Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Peyton E. Robertson March 31, 2023

Vulnerability Detection: A Machine Learning Approach to Identifying Security Vulnerabilities In Code

By

Peyton E. Robertson

Ymir Vigfusson, Ph.D.
Advisor

Computer Science

Ymir Vigfusson, Ph.D.
Advisor

Davide Fossati, Ph.D.
Committee Member

Emily Wall, Ph.D.
Committee Member

Li Xiong, Ph.D.
Committee Member

2023

Vulnerability Detection: A Machine Learning Approach to Identifying Security Vulnerabilities In Code

By

Peyton E. Robertson

Ymir Vigfusson, Ph.D.
Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Arts with Honors
Computer Science
2023

Abstract

Vulnerability Detection: A Machine Learning Approach to Identifying Security Vulnerabilities In Code
By Peyton E. Robertson

Cybercrime is a rapidly growing threat that poses significant risks to individuals, businesses, and governments worldwide. With the proliferation of technology and the increasing sophistication of cybercriminals, there is an enormous need for practical and effective techniques to identify security vulnerabilities hidden in source code. We propose a novel approach to vulnerability detection that combines machine learning with fuzzing techniques in order to identify areas of a program that are more likely to contain possible security exploits. This approach strives to improve on the state-of-the-art for automated vulnerability detection by addressing the challenges that fuzzing faces in selecting and mutating seed inputs, expanding code coverage, and bypassing verification checks. The machine learning component of our study relies on Microsoft's CodeBERTa, a bimodal pre-trained model for natural language processing, that is ideal for vulnerability detection because it can utilize both natural language and source code as inputs. We fine-tune the model an expanded version of the vulnerability dataset curated by Zhou et al [54] and evaluate its performance both individually and comparatively. The results of our model include an overall accuracy score of 62.88%, an F1-score of .55, a ROC-AUC score of .70, and a PR-AUC score of .666. These findings suggest that the model can identify vulnerabilities in source code with relative accuracy and that employment of machine learning techniques can enhance the efficacy of vulnerability detection. As such, our CodeBERTa model Improves the effectiveness of vulnerability detection techniques and assists software engineers in regaining the advantage in the battle against cybercrime.

Vulnerability Detection: A Machine Learning Approach to Identifying Security Vulnerabilities In Code

By

Peyton E. Robertson

Ymir Vigfusson, Ph.D.
Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Arts with Honors
Computer Science

2023

Acknowledgments

First, I would like to express my sincere gratitude to my advisor, Dr. Vigfusson, for his support and guidance throughout my undergraduate thesis project and my time at Emory. It was his research and teaching that first drew me to Emory, and I feel incredibly fortunate to have had the opportunity to work under his mentorship. His insightful feedback, patience, encouragement, and industry insight have been invaluable to me, and I cannot thank him enough for all that he has done.

I would also like to thank my committee members, Dr. Davide Fossati, Dr. Emily Wall, and Dr. Li Xiong, for their invaluable guidance and feedback on my research. Their expertise in their respective fields and thoughtful critiques have helped me to refine and improve my work, and I am grateful for their time and commitment to serving on my committee.

In addition, I would like to express my appreciation to the many wonderful professors in the computer science and political science departments at Emory who have challenged and supported me throughout my undergraduate studies. Their passion for teaching has helped me to grow both academically and personally, and I am so appreciative of their influence on my academic journey.

Lastly, I would like to thank my friends and family for their unwavering support throughout this journey. They have spent countless hours editing, listening to my code explanations, and being my strongest support system. Their love and encouragement have kept me motivated and focused throughout the ups and downs of my undergraduate studies, and I am forever grateful for their presence in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cybercrime is at an all-time high – and it is only predicted to get worse. According to Editor-in-Chief Steve Morgan at Cybercrime Magazine, cybercrime costs are predicted to grow at least 15 percent annually over the next five years, reaching an astounding $10.5 trillion in costs by 2025 [46]. The cost of global cybercrime is higher than the damage inflicted by natural disasters in the last year, is greater than any transfer of economic wealth in history and is more profitable than the entire international illegal drug trade [46]. With the toll of cybercrime rising, it is more important than ever that developers and industry experts can identify and resolve security vulnerabilities as quickly as possible.

Software vulnerabilities refer to weaknesses in software that can be exploited by attackers to gain unauthorized access, steal sensitive information, or disrupt operations [22]. Such vulnerabilities can arise from various factors, including programming errors, code design flaws, or inadequate testing of program logic. Exploiting these vulnerabilities typically involves crafting malicious inputs or executing arbitrary code to trigger unintended behavior in the targeted system [2]. Once successful, the attacker can execute various attacks depending on the type and severity of the vulnerability. Such attacks are occurring at alarmingly high rates, as noted by director of the Com-

puter Emergency Response Team (CERT) Rich Pethia, who stated that "the number of vulnerabilities in commercial off-the-shelf software is now at the level that it is virtually impossible for any but the best-resourced organizations to keep up with the vulnerability fixes" [35]. In the two decades since Pethia's testimony, his words have proven to be increasingly accurate.

One of the ways engineers mitigate the threats posed by vulnerable code is through a process called fuzz testing or fuzzing. Fuzzing is an automated software technique that injects semi-random data into a program in order to detect bugs within the source code of an application or service [30]. The semi-random data used to test programs typically consists of invalid data, files, network packets, program codes, and other edge cases that might cause unexpected behavior to occur. Such erroneous behavior creates conditions in a program's control flow that a hacker can potentially exploit to access a computer, network, program and all the information they may contain. Since Professor Barton Miller at the University of Wisconsin Madison first introduced the concept of fuzzing in 1989, the technique has expanded to include approaches like static and dynamic analysis, both of which are intended to improve the efficiency and robustness of fuzz testing [12]. However, despite these improvements, fuzzing still faces enormous challenges in terms of selecting and mutating seed inputs (the initial test cases used to kickstart the search for vulnerabilities), expanding code coverage (especially in multi-level source code), and bypassing verification checks within a given code base [30]. These challenges create a considerable gap in the ability of developers to combat cybercrime; this project attempts to shrink this gap.

Against this backdrop, we propose an innovative approach to vulnerability discovery that combines machine learning with state-of-the-art fuzzing techniques. The machine learning component of this project relies on Microsoft's bimodal pre-trained model for natural language processing called CodeBERTa, which was first published in [25]. Unlike previous models, such as BERT and Roberta, CodeBERTa can utilize

both natural language and source code as inputs. As such, it is ideal for vulnerability detection [18]. To teach the model to distinguish between vulnerable and secure code, it is trained on a Common Vulnerability and Exposure (CVE) database that contains both patches and vulnerabilities. CVEs are unique identifiers assigned to publicly known cybersecurity vulnerabilities in software and hardware systems and often contain details on the vulnerability itself as well as the "patch", or solution for that vulnerability. Within the dataset, patches and vulnerabilities are classified as 0 for non-vulnerable or 1 for vulnerable, respectively. This methodology enables the model to identify vulnerable functions within source code or other lengthy code segments. Armed with the model's classifications, software engineers can spend less time and resources searching for vulnerable code segments, scouring stack traces for crashes or unexpected behavior, or running an entire library through a fuzzer.

Our approach relies on two important hypotheses about the nature and structure of security vulnerabilities:

1. We hypothesize that there are certain kinds of code that are inherently more prone to mistakes and vulnerabilities and therefore need to be vetted more thoroughly for vulnerabilities.

2. We hypothesize that erroneous or vulnerable code possesses distinct features that enable a trained model to distinguish it from the non-erroneous and non-vulnerable code in a program.

Researchers have devoted significant time to addressing the first hypothesis, producing systematic reviews of the National Vulnerability Database (NVD)[1], the CVE dictionary, and other vulnerability-related data. These reviews aim to identify which vulnerabilities are most common in major codebases and which subsequently pose the greatest threat to systems around the globe. Among the many possible kinds of vulnerabilities, buffer overflows are widely recognized as the most frequent issue,

constituting 8.4% of all Common Weaknesses and Enumeration (CWE) reports filed in 2022. Buffer overflows have held their position as the #1 most dangerous software vulnerability for the last three years [47][48]. NVD statistics also report that 12% of all software vulnerabilities from 2008-2014 were composed of buffer overflows and their variations [15]. Overflows occur when a read or write command is issued to a memory location beyond the buffer length, allowing attackers to exploit the system by running unauthorized code, performing denial of service attacks (DoS), consuming company resources, and more. These exploits routinely cause severe damage to systems and as such, they are a key example of code that developers struggle to write securely. Subsequently, buffer-related code segments appear generally disposed to contain vulnerabilities (refer to Figure 1.1).

