method transparent and results reproducible.

A great challenge in the study was the manual labeling process of the generated data. At a rate of 45 seconds per sample, it took a cybersecurity specialist 23 hours to label the entire dataset. As mentioned in 6.2, the finetuned model demonstrates great performance improvement at every training sample size experiment. In Figure 5.1e, we observe a 23.53% increase in MCC by adding 400 more samples to training data, implying that the model can perform significantly better if slightly more data was available. However, this could not be achieved within the timeframe of this thesis.

6.4 Auxiliary Advantages

There are a couple of secondary advantages of KARTAL that should be mentioned. The first is the dynamic properties of *Context* in the prompt. Both types of vulnerabilities we aim to detect break the logical rules defined by the application, hence we provide the boundaries of the application within each prompt. The benefit of this approach is the ability to customize the detection during inference. For instance, an enterprise application specialized access rules for each of its tenants. A non-intelligent vulnerability scanner would simply struggle while others may require retraining or finetuning process. In contrast, users of KARTAL can dynamically change the definitions of *broken access* to adapt to each situation.

Another benefit of KARTAL is its ability to detect vulnerabilities in multiple languages with transfer learning [78]. The technique leverages the existing knowledge of one topic or task to improve the performance of different tasks. LLMs are great Transfer Learners due to their architecture and size. If a pre-trained model has multilanguage data in its training set, finetuning it enables the multilingual capabilities of KARTAL. Our small set of experiments showed similar results to Figure 5.1 for detecting vulnerabilities in web applications that display languages other than English. Nonetheless, we do not highlight these results due to the limited size of multilingual samples.

Chapter 7

Conclusions

In this chapter, we complete our thesis study. Section 7.1 draws final conclusions. In Section 7.2 we discuss possible paths for future work. Finally, in Section 7.3 we reflect on the ethical and environmental impacts of our research.

7.1 Conclusions

Web applications have become lucrative targets for innovative hackers due to the multi-billion SaaS market. Finding critical logical vulnerabilities before bad actors requires intelligent and automated tools. The objective of this thesis was to automate the detection of complex and logical vulnerabilities in web applications. We modeled the problem as text classification and proposed KARTAL, a novel method for achieving this task. Our study questioned the performance, key factors, and limitations of the proposed method.

Due to a lack of existing training data, we custom-generated our own dataset. For this purpose, we used an auto-regressive LLM with SOTA few-shot prompting techniques. Next, we finetuned 3 types of decoder-only pre-trained transformers for detecting 2 sophisticated vulnerabilities using SetFit. Our best model attained an accuracy of 87.19%, with F1 and MCC scores of 0.82 and 0.7 respectively. By using hardware acceleration on a consumer-grade laptop, our fastest model can make up to 539 predictions per second. The experiments on varying the training sample size demonstrated the great learning capabilities of our model. Every 400 samples added to training resulted in an average MCC score improvement of 19.58%.

The study successfully achieved all of its objectives by creating a high-accuracy web application vulnerability detection method using LLMs. The usage of LLMs in cybersecurity is a novel but fastly evolving field, and our method proves its great potential. The availability of open-source pre-trained transformers has been steadily increasing. Consequently, cybercriminals have already started to utilize such models for their malicious activities. As cybersecurity researchers, we must proactively adopt cutting-edge technologies, such as LLMs, to create intelligent and adaptive defense systems. We predict that in the next couple of years, we will see bursts of innovation in the cybersecurity field using this technology that change the way we build secure software.

7.2 Future Work

As we analyzed in Section 5.2, the model is highly appreciative of increments in training data size. Manual labeling of data at scale is not viable. A possible research direction would be to use Active Learning, which is a technique to improve the training performance of a model by selectively marking unlabeled samples for labeling [79]. It then uses the new samples to improve its prediction performance. By employing this method, we can rapidly scale our dataset which will notably improve the performance of the LLM.