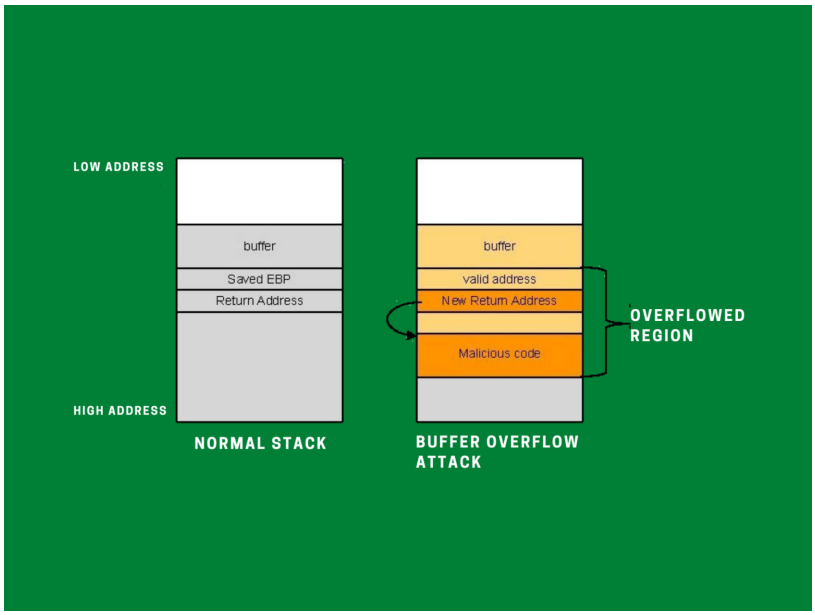

Figure 1.1: Buffer Overflow Attack [49]

Cross site scripting (XSS) (Figure 1.2) and access control / authorization issues (Figure 1.3) are also two of the most reported vulnerabilities [15]. Though not as common as buffer overflows according to CWE reports from 2015-2017, these two vulnerabilities accounted for 13% and 9% respectively of all NVD vulnerabilities from

2008-2014 [15]. Furthermore, NVD data suggests that "60 percent of all vulnerabilities [from 2008-2014] wouldn't have occurred" if programmers had prevented buffer overflows, XSS, and access control, as well as a few less common vulnerability categories (input validation, resource management, and SQL/code injections) [15]. This suggests that there are clear patterns in the kinds of vulnerabilities that plague existing programs. Therefore, such patterns are possible sections our model might identify and recommend to a developer as places in need of fuzzing.

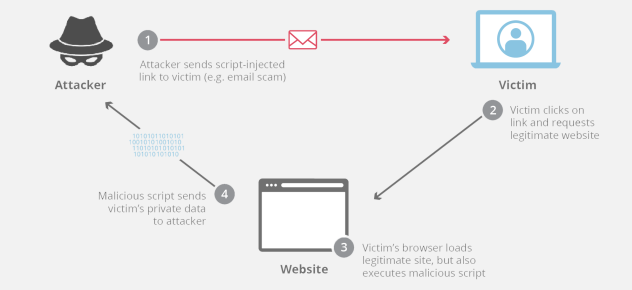

Figure 1.2: Cross Site-Scripting Attack [10]

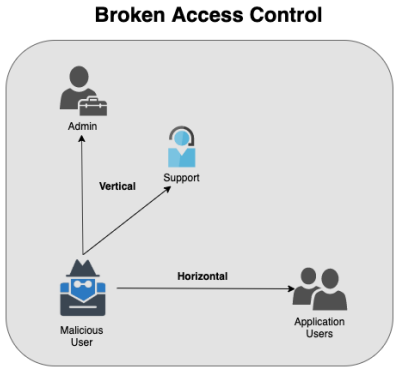

Figure 1.3: Access Control Exploit [31]

The second hypothesis – that vulnerable code can be differentiated from non-vulnerable code – is evaluated as a component of our experiment. We accomplish this by training a CodeBERTa model (a pre-trained language model that generates contextualized word embeddings for downstream Natural Language Processing tasks) perform a classification task on a CVE vulnerability dataset and examining the accuracy at which the model identifies vulnerable functions [18]. In doing so, our project will assess whether vulnerable code has a discernible style and whether natural language processing (NLP) can identify the semantics of a security defects.

Armed with these two hypotheses, our project ultimately evaluates the benefit of incorporating ML techniques into the traditional fuzzing approaches and strives to improve on the current techniques used to detect vulnerable code. It takes organizations, on average, 207 days to identify a data breach and another 70 to implement

a patch to contain or resolve that breach; it takes cybercriminals a fraction of that to exploit a small vulnerability inside even the most secure companies, at the cost of over 4 million USD per attack [16]. This discrepancy is the heart of the battle between software architects and hackers and one that continues to tilt in favor of the latter. However, the hybrid system of ML and fuzzing that we propose reduces the amount of time and manual effort required to detect and fix vulnerabilities by directing developers to the security mistakes, without the need for information beyond the code itself. In doing so, it helps ensure that vulnerabilities are discovered before they are exploited and enables developers to tip the scale back in their favor.

The contributions of this thesis are as follows:

1. We developed and expanded the only publicly available database of code that encompasses both vulnerable and non-vulnerable functions, facilitating novel research in software security and vulnerability analysis.

2. We conducted a comprehensive and critical survey of state-of-the-art fuzzing and machine learning technologies, offering insights into their strengths and limitations in detecting code vulnerabilities.

3. We presented a systematic analysis of code vulnerabilities and their corresponding patches, shedding light on the nature of "bugs" versus "flaws" in software code and providing valuable guidance for the development of more effective vulnerability detection tools.

4. We devised a novel and adaptable version of CodeBERTa, an advanced natural language processing model, that has been fine-tuned to identify vulnerabilities in code segments.

5. We proposed a novel approach to vulnerability detection that significantly reduces the time and resource costs of existing methods, enabling more efficient

and effective detection of software vulnerabilities, and paving the way for further research on defect detection in the face of increasing cyberthreats.

Chapter 2

Background

2.1 The Art of Fuzzing

Fuzzing is broadly defined as a process used by developers to discover security vulnerabilities in their program [30]. The importance of fuzzing lies in its crucial role in ensuring the security of software systems, as the repercussions of a single vulnerability can be enormous. As software systems continue to play a larger role in various aspects of life and business, there is an immediate need for effective methods to mitigate the risk of cyberattacks by identifying and remediating vulnerabilities before malicious actors can exploit them. Fuzzing was developed as a possible solution to this problem. Compared to manual detection techniques, fuzzing has clear advantages because it can systematically explore possible paths and inputs to the program, can be repeated multiple times, and can simulate real-world attacks [5].

The general fuzzing process is rather straightforward: an input is given to a program, a program is run, and developers wait to see if a core dump (the recorded state of a program's working memory at the time of a crash or abnormal termination) is created [53]. More specifically, all fuzzing techniques, despite many variations, consist of four main stages: testcase generation, program execution, runtime monitoring, and

crash analysis [30].

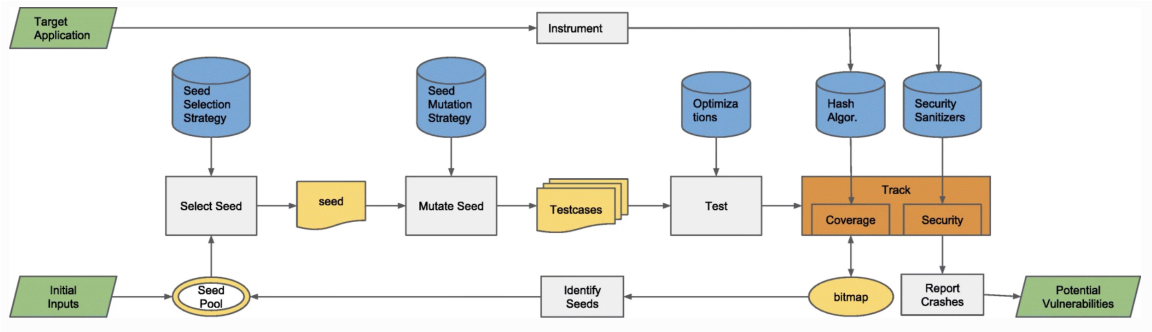

Figure 2.1: Working Process of Fuzzing [19]

The first of these stages, testcase generation, involves creating inputs for the fuzzing process. Inputs that "conform to the format of the program" are known as seed files and are used to generate a list of testcases that might reach a different part of the program, spark unexpected behavior, or trigger new vulnerabilities [53]. The goal of testcase generation is to create an expansive and diverse list of inputs; fuzzers then use this list to thoroughly test and evaluate areas of a program for security vulnerabilities or other errors. Input lists are generated using either a mutation-based generator, which provides new testcases by modifying known seed files, or a generation-based generator, which creates new testcases "based on the format information of the input sample without mutation" [53]. Both approaches are valuable depending on the program in question and as such, there is no universal heuristic to determine which approach is better for a given program. Miller and Peterson's empirical analysis of the two methods found that generation-based fuzzing can execute 76% more code than mutation-based fuzzing but also note that the mutation-based fuzzer was at a significant disadvantage due to a lack of diverse files for testing and may perform better in a different experiment [26]. Given the lack of a definitive answer as to which approach is superior, many fuzzers employ genetic algorithms to deter-

mine the optimal method for maximizing code coverage and increasing the chances of detecting potential vulnerabilities.