Another following research focus could be to add detection capabilities for a greater number of vulnerabilities. In this study, we finetuned our model to only identify two types of vulnerabilities. Extending the model in this way will increase its multitasking capabilities and enable direct comparison with existing tools. The model can be also trained for multi-label classification which will further improve the performance of KARTAL by identifying multiple vulnerabilities at once.

A final future direction can be to connect the output of the Detector component to Fuzzer. When the Detector senses a vulnerability without great certainty, it can communicate this with the Fuzzer. Using this knowledge, the Fuzzer will then probe the web application for more request samples that are likely to increase the prediction confidence of the model. This process extends the automation capabilities of the model far beyond any other existing tool, essentially turning the model into a self-supervised penetration tester.

7.3 Reflections

Most often, the malicious usage of technology is an accepted risk. This also applies to our research. Bad actors can utilize our proposed method to find vulnerabilities in systems they plan to breach. However, this is the case with all vulnerability detection tools. As the intellectual property owner of the model, we would like to issue the following disclaimer: KARTAL is published for ethical use cases only. We do not condone unethical usage, and do not grant permission for KARTAL to be used against systems without legal authorization.

Our research contributes to Sustainable Development Goals (SDGs) [10] 8,9 and 10. SDGs 8 and 9 promote sustainable economic growth and resilient infrastructure. Cyberattacks have the power to disrupt critical infrastructure. At scale, a targeted attack on a city by threat actors with extensive resources could create a chain reaction of events that lead to a massive loss in productivity, economic growth, and unemployment. KARTAL enables the preemptive detection of vulnerabilities that the attackers could utilize, completely preventing them from happening. SDG 10 is related to reducing inequalities among countries. KARTAL is built for high performance and can effortlessly run on lower-end hardware. As a result, commercial and non-profit organizations in countries with insufficient computing resources can also benefit from the security features KARTAL offers.

Bibliography

[1] Gartner, “Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $600 Billion in 2023 — gartner.com,” https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023, 2022, [Accessed 06-Jan-2023].

[2] A. Nguyen-Duc, M. V. Do, Q. Luong Hong, K. Nguyen Khac, and A. Nguyen Quang, “On the adoption of static analysis for software security assessment–a case study of an open-source e-government project,” *Computers & Security*, vol. 111, p. 102470, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821002947

[3] F. O. Sonmez and B. G. Kilic, “Holistic web application security visualization for multi-project and multi-phase dynamic application security test results,” *IEEE access*, vol. 9, pp. 25 858–25 884, 2021.

[4] F. Mateo Tudela, J.-R. Bermejo Higuera, J. Bermejo Higuera, J.-A. Sicilia Montalvo, and M. I. Argyros, “On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications,” *Applied sciences*, vol. 10, no. 24, pp. 9119–, 2020.

[5] W. Guo, Y. Fang, C. Huang, H. Ou, C. Lin, and Y. Guo, “Hyvuldect: A hybrid semantic vulnerability mining system based on graph neural network,” *Computers & security*, vol. 121, pp. 102 823–, 2022.

[6] A. Van Der Stock, B. Glas, N. Smithline, and T. Gigler, “OWASP Top 10:2021 — owasp.org,” https://owasp.org/Top10/, 2021, [Accessed 01-Jan-2023].

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learn-

ers," 2019. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[8] MITRE, "CWE-639: Authorization Bypass Through User-Controlled Key," https://cwe.mitre.org/data/definitions/639.html, 2008, [Accessed 09-Jan-2023].

[9] ——, "CWE-209: Generation of Error Message Containing Sensitive Information," https://cwe.mitre.org/data/definitions/209.html, 2006, [Accessed 09-Jan-2023].

[10] U. N. Press, "Transforming our world: the 2030 agenda for sustainable development," https://sdgs.un.org/2030agenda, 2015.

[11] N. C. Oy, "Netlight webpage," https://www.netlight.com/, 201623, [Accessed 09-Jan-2023].

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv.org*, 2017.