The next two stages focus on running the target program. Using the inputs produced in the testcase generation stage, the target program is executed within the program execution stage. Upon running the program, the state of the program is closely monitored to collect information that will be "fed back into the testcase generation stage to guide the next generation of testcases" [53]. This is known as the runtime monitoring stage. If a target program crashes, reports an error, or becomes stuck in an infinite loop, the behavior is recorded and collected for later replay and analysis.

The fourth and final stage, crash analysis, examines data reported from the runtime monitoring stage to determine whether the crash or error is a bug or some other kind of issue. If the crash is determined to be a bug, the fuzzer performs an exploitability analysis to decide if the bug is a vulnerability [53]. Developers then use this information to manually debug the program for the vulnerability and confirm or reject the fuzzer's findings. Although the fuzzing process cannot resolve the security vulnerabilities it finds, it assists developers in identifying the presence of an exploit – which can be immensely difficult when working with large, interconnected codebases.

2.1.1 Strategies, Types, and Limitations

Traditional fuzzing approaches are relatively simple. While the original fuzzing process was able to identify some of the software bugs hiding in the code, it was not as exhaustive as it needed to be. Limitations of this approach (discussed in greater depth below) included an inability to mutate seed inputs, improve code coverage, or bypass the validation required to reach certain parts of a given programs [53]. Three new analysis techniques, static analysis, dynamic analysis and symbolic execution, were eventually developed to help alleviate some of these limitations.

Static Analysis

Static analysis is defined as the "analysis of programs that is performed without actually executing the programs" [19]. Because the target program is never run, this kind of fuzzing is performed mostly on source code or occasionally on object code. Source code refers to the human-readable instructions that are created by a developer while object code broadly refers to the output of compiled file that consists of machine-readable statements processed by the CPU. Fuzzers using static analysis examine the lexical, grammar, and semantic features of these code inputs and perform data flow analysis and model checking to identify vulnerabilities. The purpose of data flow analysis is to identify how data is used in a program and how it propagates from one point to another. This makes it particularly effective at identifying buffer overflows, data leakage, and other data dependent bugs. Alternatively, model checking compares the program's behavior against a formal model of its expected behavior [41]. This automated method individually reasons about all possible execution paths of a program in order to verify that a program satisfies the specified requirements and declare the program free of any concurrency issues [41]. Each of these processes gives static analysis a high detection speed and allows developers to quickly check the target code for bugs. However, this speed comes with a considerable tradeoff in terms of accuracy. Without an "easy to use vulnerability detection model," static analysis tools routinely report large quantities of false positives and are therefore difficult to use in practice.

Dynamic Analysis

Conversely, dynamic analysis relies on execution to detect vulnerabilities. Target programs are executed in real systems and emulators, which allows developers to monitor the running states and analyze runtime behavior to find vulnerable code. This method of searching for bugs is highly accurate when compared to static analysis,

but its accuracy comes at the cost of high human interaction [19]. To properly debug, analyze, and run target programs, developers must have strong technical skills and dedicate significant effort to hunting for vulnerabilities. For large-scale testing at the level most companies require, dynamic analysis is difficult to scale, time consuming, and complex.

Symbolic Execution

Symbolic execution also struggles with scalability. By symbolizing program inputs so that a set of constraints are maintained for each execution path, this approach works backwards to determine what kind of inputs spark crashes or other kinds of unexpected execution behavior. It accomplishes this by employing constraint solvers after execution to solve the constraint and use this solution to identify which seed inputs result in the given execution path [19]. In theory, symbolic execution could cover all possible execution paths for a given programs; in practice, however, the number of possible paths explodes as the scale of the program grows and at some point, the number exceeds the solving capacity of constraint solvers. Furthermore, interactions between the target program and elements of the execution environment outside of the constraints established by symbolic execution such as system calls and signal handling can create issues with consistency. These issues, combined with the path explosion problem, make symbolic execution suitable only for small programs or programs with few execution paths and minimal environment interactions.

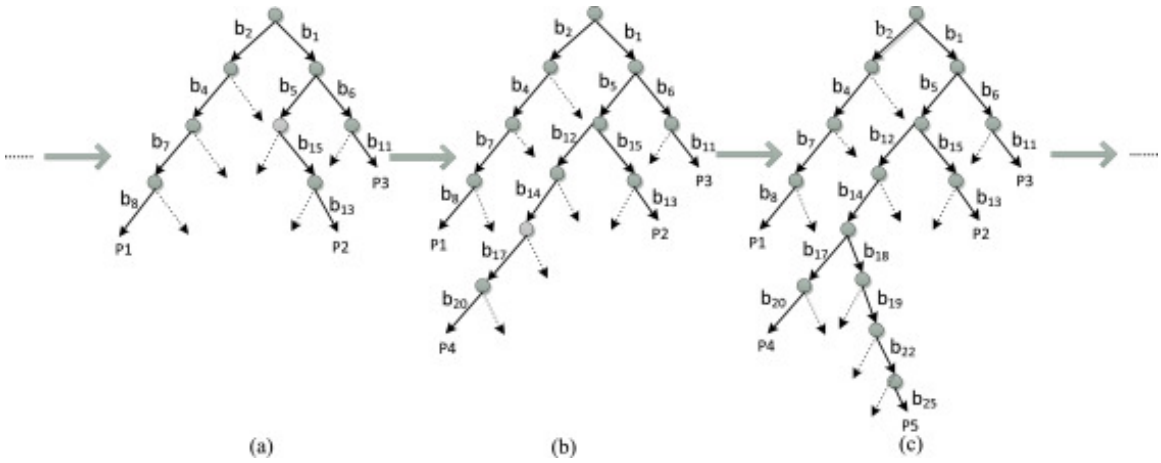

Figure 2.2: Path Explosion Problem For Three Nodes [37]

Directed vs. Coverage-Based Fuzzing

Another important classification in fuzzing approaches is the strategy of program exploration. Fuzzers investigate target programs either through directed fuzzing or through coverage-based fuzzing, depending on the goal of the fuzzing process [19]. Directed fuzzers focus on testcase generation to cover particular parts the target code and target paths of a given program, while also using additional information about a program (bug stack traces, patches, risky operations, etc.) to test specific parts of the code [28]. Alternatively, coverage-based fuzzers focus on accessing as much of the target program as possible, analyzing program behavior and altering inputs to maximize code coverage. Each of these approaches has advantages – directed fuzzers are considerably faster and coverage-guided fuzzers are more thorough and detect more bugs – but each also struggles to extract information from executed paths [19]. Consequently, both strategies are minimally successful at mutating and generating testcases capable of spanning the entire program and triggering vulnerabilities.

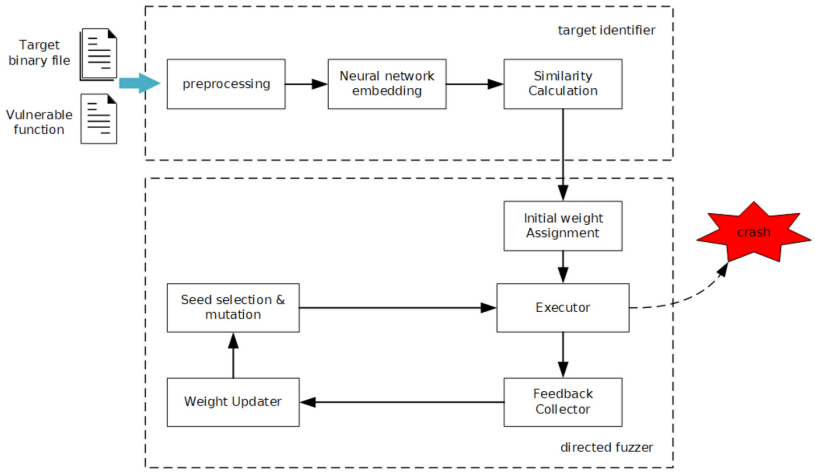

Figure 2.3: Working Process of Directed Fuzzing [34]

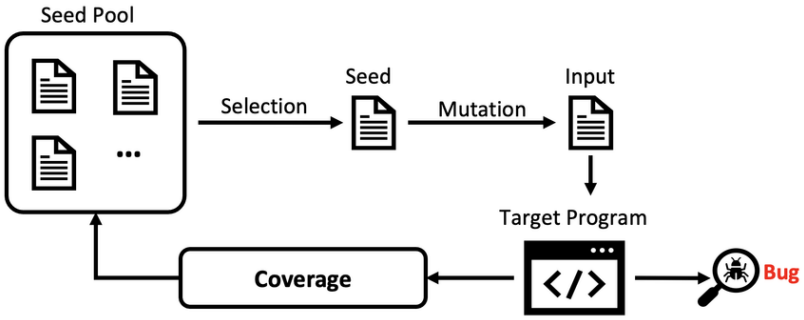

Figure 2.4: Working Process of Coverage-Based Fuzzing [11]

Mutation-Based Fuzzers

Gathering data from the paths taken during execution and using it to inform test-cases is one of the key challenges state-of-the-art fuzzers face when put into practice. Mutation-based fuzzers, for example, must decide where and how to mutate inputs to alter the control flow of the program. Therefore, locating the critical positions in a program where mutating affects the execution path is incredibly important and one reason why blindly mutating the seed input is a waste of resources and time [26]. It also suggests that the inclusion of a model to better direct fuzzing efforts towards parts of the program (and therefore making it more clear what kind of seed inputs need to be generated and where/how they should be mutated) could significantly

improve the efficiency and effectiveness of fuzzing.

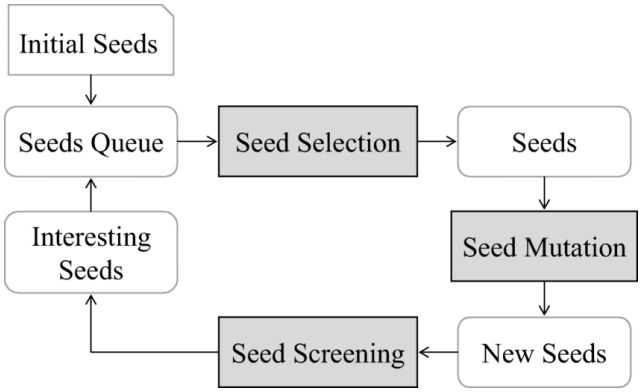

Figure 2.5: Working Process of Mutation-Based Fuzzing [44]

White, Gray, and Black Fuzzers

Outside of directed, coverage-based, and mutation-based fuzzing techniques, there are three primary types of "fuzzers" that vary from one another in terms of their dependence on source code and the degree of program analysis (refer to Figure 2.6) [30]. The types are white box, gray box, or black box. White box fuzzers have complete access to the source code of target programs and can therefore gather more insight about how testcases affect the target program's running state. Black box fuzzers, on the other hand, do not have any information about the target program's internals. Instead, black box fuzzers generate potential seed inputs from scratch and mutate them without direction from the source code. Grey box fuzzers also operate without access to source code, but they do have access internal program information through program analysis. As such, they are neither as blind as black box fuzzers nor as unrestricted as white box fuzzers [19]. The type of fuzzer used by developers depends on several factors, including the size of the target, the architecture of the library being searched for vulnerabilities, and whether the target is deterministic. Additionally, there may be other factors that limit an analyst's ability to access or utilize source code information.

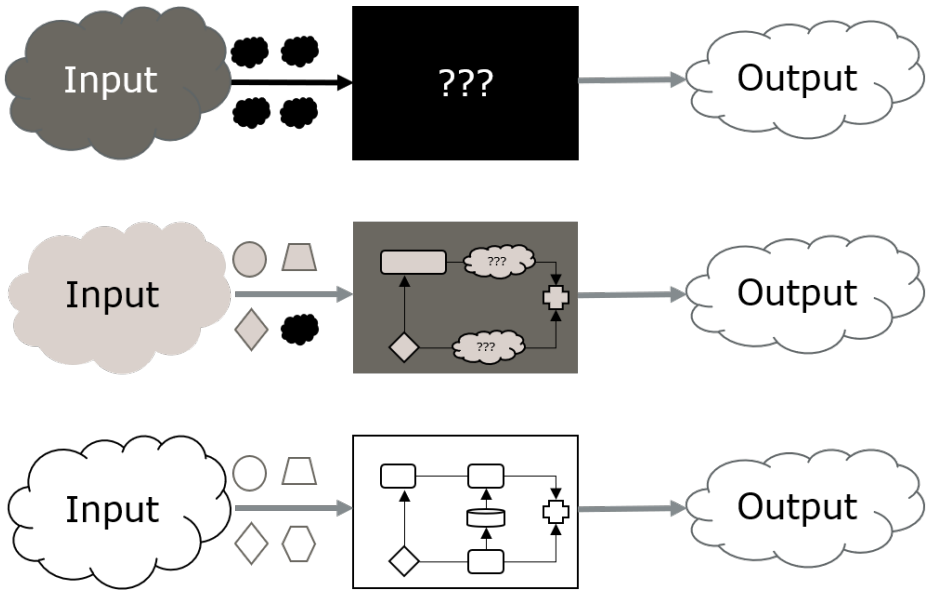

Figure 2.6: Types of Fuzzers [17]

Enormous State Space

The issue of state space in fuzzing refers to the challenge of efficiently exploring all of the possible states or conditions that a program can assume during its execution. A program's state space is defined as the collection of all the possible inputs, control flows, and data structures that can be encountered during program execution [3]. In order to effectively fuzz a program, it is necessary to explore the state space thoroughly to ensure that all possible paths are covered and to focus on critical and high-risk areas of the state space, while avoiding less important or uninteresting areas that may consume a lot of resources without yielding useful results. However, as the state space of programs grows exponentially, it can become infeasible to exhaustively investigate the entire state space [3]. This issue is compounded by the fact that current fuzzing approaches struggle to identify high-risk areas, especially in large and complex programs, as noted by Aschermann et. al [3]. Therefore, despite the significant progress made in the development of fuzzing techniques, the challenges of exploring the vast state space of complex programs and identifying high-risk areas

remain significant shortcomings of current state-of-the-art fuzzers.

Low Code Coverage

Another significant problem traditional fuzzers face is the issue of low code coverage. It is generally assumed that higher code coverage leads to a greater number of program execution states and therefore a more thorough testing of a target program. An important caveat to this assumption is that generating large numbers of testcases is not a sufficient solution to increasing code coverage. To aptly test all parts of a program, testcases capable of reaching "distant" or hard to reach sections of the code must also be created [19]. This requires a thorough understanding of the target program and extensive resources that can all be consumed by fuzzing, both of which can be difficult to achieve on a large scale and on a constant basis.

Validation Checks

Furthermore, even if traditional fuzzing approaches were able to generate a substantial list of testcases capable of reaching difficult parts of the program, there is another obstacle fuzzers must overcome: passing validation. Most complex programs validate inputs to protect against targeted attacks (or, even just to prevent a program from crashing) in the form of maliciously crafted strings, number sequences, and so forth. As a consequence of this validation process, invalid testcases are often ignored or thrown out from execution altogether – even in the case of testing for vulnerabilities. Consequently, possible vulnerabilities within a program could be left undiscovered [19]. For blind black box and grey box fuzzers, producing testcases that pass a program's validation checks can be incredibly difficult and results in inefficient and ineffective fuzzing.

Multiple methods have been proposed as countermeasures or even solutions to these limitations – most of which were discussed above in terms of approaches, dif-

ferent kinds of fuzzers, etc. – but the problems plaguing fuzzing techniques remain. Although certain developments have made modern fuzzers more flexible and detailed, these developments operate at the expense of time and resources and by making tradeoffs between code coverage, efficiency, and quality. Not all codebases and target programs are compatible with techniques like coverage-based fuzzing or a white-box/grey-box approach, and not all companies have the financial and logistical capacity to dedicate large amounts of funds, manpower, and hours to performing several fuzzing tests. As such, there is a demonstrated need for an easier, faster, and less human involved approach to fuzzing and eliminating even the most elusive of security threats.

2.2 Machine Learning and CodeBERT

Machine learning (ML) is a process by which a model "acquires new knowledge or skills by learning from existing example data or experiences" [51]. It is a branch of artificial learning that attempts to imitate human learning patterns and has a multitude of applications ranging from self-driving cars to medical diagnosis software to natural language processing. Most scholars divide machine learning tasks into three categories (traditional machine learning, deep learning, and reinforcement learning), each of which has specific subcategories of learning types that fall under their jurisdiction. In the context of this experiment, only deep learning is relevant.

Deep learning is a subset of machine learning that relies on layered, artificial neural networks modeled after the brain to analyze data in a human-like way [6]. By passing information between interconnected nodes that are inspired by neurons, deep learning models can process large amounts of data, identify complex patterns, and make intelligent decisions independently. The decision-making process is designed to learn and improve from experience without a developer needing to explicitly program it to

do so, rendering deep learning models ideal for tasks like image classification, speech recognition, and natural language processing [6]. Furthermore, the training process for deep learning models typically involves adjusting model parameters to minimize a loss function, which measures the discrepancy between the model's predictions and the actual outputs [6].

2.2.1 Learning Paradigms

The kinds of algorithms and data input into the model determines which of the following learning methods the model will use: supervised learning, unsupervised learning, or semi-supervised learning [23]. Supervised learning occurs when a model is trained on a labeled dataset, meaning the model can learn, grow, and eventually predict observations based on the inferred relationship between the input and output variables [23]. In addition to these predictions, supervised learning gives the model sufficient information to learn from labeled examples where the correct outputs are provided for a given set of inputs. By comparing its output with the correct intended output, the model can calculate its prediction errors and adjust its parameters to improve its performance over time.

Alternatively, during unsupervised learning, unlabeled training sets and data are used to teach the model to search for hidden patterns [23]. Unsupervised learning is unique because it creates inferences and connections about non-obvious structures in the data, whereas supervised learning predicts observations and compares its output with the intended output. These two approaches are combined in semi-supervised learning, a technique that uses both labeled and unlabeled data for training purposes. Semi-supervised learning equips models with the ability to draw inferences and connections, in addition to the ability to predict, by utilizing both a small number of labeled examples and a primarily unlabeled dataset [23]. Despite the benefits of unsupervised and semi-supervised learning, supervised learning is the most common

and widely employed method, and the method that we utilize for training our model.