[13] R. Bommasani, D. A. Hudson, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch, D. Card, R. Castellon, N. Chatterji, A. Chen, K. Creel, J. Q. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, F.-F. Li, C. Finn, T. Gale, L. Gillespie, K. Goel, N. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, D. E. Ho, J. Hong, K. Hsu, T. Icard, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. Krass, R. Krishna, R. Kuditipudi, A. Kumar, F. Ladhak, M. Lee, T. Lee, J. Leskovec, X. L. Li, X. Li, A. Malik, S. Mirchandani, E. Mitchell, Z. Munyikwa, S. Nair, A. Narayan, D. Narayanan, A. Nie, J. C. Niebles, H. Nilforoshan, J. Nyarko, G. Ogut, I. Papadimitriou, J. S. Park, C. Piech, E. Portelance, C. Potts, A. Raghunathan, R. Reich, H. Ren, F. Rong, Y. Roohani, C. Ruiz, J. Ryan, C. Ré, D. Sadigh, S. Sagawa, K. Santhanam, A. Shih, A. Tamkin, R. Taori, A. W. Thomas, R. E. Wang, W. Wang, B. Wu, J. Wu, Y. Wu, M. Yasunaga, J. You, M. Zaharia, M. Zhang, Y. Zhang, L. Zheng, K. Zhou, and P. Liang, "On the opportunities and risks of foundation models," *arXiv.org*, 2021. [Online]. Available: https://crfm.stanford.edu/assets/report.pdf

[14] J. Devlin, C. Ming-Wei, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv.org*, 2019.

[15] J. Oh and Y.-S. Choi, “Reusing monolingual pre-trained models by cross-connecting seq2seq models for machine translation,” *Applied sciences*, vol. 11, no. 18, pp. 8737–, 2021.

[16] V. Pandelea, E. Ragusa, T. Young, P. Gastaldo, and E. Cambria, “Toward hardware-aware deep-learning-based dialogue systems,” *Neural computing & applications*, vol. 34, no. 13, pp. 10 397–10 408, 2022.

[17] C. Xu, F. Yuan, and S. Chen, “Bjbn: Bert-join-bilstm networks for medical auxiliary diagnostic,” *Journal of healthcare engineering*, vol. 2022, pp. 3 496 810–7, 2022.

[18] R. Irwin, S. Dimitriadis, J. He, and E. J. Bjerrum, “Chemformer: a pre-trained transformer for computational chemistry,” *Machine learning: science and technology*, vol. 3, no. 1, pp. 15 022–, 2022.

[19] R. Sawai, I. Paik, and A. Kuwana, “Sentence augmentation for language translation using gpt-2,” *Electronics (Basel)*, vol. 10, no. 24, pp. 3082–, 2021.

[20] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” *arXiv.org*, 2021.

[21] M. DeLeo and E. Guven, “Learning chess with language models and transformers,” in *Data Science and Machine Learning.* Academy and Industry Research Collaboration Center (AIRCC), 9 2022. [Online]. Available: https://doi.org/10.5121%2Fcsit.2022.121515

[22] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 5 2015.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 10 1986.

[24] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[26] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 10 2014, pp. 1724–1734. [Online]. Available: https://aclanthology.org/D14-1179

[27] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent models of visual attention," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 2204–2212.

[28] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016.

[29] P. Feldman, A. Dant, and D. Rosenbluth, "Ethics, rules of engagement, and ai: Neural narrative mapping using large transformer language models," 2022.

[30] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 7 2021, pp. 8821–8831. [Online]. Available: https://proceedings.mlr.press/v139/ramesh21a.html

[31] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021.

[32] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang, "Conformer: Convolution-augmented Transformer for Speech Recognition," in *Proc. Interspeech 2020*, 2020, pp. 5036–5040.

[33] B. Shin, S. Park, K. Kang, and J. C. Ho, "Self-attention based molecule representation for predicting drug-target interaction," in *Proceedings of the 4th Machine Learning for Healthcare Conference*, ser.