There are several examples of deep learning models that rely on supervised, unsupervised, and semi-supervised learning [39]. Inception-v3, ResNet, and VGGNet are three state-of-the-art supervised learning models that were created to perform image classification tasks, categorizing images into two classes similar to the way this project's model categorizes code segments into classes [39]. Furthermore, unsupervised learning models such as Restricted Boltzman Machines (RBMs) and Generative Adversarial Networks (GANs) have been used for a range of tasks including dimensionality reduction, feature extraction, and generative modeling [39]. Semi-supervised learning models tend to combine the objectives and architecture of the above models, performing classification tasks like image classification, feature learning, and face recognition within a generative modeling framework [39].

2.2.2 BERT-Based Models

Another important element of the machine learning techniques discussed in this project is our use of CodeBERT and its variations. CodeBERT is classified as a "bimodal extension of Bert," which indicates that the model has modified to better suit the distinctive features and complexities involved in processing programming languages [7]. In more technical terms, CodeBERT operates by fine-tuning a transformer-based language model that has been pretrained on a collection of programming and natural language text [7]. Pre-training is accomplished via a Masked Language Modeling (MLM), meaning the model is trained to predict masked tokens – both for programming and natural language tokens – in a given input sequence. Masked tokens refer to the randomly selected input tokens (usually around 15% of all input tokens) that are replaced with a special [MASK] token [7]. The model is subsequently trained to predict the original value of the masked token based on the surrounding context. In doing so, BERT models learn to understand the relationships

between words and phrases in an input sequence.

One relevant variation of CodeBERT is called RoBERTa, which stands for Robustly optimized BERT approach [21]. Similarly to CodeBERT, RoBERTa was trained on a large corpus of text using a standard MLM objective. Unlike CodeBERT, however, RoBERTa was trained on a greater amount of data and for a longer duration, resulting in improved performance for most natural language processing tasks [21]. These models are both based on BERT architecture and use similar MLM processes for training, but they differ in their training data and objectives and are therefore better suited for different tasks. Such tasks include language modeling, sentiment analysis, and question-answering for RoBERTa and code completion, code search, and code summarization for CodeBERT [21].

The model implemented in this study attempts to combine both the programming and natural language processing skill of CodeBERT and RoBERTa into one model, known as CodeBERTa. CodeBERTa is a variant on the RoBERTa architecture that was pre-trained on a large corpus of natural language and then adopted to the domain of programming language. As such, the model possesses the key capabilities of both CodeBERT and RoBERTa. When it comes to vulnerability detection, the primary objective of our CodeBERTa model is to learn to predict which elements of code are "vulnerable." This prediction task translates to the model classifying source code as a "1" if it is vulnerable or as a "0" if it is most likely free of known security threats. A detailed visualization of CodeBERT's base model is included below (Figure 2.1).

2.3 Approach

When security engineers inspect code for potential vulnerabilities, they often lack a comprehensive understanding of the code's level of insecurity. They may have some general ideas about areas of potential weakness, such as the author's reliability, the

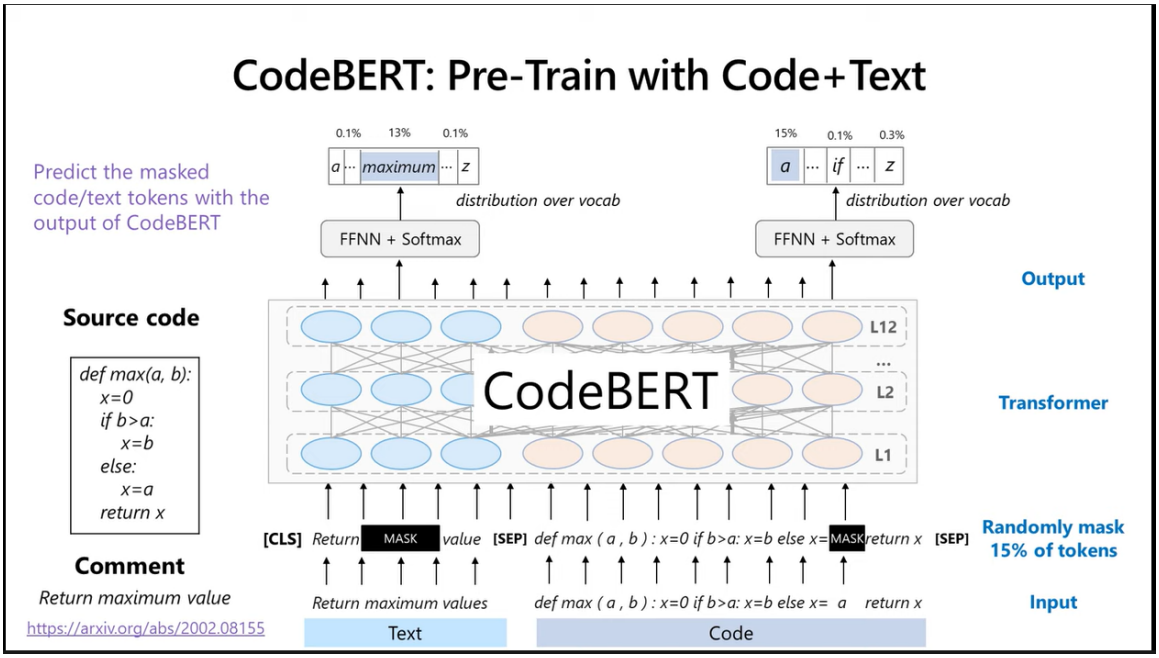

Figure 2.7: Visualization of CodeBERT architecture [14]

code's architecture, and functions that are prone to bugs but without performing an exhaustive analysis and testing, it is difficult to assess the code's security status accurately. Currently, there is no easy way to assign a vulnerability score to code or assess the security status of code by assigning it a rating on some kind of gradient. Developing this kind of tool requires significant amounts of data collection and computational resources that exceed the resources allocated for a project of our size.

However, a key component of developing such a system is the creation of a vulnerability classification model.In order to assign scores or ratings to code based on the severity of identified vulnerabilities, a model must first be able to determine if a vulnerability is present or not. By training a BERT-based model to perform binary classification of code segments, our project aims to lay the foundation for creating a vulnerability scoring system or gradient and ultimately provide faster and more efficient methods for defect detection.

Chapter 3

Related Work

3.1 The Status of Machine Learning

The success of machine learning (ML) in processing data and pattern recognition has encouraged researchers to consider its potential use case in the field of cybersecurity, especially in the area of vulnerability detection. Recent works examine the effectiveness of various CodeBERT models in software defect prediction and explore whether the models vary in terms of prediction performance, or if they are capable of vulnerability detection at all [32]. The findings of Pan et. al's comparative study on CodeBERT models suggest that BERT-based models can successfully identify vulnerabilities within code [32]. Specifically, pre-trained CodeBERT models performed better statistically in this study in terms of prediction performance and therefore could be utilized to reduce the amount of time, resources, and money required of largescale vulnerability detection operations [32].

Some groups advocate for alternative approaches for vulnerability detection, employing graph neural network-based models rather than the NLP techniques of CodeBERT [54] and even developing a new detection models, such as ML-FEED [38]. For instance, Zhou et. al employs gated graph recurrent layers (derived from [20]) to gen-

erate a three-layer model called Devign that accounts for the semantics and structural features of nodes [54]. This architecture allows the Devign model to represent code as a control flow graph and subsequently use its neural networks to learn the relationships between different parts of the graph. Ultimately, the tool was found to be highly effective in identifying vulnerabilities, thereby suggesting that ML could be used by software developers and security researchers to identify potential vulnerabilities that may have been missed by traditional static analysis techniques [54].

Alternatively, the architects of the ML framework ML-FEED focus less on improving accuracy of vulnerability detection and more on optimizing long short-term memory networks (LSTMs) and transformer-based exploit detection models [38]. The model operates by analyzing network traffic and using the information it derives to detect attempted exploits as they occur. Like the Devign model, the ML-FEED framework was able to identify security breaches in real time with high accuracy and low false positives [38]. Though the results suggest ML-FEED can help improve the security of computer networks and prevent cyberattacks, the study has shortcomings in terms of a limited testing dataset (it only includes 79 exploits and evaluates three CVE attack traces) and a language constraint of only Java-based programs [38] (which tend to be more secure by default, at only 12% of all vulnerabilities reported in 2020) [52] that make it difficult to determine its comparative accuracy outside of the test conditions.

A separate line of work is dedicated to incorporating ML into fuzzing. Many security researchers believe that ML technology can minimize the severity of the challenges discussed previously regarding fuzz testing (refer to Section 2.1.1). Areas such as testcase generation, testcase execution, and bypassing format or validation checks have been of particular interest.

The most common ML technique for input generation is the use of genetic algorithms (GAs), a type of unsupervised learning inspired by biological evolution that

acts as the core input generation algorithm in evolutionary fuzzers [36]. The steps of using GAs are as follows: 1) generate a small base population of inputs, 2) perform the necessary transformation on these inputs, 3) measure the fuzzer's performance and adjust inputs as necessary. By building off of previously successful inputs and mutating new seeds accordingly, GAs can increase code coverage and assist developers in reaching unexplored code paths that can then be fuzzed for vulnerabilities. Fuzzing techniques such as this typically rely on a fitness function to rank inputs for selection and mutation. Though there are many reliable and useful fitness functions, choosing which function to implement can have a tremendous impact on a fuzzer's performance, ability to find certain bugs, and tendency to get stuck in local minima. The developers of the Dynamic Markov Model (DMM), a statistical model that considers the underlying dynamics of a system when modeling probability, attempted to bypass this selection process by introducing a new metric. Rather than measuring success via code coverage, the DMM incorporates transition probabilities into the program control graph and uses these probabilities to create a unique fitness function that allows for more precise control over the fuzzer [45].

Deep learning and neural network graphs have also been utilized for input generation, especially in generation-based or mutation based fuzzers. For example, Rajpal et al. incorporated deep learning into the American Fuzzy Lop (AFL) fuzzer, using a neural network to generate a heatmap of predicted code coverage for a particular byte if that byte were to be mutated [4]. Similarly, the NEUZZ method modeled program behavior through a collection of shallow neural networks [42]. NEUZZ networks were trained to predict program paths as a seed input was mutated, generating a tree-like structure that developers could use to evaluate code coverage and guide fuzzing efforts. Both examples found that the inclusion of machine learning techniques outperformed standard fuzzers in a majority of cases. In particular, Rajpal et. al concluded that their deep-learning version of AFL was superior to traditional AFL

in all input formats except PNG (a kind of image file format) [4] and the NEUZZ method demonstrated significant experimental improvement compared to state-of-the-art taint fuzzers such as AFL and Angora. [42].

In addition to resolving the challenge of input generation, overcoming the path explosion problem and high computational cost of symbolic execution has also been primary goal of researchers. However, despite being a key area of interest, current efforts in this field are primarily focused on testing the feasibility of incorporating ML into fuzzing techniques and are not yet on par with the latest graph algorithms. For instance, ML was used to solve constraint equations in two separate studies [40][43] but was not as efficient as non-ML constraint solvers in either example. In the first study, Selsam et. al developed a message passing neural network called NeuroSAT that can solve propositional satisfiability problems (SAT), a class of problems that involve determining whether a given Boolean formula can be true or not. [40]. The NeuroSAT model is trained as a classifier to predict the satisfiability of a given Boolean formula by using a dataset of randomly generated SAT problems [40]. Though this method did not "beat" modern constraint solvers, the primary takeaway of the study is that NeuroSAT could solve equations beyond the domains it was trained on. Therefore, the model is presumed to be capable of generalization. The ability to generalize implies that NeuroStat and possibly all neural networks could be used to perform discrete searches without hard-coded search procedures or intense amounts of training/supervision, which would subsequently reduce the time and effort required to solve complex problems in fields such as artificial intelligence, optimization, and decision-making [40].

In the second study, Shiqi et. al adopted a similar approach by implementing neural networks as representations of constraint equations [43]. The neural networks are trained on patterns in the behavior of a program and then taught to use these patterns to guide the symbolic execution process and find solutions to the equations

via a gradient descent [43]. While Shiqi et. al conclude that this neural network approach can be effective in resolving constraint equations and analyzing programs with complex data structures, the network required significant computation time to do so and therefore could not compete with modern constraint solvers [43]. Still, both studies lay a strong foundation for using supervised learning and other ML techniques to answer constraint equations and to minimize the limitations of symbolic execution fuzzers.

3.2 Defect Detection with Machine Learning

The idea that machine learning models could be used to automate the process of vulnerability detection is not new. Researchers at the University of Central Florida extended traditional black box fuzzing (refer to Section 2.1.1) to include elements of the Dynamic Markov Model (DMM) algorithm [45]. The DMM algorithm is used to generate a control flow graph capable that can intelligently guiding a "blind" black box fuzzer to potential code defects via evolutionary input crafting [45]. During evolutionary input crafting, developers define a set of program properties and behaviors that all possible inputs have to satisfy and evaluate the quality of generated inputs via the DMM fitness function. The best inputs are chosen for the next generation of inputs, and the process repeats [45].

The results of this study were not measured in terms of traditional machine learning metrics – such as accuracy, F1-score, or PR-AUC and ROC-AUC – but rather in terms of code coverage and graph evolution. Compared to random input generation (as is common with standard black box fuzzing), the genetic algorithm nearly doubled code coverage (84.81% vs. 49.54%) while only requiring a fraction of the input generations to reach a given depth in the control flow graph [45]. Therefore, The study's ML approach is more effective than random black box fuzzing for creating

diverse and effective test inputs and contributes to the belief that ML can save time and resources when identifying software vulnerabilities [45].

Angora, a newly developed mutation-based fuzzer, is another example of using machine learning techniques for software bug discovery [8]. Unlike symbolic execution fuzzers (which produce quality inputs but with enormous time costs) and random mutation fuzzers (which are fast but struggle to create effective inputs), Angora uses ML to solve path constraints and selectively mutate inputs based on their effectiveness in achieving high code coverage [8]. More specifically, the Angora fuzzer uses an ML gradient to solve constraint problems by training a gradient boosting model to predict the likelihood of a constraint being satisfiable based on the program's behavior [8]. When Angora generates a new input that violates a constraint, it uses the trained model to predict the likelihood of the constraint being satisfied by the mutated input. If the model predicts a high probability of the constraint being satisfied, Angora uses the mutated input as the basis for further mutations. By using an ML gradient to predict constraint satisfaction, Angora is able to more efficiently guide its search for inputs that trigger new paths and achieve high code coverage while avoiding the issues that plague symbolic execution and random mutation fuzzers [8]. When evaluated on a range of programs – including those anticipated to have vulnerabilities and those expected to be "free" of bugs – Angora routinely outperformed AFL in terms of bug detection, line coverage, code coverage, and generation of shorter and more effective inputs [8]. Therefore, the fuzzer is a prime example of the advantages of incorporating machine learning techniques into traditional fuzzing mechanisms.

3.3 CodeBERT and Vulnerability Identification

CodeBERT was originally introduced as a pre-trained model for natural language (NL) and programming language (PL). It was designed to capture the semantic con-

nection between NL and PL and to use this connection to support tasks such as code documentation generation, natural language code search, and more [14]. In 2020, Feng et. al set out to evaluate the effectiveness of CodeBERT and RoBERTa on several benchmark datasets for programming and natural language processing tasks. The goal of this evaluation was to demonstrate the potential of these models to improve the effectiveness and efficiency of a wide range of tasks, including bug detection [14]. For both programming tasks and natural language processing tasks, the authors used several benchmark datasets, including CodeCompletionNet, Codeforces, and CodeSearchNet (PL), and GLUE, SuperGLUE, and SQuAD (NL). They then fine-tuned the models on these datasets for each task. The results of this study demonstrated that CodeBERT outperformed other state-of-the-art models on several programming tasks, including code completion and bug detection. For example, CodeBERT achieved an accuracy of 79.4% on the CodeCompletionNet dataset, which was significantly higher than the next best model at 72.9% [14]. Similarly, CodeBERT achieved an F1-score of .575 on the CodeSearchNet dataset compared to the second highest model's F1-score of .517 [14]. RoBERTa also outperformed other state-of-the-art models on several benchmarks, including GLUE and SuperGLUE, with an accuracy score of 89.3% [14]. Although the task performed in our project does not directly involve NL and PL probing, the results of this experiment suggest that both models can greatly improve the effectiveness of NL/PL processing tasks and provide an important baseline in terms of performance for our CodeBERTa model.

Previous efforts have also been made to incorporate CodeBERT and its variations into vulnerability detection methods. Empirical analyses performed on various CodeBERT models (CodeBERT-NT, CodeBERT-PS, CodeBERT-PK and CodeBert-PT) determined that neural language models could improve prediction performance for cross-version and cross-project defect prediction [32]. Under the conditions of this analysis, each model was given three prediction patterns (binary, sentence-based, and

keyword-based) and evaluated on its ability to identify defects from the cross-version PROMISE source code (CVPSC) and cross-project PROMISE source code (CPPSC) databases [32]. For these two datasets, CodeBERT-PS reported an average F1-score of 0.616 and 0.570, CodeBERT-PK reported an average F1-score of 0.631 and 0.551, CodeBERT-PT reported an average F1-score of 0.565 and 0.519 and the RANDOM model reported an average F1-score of 0.397 and 0.402 [32]. Based on these metrics, it can be inferred that CodeBERT has the ability to extract both semantic and syntactic information from source code and can predict vulnerabilities with a performance that exceeds that of random guessing.

Chapter 4

Materials and Methods

Our CodeBERTa model was trained, tested, and evaluated on a remote GPU server with 200GB of local storage space and SLURM system for queue management.