Proceedings of Machine Learning Research, F. Doshi-Velez, J. Fackler, K. Jung, D. Kale, R. Ranganath, B. Wallace, and J. Wiens, Eds., vol. 106. PMLR, 09–10 Aug 2019, pp. 230–248. [Online]. Available: https://proceedings.mlr.press/v106/shin19a.html

[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 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)*, 2019, pp. 3287–3293.

[35] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations (ICLR)*, 2020.

[36] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," in *Proceedings of the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing*, 2020.

[37] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pretrained transformers," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[38] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "Mpnet: Masked and permuted pre-training for language understanding," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[39] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 11 2019. [Online]. Available: https://arxiv.org/abs/1908.10084

[40] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M.

Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *arXiv.org*, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[41] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019.

[42] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, 6 2020.

[43] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, 11 2018, pp. 353–355. [Online]. Available: https://aclanthology.org/W18-5446

[44] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, W. Y. Adams, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2 2022. [Online]. Available: https://www.proquest.com/working-papers/finetuned-language-models-are-zero-shot-learners/docview/2569483346/se-2

[45] M. E. Peters, S. Ruder, and N. A. Smith, "To tune or not to tune? adapting pretrained representations to diverse tasks," in *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*. Florence, Italy: Association for Computational Linguistics, 8 2019, pp. 7–14. [Online]. Available: https://aclanthology.org/W19-4302

[46] Y. Wang, S. Si, D. Li, M. Lukasik, F. Yu, C.-J. Hsieh, I. S. Dhillon, and S. Kumar, "Preserving in-context learning ability in large language model fine-tuning," 2022.

[47] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for nlp," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2790–2799.

[48] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel, "Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 1950–1965, 2022.

[49] T. Schick and H. Schütze, "Exploiting cloze-questions for few-shot text classification and natural language inference," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume.* Online: Association for Computational Linguistics, 4 2021, pp. 255–269. [Online]. Available: https://aclanthology.org/2021.eacl-main.20

[50] L. Tunstall, N. Reimers, U. E. S. Jo, L. Bates, D. Korat, M. Wasserblat, and O. Pereg, "Efficient few-shot learning without prompts," 2022.

[51] Z. Cheng, J. Kasai, and T. Yu, "Batch prompting: Efficient inference with large language model apis," 2023. [Online]. Available: https://arxiv.org/abs/2301.08721

[52] Z. Tan, X. Zhang, S. Wang, and Y. Liu, "MSP: Multi-stage prompting for making pre-trained language models better translators," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 6131–6142. [Online]. Available: https://aclanthology.org/2022.acl-long.424

[53] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[54] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," 2022. [Online]. Available: https://arxiv.org/abs/2212.06094

[55] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," *arXiv.org*, 2022.

[56] X. Zhao, S. Ouyang, Z. Yu, M. Wu, and L. Li, "Pre-trained language models can be fully zero-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2212.06950

[57] S. Shin, S.-W. Lee, H. Ahn, S. Kim, H. Kim, B. Kim, K. Cho, G. Lee, W. Park, J.-W. Ha, and N. Sung, "On the effect of pretraining corpora on in-context learning by a large-scale language model," in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Seattle, United States: Association for Computational Linguistics, 7 2022, pp. 5168–5186. [Online]. Available: https://aclanthology.org/2022.naacl-main.380

[58] J. Wei, X. Wang, D. Schuurmans, M. Bosma, brian ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022.

[59] S. Arora, A. Narayan, M. F. Chen, L. Orr, N. Guha, K. Bhatia, I. Chami, and C. Re, "Ask me anything: A simple strategy for prompting language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[60] P. Michel, X. Li, G. Neubig, and J. Pino, "On evaluation of adversarial perturbations for sequence-to-sequence models," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, 6 2019, pp. 3103–3114. [Online]. Available: https://aclanthology.org/N19-1314

[61] MITRE, "Common weaknesses enumaration," https://cwe.mitre.org/about/index.html, 2006, [Accessed 09-Jan-2023].