4.1 Data

Publicly available datasets of code vulnerabilities are scarce, with only a few existing repositories. Though the MITRE corporation has made a catalogue of cybersecurity vulnerabilities public [27] and the National Vulnerability Database (NVD) provides a description of all Common Vulnerabilities and Exposures (CVEs) published each month [1], concrete details of these bugs are omitted. In other words, neither MITRE nor NVD maintains a database that contains the vulnerable code itself. Therefore, the dataset used to train our CodeBERTa model was acquired in two ways. The initial data was adopted from a dataset curated by researchers at Nanyang Technological University in 2019 for use in a neural network model called Devign (refer to Section 3.1 for details) [54]. This dataset combines source code segments from four popular open-source libraries (Linux, FFmpeg, Qemu and Wireshark) that were collected by a team of security experts and manually labeled as "non-vulnerable functions" or "vulnerable functions" in an effort that took 600-man hours to complete [54]. This

dataset included 14,858 examples of non-vulnerable functions and 12,460 examples of vulnerable functions.

One of the shortcomings of this dataset, however, is that vulnerable functions are not matched with their respective patches. Though there are many valid examples of vulnerable or non-vulnerable code segments in the database, vulnerable commits were not matched with their corrected versions. As such, an additional 102 Linux-kernel source code examples were added to this dataset from NVD-identified CVEs. These code segments were sourced manually from GitHub and other version control platforms and include both the vulnerability and its patch. After formatting the code segments to adhere to the existing data schema and eliminating extraneous functions, the examples were appended to the database. The expanded database now includes 14,909 examples of "clean" code and 12,511 examples of "buggy" code and is one of the project's primary contributions (as noted in Section 1). Moreover, the newly added testcases not only provide a crucial set of examples for the model's training but also contribute to legitimizing the study's outcomes by ensuring that the model is evaluated on its capacity to differentiate between two nearly identical code segments. The table below provides more details on the dataset and its components.

Table 4.1: Dataset Statistics

Operating System	Number of Examples
Qemu	17549
FFmpeg	9769
Linux Kernel	102

Code Vulnerability	Number of Examples
Vulnerable	14909
Non-Vulnerable	12511

Set ID	Number of Examples
Training	19194
Testing	7403
Validation	823

4.1.1 Pre-Processing

Before training the model on the updated dataset, the data was preprocessed to fit the format required for the pre-trained RoBERTa transformers. The dataset was cleaned of unnecessary whitespace and indentation, and each row was checked for duplicate functions. Because CodeBERT models cannot process input sequences greater than 512 tokens in length without extensive alteration and computational power, all tokenizers were set to the maximum sequence length of 512 (refer to table 4.2). Though some implementations of CodeBERT and its versions uniform inputs in lower cases [33], capitalization within the code segments was maintained due to C programming language's case sensitivity.

Table 4.2: Input String Statistics

Token Count	Number of Examples
Greater than 512	18337
Less than 512	9083

	Average	Minimum	Maximum
Input Sequence Length	1912	21	134621

The 27,420 datapoints included in the dataset were split into training, testing, and validation sets on a ratio of 70/25/5. The 70/25/5 split is a well-established choice in ML because the ratio allocates a significant portion of the data to the training set, which enables the model to learn from a more extensive range of examples. Furthermore, the smaller testing and validation sets provide a more thorough evaluation of the model's generalization capacity. Given the complexity of the classification task at hand, these advantages are crucial in constructing a more robust and effective model.

4.2 Model Architecture and Hyperparameters

The base, pre-trained version of CodeBERTa was the model utilized at the start of our experimental process. This model was implemented using PyTorch and HuggingFace Transformers library, and its weights were pulled from Microsoft's CodeBERTa hugging face repository [25].

The base CodeBERTa model included a weight decay function that penalized large weights in the model during training to help it generalize better to unseen data and used the AdamW optimizer during the training process. However, the base version did not include early stopping mechanisms or a learning rate scheduler. Given how important these components can be for optimizing the model's training performance and preventing overfitting, these two elements were added to the base code. During

the training sequence, if the model's performance (defined in terms of model loss) did not improve significantly for three consecutive epochs, training was stopped early and the best model was returned. We chose to prioritize the loss function over accuracy due to the nature of the classification problem and because accuracy measures do not distinguish between false positives and false negatives.

The learning rate for the model was set to 2e-5, which means the the optimizer will adjust the model's parameters by a fraction of 2e-5 times the calculated gradient during each training iteration. In doing so, the model should converge faster and achieve better performance. The learning rate scheduler programs the learning rate to linearly increase from 0 during the warmup steps and linearly decrease to 0 thereafter for the remaining training steps. The number of warmup steps was concretely set to 1000 while the number of training steps varies based on the input parameters passed into the model pipeline, such that:

$$\text{num\_training\_steps} = \text{len}(\text{train\_dataset}) \times \text{args.num\_train\_epoch}. \tag{4.1}$$

The other hyperparameters in the training and testing pipeline were set as follows.

```
python run.py \
    --output_dir=./saved_models \
    --model_type=roberta \
    --tokenizer_name=./saved_models/tokenizer_bert \
    --model_name_or_path=./saved_models/model_bert \
    --cache_dir=./saved_models/ \
    --do_train \
    --train_data_file=train.json \
    --eval_data_file=validate.json \
    --test_data_file=test.json \
    --epoch 100 \
    --block_size 512 \
    --train_batch_size 32 \
    --eval_batch_size 64 \
    --learning_rate 2e-5 \
    --max_grad_norm 1.0 \
    --evaluate_during_training \
    --seed 123456 \
    --warmup_steps 1000
```

Figure 4.1: CodeBERTa Model Hyperparameters

4.3 Evaluation Metrics

Following ML standard, the model's performance on the vulnerability detection task is evaluated using the following five benchmarks. The initial three metrics are thresholded, as they depend on a classification threshold of 0.5. For our binary classification problem, a classification threshold of 0.5 means that if the probability of the input belonging to the "vulnerable" class is greater than or equal to 0.5, the model will classify it as "vulnerable". Similarly, if the probability of the input belonging to the "vulnerable" class is less than 0.5, the model will classify it as "non-vulnerable". In contrast, the latter two metrics are thresholdless.

4.3.1 Accuracy

Mean accuracy is a standard method of evaluating the accuracy of a binary classification model. In a binary classification task – such as classifying code as "1" if vulnerable or "0" of non-vulnerable – the model is trained to predict which class a sample belongs to. Its accuracy is subsequently calculated by comparing the predicted labels with the true labels in the dataset [33]. While this metric does not represent the nuances of a model's performance or encapsulate all the factors that alter a model's behavior, it is a simple and easy-to-understand measure of overall performance. Consequently, it can be used to compare accuracy across models or evaluate the performance of a model over time as it is trained on additional data. The mean accuracy score is expressed by the following equation:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{4.2}$$

4.3.2 F1-Measure

The F1-measure, also known as the F1-score, is a statistical metric that considers both precision and recall, and provides a balanced view of a model's performance. The formula for F1-measure is as follows:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4.3}$$

In this context, precision refers to the proportion of true positives among all positive predictions made by the model and recall refers to the proportion of true positives among all positive cases in the dataset [33]. By calculating the harmonic mean of precision and recall, the score provides a useful way to evaluate the result of a binary classification task and balanced assessment of our CodeBERTa model's performance.

4.3.3 Confusion Matrix

A confusion matrix is a table that summarizes the predictions made by the model during the testing phase and compares them with the true labels assigned to the dataset. It is organized into four quadrants: true positives (TP), false positives (FP), true negative (TN), and false negative (FN). In our study, a true positive occurs when the model identifies a vulnerability within a code segment and a false positive occurs when the model thinks a non-vulnerable piece of code has a bug. Alternatively, a true negative occurs when the model classifies non-vulnerable code as vulnerability-free and a false negative occurs when the model fails to identify a security vulnerability inside the example. This breakdown of classification rates provides insight into possible areas of improvement for the model. For example, if the model has a high false negative rate, it may mean that more data is needed to improve the model's ability to identify positive instances. The general format for a

2x2 confusion matrix is described below.

	Actual Positive	**Actual Negative**
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

While it would be ideal for our model to minimize both false positives and false negatives, prioritizing the minimization of false negatives aligns with our objective. False positives can be tolerated, as there are limited consequences of mistaking secure code as vulnerable. Conversely, classifying vulnerable code as secure can result in substantial financial and cybercrime-related damages. Therefore, a threshold of 0.5 was maintained in classifying code segments with the aim of reducing the likelihood of identifying code as secure when it is actually vulnerable.

4.3.4 ROC-AUC Curve

A Receiver Operating Characteristic (ROC-AUC) curve is a graph of a model's performance across a binary classification task. During a classification task, the model predicts whether each sample corresponds to one of two classes, generating a true positive rate (TPR) that measures the proportion of positives correctly identified by the model. Likewise, a false positive rate (FPR) is created, which measures the proportion of actual negatives the model incorrectly classified as positive. In an ROC-AUC curve, the TPR is plotted against the FPR at various classification thresholds. This creates a visual representation of the trade-off between TPR and FPR [33].