[62] S. Elder, N. Zahan, R. Shu, M. Metro, V. Kozarev, T. Menzies, and L. Williams, "Do i really need all this work to find vulnerabilities?: An empirical case study comparing vulnerability detection techniques on a java application," *Empirical software engineering : an international journal*, vol. 27, no. 6, 2022.

[63] M. Liu, B. Zhang, W. Chen, and X. Zhang, "A survey of exploitation and detection methods of xss vulnerabilities," *IEEE Access*, vol. 7, pp. 182 004–182 016, 2019.

[64] R. Amankwah, J. Chen, P. K. Kudjo, and D. Towey, "An empirical comparison of commercial and open-source web vulnerability scanners," *Software, practice & experience*, vol. 50, no. 9, pp. 1842–1857, 2020.

[65] M. Albahar, D. Alansari, and A. Jurcut, "An empirical comparison of pen-testing tools for detecting web app vulnerabilities," *Electronics*, vol. 11, no. 19, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/19/2991

[66] I. Medeiros, N. Neves, and M. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–11. [Online]. Available: https://doi.org/10.1145/2931037.2931041

[67] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," *arXiv.org*, 9 2022. [Online]. Available: https://www.proquest.com/working-papers/transformer-based-language-models-software/docview/2648330330/se-2

[68] C. Torrano-Gimenez, A. Perez-Villegas, and G. Alvarez, "TORPEDA: Un conjunto de datos ampliable para la evaluación de cortafuegos de aplicaciones web," in *Proceedings of the XII Reunión Española sobre Criptología y Seguridad de la Información*, San Sebastián, 2012.

[69] LIRMM, "Analyzing web traffic ecml/pkdd 2007 discovery challenge september 17-21, 2007, warsaw, poland," 2007. [Online]. Available: https://www.lirmm.fr/pkdd2007-challenge/index.html

[70] G. Ltd, "Google colab," https://colab.research.google.com, 2015, [Accessed 09-Jan-2023].

[71] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[72] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison,

S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, 10 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[73] D. Inc, "Docker," https://www.docker.com/, 2013, [Accessed 09-Jan-2023].

[74] M. Feurer and F. Hutter, "Hyperparameter optimization," *Automated machine learning: Methods, systems, challenges*, pp. 3–33, 2019.

[75] N. Reimers, "Sbert pretrained model comparison," https://www.sbert.net/docs/pretrained_models.html#model-overview, 2020, [Accessed 09-Jan-2023].

[76] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, p. 6, 1 2020.

[77] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[78] A. Hosna, E. Merry, J. Gyalmo, Z. Alom, Z. Aung, and M. A. Azim, "Transfer learning: a friendly introduction," *J. Big Data*, vol. 9, no. 1, p. 102, 10 2022. [Online]. Available: https://doi.org/10.1186/s40537-022-00652-w

[79] M. Gao, Z. Zhang, G. Yu, S. O. Arik, L. S. Davis, and T. Pfister, "Consistency-based semi-supervised active learning: Towards minimizing labeling cost," 2020.

Appendix A

Environment Setup

Listing A.1: Dockerfile contents for experiment environment

```
# Use the huggingface/transformers-pytorch-gpu base
   image
FROM huggingface/transformers-pytorch-gpu:latest

# Set the working directory
WORKDIR /app

# Copy the requirements.txt file to the container
COPY requirements.txt .

# Install the Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the entire folder to the container
COPY . .
```

Listing A.2: Instructions for installing and running experiments

```
## Directory Structure

- data/ (contains datasets in CSV format)
- prompt-templates/ (contains prompt template files)
- requirements.txt (contains the list of Python
   dependencies)
- Dockerfile
- src/ (contains source code)
  - benchmark/ (contains benchmark-related code)
```

~~~
  - data-generation/ (contains data generation code)
  - plotting/ (contains plotting code)


## Requirements

To run the KARTAL project, you need the following:

- Python 3.10+
- PyTorch 2

or

- Docker 20+

## How to Run

For data generation using the `prompter.py` script you
   will need to create a `.env` file. Use the `.env-
   example` file as the template and fill in the
   required values. You can get them from the
   respective LLM API dashboard.

### Native

1. Install Python 3.10+ on your system.
2. Install the required Python packages by running the
   following command:

```shell
pip install -r requirements.txt
~~~
3. Run experiments or data generation scripts located
   in the `src/` directory.

### Docker

1. Build the Docker image by running :

```shell
docker build -t kartal .
```

```
```
2. Run the Docker container using the built image and
   specify the Python file to run:

```shell
docker run --gpus all -it kartal python src/benchmark/{
   file_name}.py
```
Replace `{file_name}` with the path to your desired
   Python file within the `src/` directory.
```
```

Appendix B

Data Generation Prompt

Listing B.1: Prompt example used for generating samples

```
Task: Generate 10 examples based on the template, do
   not give explanations, label titles, or example
   count, only provide the examples.
Task description: Examples SHOULD demonstrate CWE-209:
   Generation of Error Message Containing Sensitive
   Information security flaw. The sensitive information
    may be valuable information on its own (such as a
   password), or it may be useful for launching other,
   more serious attacks. The error message may be
   created in different ways:
self-generated: the source code explicitly constructs
   the error message and delivers it.
externally-generated: the external environment, such as
    a language interpreter, handles the error and
   constructs its own message, whose contents are not
   under direct control by the programmer.
An attacker may use the contents of error messages to
   help launch another, more focused attack. For
   example, an attempt to exploit a path traversal
   weakness (CWE-22) might yield the full pathname of
   the installed application. In turn, this could be
   used to select the proper number of ".." sequences
   to navigate to the targeted file. An attack using
   SQL injection (CWE-89) might not initially succeed,
   but an error message could reveal the malformed
   query, which would expose query logic and possibly
   even passwords or other sensitive information used
   within the query.
Template:
Context: [Details on the app, Roles of people using,
   What each role can and cannot do]. Requests: [An
   array of Request&Response pairs. Min. 1 Max. 5 pairs
   . A Request must be followed by corresponding
   Response.]
List of Examples, each line represents a unique example
   :
#Comment:The following app fails to sanitize its error
   response, leaking information about account
   existance
Context: An ecommerce app with 3 types of users
   Customer, Merchant and Admin. Customers and
   Merchants can only view and edit their own data.
   Admins can view and edit all data. Requests: [
   Request-1: Customer-A POST /workshop/api/auth/login
   with parameters username='micheal',password='123';
   Response-1: Unauthorized with parameters success=
   false,message='123 is incorrect password';Request-2:
    Customer-B POST /workshop/api/auth/login with
   parameters username='beatrice',password='123123';
   Response-2: OK with parameters token='uYda27...']
#Comment: The following app leaks sensitive system info
    to end users after a crash
Context: Travel booking website with 1 type of users
   Customer, where customers book their own flights.
   Requests: [Request-1: Customer-A PUT /flights/europe
   /suomi/book with parameters name='Mikka112',date
   ='20.10.2022';Response-1: Bad Request with
   parameters message='Ineffective mark-compacts near
   heap limit Allocation failed - JavaScript heap out
   of memory, total=15394 available=13, core dumped']
#Comment: The following app leaks information about
   accounts in response error message
Context: A school management system where teachers can
   only view and modify their own grades and attendance
    records. Requests: [Request-1: Customer-A POST /
   school/records/security/check with parameters
   credential='micheal@gmail.com',password='A2B1C';
```

```
    Response-1: Forbidden with parameters details='
    account not found, did you mean micheal@school.com
    ?']
#Comment: The following app leaks email of user in
    response message without verification
Context: A healthcare platform with 2 type of users
    Patient and Doctor, where patients can view and
    manage their own healthcare records. Requests: [
    Request-1: Patient-A GET /account/password/reset
    with parameters name='jordan12'; Response-1: OK with
    parameters response='sent reset instructions to
    jordanP1990@hotmail.com. If this is not your email
    please check the name you provided.']
#Comment: The following app leaks the contents of user'
    s private file in the response message
Context: A cloud-based storage app with 2 type of users
     User and Admin, where each user can only view and
    download their own files. Admins cannot view user
    files. Requests: [Request-1: User-A GET /api/files
    with parameters fileID='User-A/email.docx'; Response
    -1: OK with parameters fileID='User-A/email.docx',
    content='Hello World!', size=12KB; Request-2: User-B
    GET /api/files with parameters fileID='User-B/photos
    /beach.jpg'; Response-2: OK with parameters fileID='
    User-B/photos/beach.jpg', content ['...'], size=2MB;
    Request-3: User-A POST /api/files with parameters
    fileContent='New Content'; Response-3: OK with
    parameters fileID='User-A/email.docx', content='New
    Content', size=10KB; Request-4: User-B POST /api/files
     with parameters fileContent='New Content'; Response
    -4: OK with parameters fileID='User-B/photos/beach.
    jpg', content='New Content', size=1.5MB; Request-5:
    Admin GET /api/files with parameters fileID='User-A/
    email.docx'; Response-5: OK with parameters message='
    Cannot parse email.docx: 'from=User-A@gmail.com to=
    lover@gmail.com title=Secret body=6s76sad/&TA29' is
    missing quotes']
#Comment: The following app leaks customer's order
    history to another customer in error message
Context: A food ordering app with 2 type of users
    Customer and Restaurant where each customer can only
    access and modify their own orders. Restaurant can
    view all orders. Requests: [Request-1: Customer-A
    GET /api/orders with parameters orderID='Order-A';
    Response-1: OK with parameters orderID='Order-A',
    menu=['pizza','coke'],status='Delivered';Request-2:
    Customer-B GET /api/orders with parameters orderID='
    Order-A';Response-2: Forbidden with parameters
    results='you are not allowed to view orderID=\'Order
    -A\',menu=[\'pizza\',\'coke\'],status=\'Completed
    \'';]
#Comment: The following app leaks name and function of
    internal library in use
Context: Travel booking website with 1 type of users
    Customer, where customers book their own flights.
    Requests: [Request-1: Customer-A PUT /store/login
    with parameters email='micheal@gmail.com',password='
    ABC';Response-1: Bad Request with parameters code
    ='400',error='php-hash-lib: cannot hash, digest size
     surpassed']
#Comment: The following app leaks confidential internal
     network communication information
Context: An online marketplace with 2 type of users
    Seller and Buyer for advertisements where each
    seller can only see their own listings. Admins can
    view and delete all listings. Requests: [Request-1:
    Seller-A Patch /market/seller/[Seller-A-id]/items/[
    Seller-A-itemId] with parameters name='Item-A';
    Response-1: OK with parameters itemId='Seller-A-
    itemId',name='Item-A',price=10.5,quantity=103;
    Request-2: Seller-B Get /market/seller/[Seller-A-id
    ]/items/[Seller-B-itemId] with parameters;Response
    -2: Forbidden with parameters status='unsuccessful',
    reason='request to http://internal-service:4568
    returned gateway timeout']
#Comment: The following app leaks database sql query
    details
Context: A messaging app with 2 types of users User and
     Admin where Users can chat with each other. Chats
    are encrypted end-to-end. Requests: [Request-1: User
    -1 /messages/august/search with parameters terms='
    who should win eurovison?';Response-1: Error with
```

```
parameters msg='cannot serialize response {command:
'SELECT',rowCount: 3,oid: null,rows:[{ id: 1, name:
'John', age: 25 },{ id: 2, name: 'Jane', age: 30 },{
 id: 3, name: 'Sam', age: 35 }],fields: [{ name: 'id
', tableID: 123, columnID: 1, dataTypeID: 23 },{
name: 'name', tableID: 123, columnID: 2, dataTypeID:
 25 },{ name: 'age', tableID: 123, columnID: 3,
dataTypeID: 23 },commandComplete: 'slct * fr tb lim
3 }']
```