The most important element of an ROC-AUC curve in terms of accuracy is the area under the curve (AUC). The AUC indicates how capable the model is at distinguishing between classes. Given a binary classification task, a higher AUC is

associated with the model being better at predicting classes according to their true values – or, in the case of our study, at identifying the difference between code with a vulnerability and without a vulnerability.

4.3.5 PR-AUC Curve

The Precision-Recall (PR-AUC) curve essentially combines the methodology of an ROC curve with the metrics defined by the F1-measure. Rather than plotting the TPR and FPR, a PR-AUC plots the model's precision against its recall at different thresholds. The variance between precision and recall is important because the two usually share an inverse relationship, where developers are often forced to increase precision at the cost of decreasing recall and vice versa [33].

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.4}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.5}$$

A perfect classifier (one that always correctly discriminates between examples) will have a PR-AUC value of 1, whereas a model that cannot tell the difference between classes will return a value much closer to 0. Thus, as is the case with ROC-AUC, the closer the model's PR-AUC is to 1, the better the model is at the classification task it is being tested on.

Chapter 5

Results

5.1 Semantic Differences between Vulnerable and Non-Vulnerable Code

To address our first research question, we examine how our CodeBERTa model tokenized input strings from our database of code segments. Figure 5.1 displays an example of a tokenized input string. The tokenization process involves two major steps. First, the code is broken down into a sequence of smaller units that represent individual words, subwords, or characters. Second, a portion of these tokens are masked to create a partially masked input sequence that will be used by the model during training [14]. For CodeBERTa specifically, the subword tokenization is performed using Byte-Pair Encoding (BPE), which is a widely used algorithm for tokenizing text. BPE functions by iteratively merging pairs of the most frequently adjacent byte-pairs in the text until a predefined vocabulary size is reached. After tokenization, the tokens are encoded using a combination of token embeddings and position embeddings. These embeddings produce a final representation of the input sequence and indicate the position of each token in the input sequence [13].

```
input_tokens: ['<s>', 'static', 'void', 'v', '4', 'l', '2', '
    ', 'free', '', 'buffer', '(', 'void', '*', 'opaque', ',',
    'uint', '8', '', 't', '', 'un', 'used', '){',
'_V', '4', 'L', '2', 'Buffer', '', 'av', 'buf', '=', 'opaque'
    , ';',
'V', '4', 'L', '2', 'm', '2', 'm', 'Context', '*', 's', '=',
    'buf', '', 'to', '', 'm', '2', 'm', 'ctx', '(', 'av', 'buf
    ', ');',
'if', '(', 'atomic', '', 'fetch', '', 'sub', '(&', 'av', 'buf
    ', '->', 'context', '', 'ref', 'count', ',', '1', ')', '==
    ', '1', ')', '{',
'atomic', '', 'fetch', '', 'sub', '', 'explicit', '(&', 's',
    '->', 'ref', 'count', ',', '1', ',', 'memory', '', 'order'
    , '', 'acq', '', 'rel', ');',
'if', '(', 's', '->', 're', 'init', ')', '{',
'if', '(!', 'atomic', '', 'load', '(&', 's', '->', 'ref', '
    count', '))', 'sem', '', 'post', '(&', 's', '->', 'ref', '
    sync', ');',
'}', 'else', 'if', '(', 'av', 'buf', '->', 'context', '->', '
    stream', 'on', ')', 'ff', '', 'v', '4', 'l', '2', '', '
    buffer', '', 'enqueue', '(', 'av', 'buf', ');',
'av', '', 'buffer', '', 'unref', '(&', 'av', 'buf', '->', '
    context', '', 'ref', ');', '}}', '</s>']
```

Figure 5.1: Example of a Tokenized Input Sequence

The process of token extraction is an essential aspect of our CodeBERTa model because it allows the model to capture semantic relationships and contextual information present in the code and thereby improves the effectiveness of the model for various code-related tasks. However, it should be noted that CodeBERTa models have a maximum input sequence length of 512 tokens. Of the 27,420 input sequences in our database, 18,337 are longer than 512 tokens and 9,083 sequences are not. In instances where the input sequence exceeds this length, the model performs "top-n" truncation, where the most important subtokens are selected based on their assigned importance score, which was determined during pre-processing by constructing a subtoken vocabulary based on frequency in the training data [13] [14]. The selected subtokens

are included in the truncated sequence, while the rest are discarded. Although this procedure may result in the elimination of some information from the initial sequence, it allows the model to handle the input sequence and decreases the computational resources necessary for processing. Furthermore, while information loss may limit the model's ability to fully capture the relationship between tokens and vulnerabilities, it should be noted that the model can still view entire code segments and perform accurate binary classifications. For a more comprehensive explanation of the tokenization process and its limitations, refer to Section 6.3.

To examine the classes for inherent differences or token-clustering patterns, we applied the t-distributed stochastic neighbor embedding (t-SNE) algorithm to visualize the embeddings of 500 randomly selected input strings. Specifically, we used the PyTorch implementation of the t-SNE algorithm provided by the scikit-learn library. We also used the KMeans clustering algorithm from scikit-learn to group the input sequences into two clusters based on their embeddings and predicted cluster labels for each input sequence. Distance between points in a t-SNE plot reflects the similarity between the corresponding data points in the high-dimensional space. In less technical terms, points that are close together in the plot are considered similar to each other in the high-dimensional space and vice versa. The resulting t-SNE plot is shown in Figure 5.2.

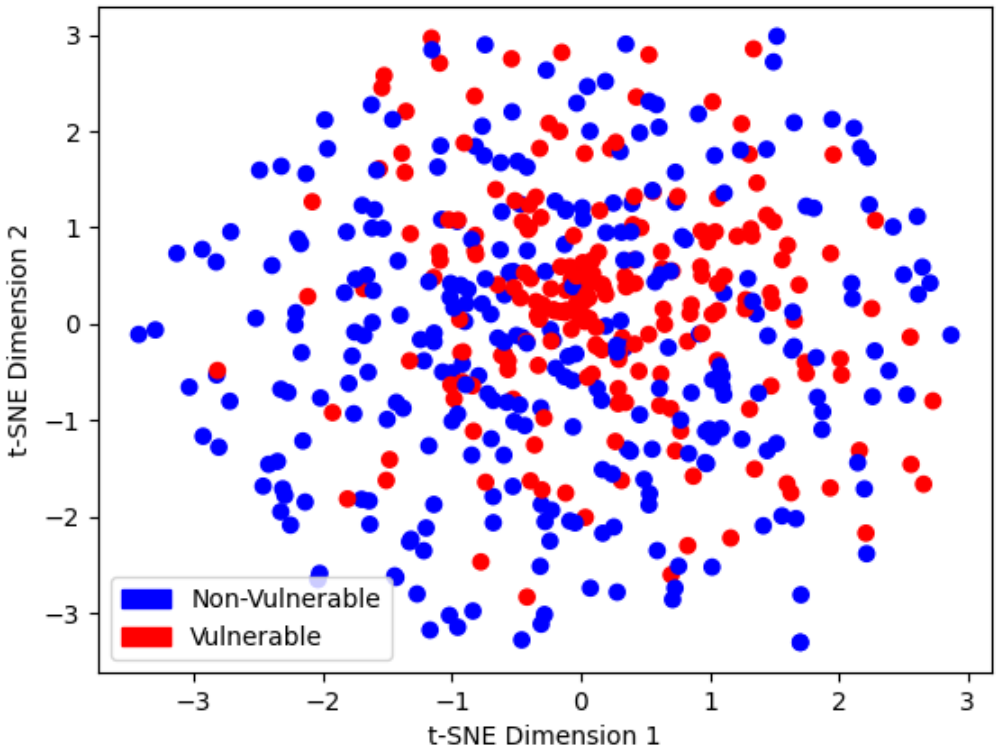

Figure 5.2: T-SNE Plot for CodeBERTa Encodings

5.2 Model Evaluation

Table 5.1 shows the evaluation of our model based the four metrics discussed above. Our CodeBERTa model achieves an accuracy score of 62%, meaning that it correctly predicts the class label of a code sequence for 62% of instances in the test set. Additionally, the model returns an F1-measure of 0.55. This number is greater than the F1-measure that is associated with random guessing (.50), implying that the model's performance on the dataset is moderate but has room for improvement.

Metric	Value
Accuracy	62.88%
F1-Measure	0.55
ROC-AUC	0.70
PR-AUC	0.66

Table 5.1: CodeBERTa Model Performance

The results of our CodeBERTa model's performance are exemplified in the ROC-AUC curve and the PR-AUC curve in Figure 5.3. The ROC-AUC score for this model was 0.70, illustrating that the model is reasonably capable of distinguishing between positive and negative instances. The model's PR-AUC curve demonstrated similar results. Our model achieved a PR-AUC score of 0.666 for the binary classification task. Given that a rating of 1.0 indicates perfect performance and a rating of .5 indicates random guessing, a PR-AUC score of .666 suggests the model is performing adequately in this task.

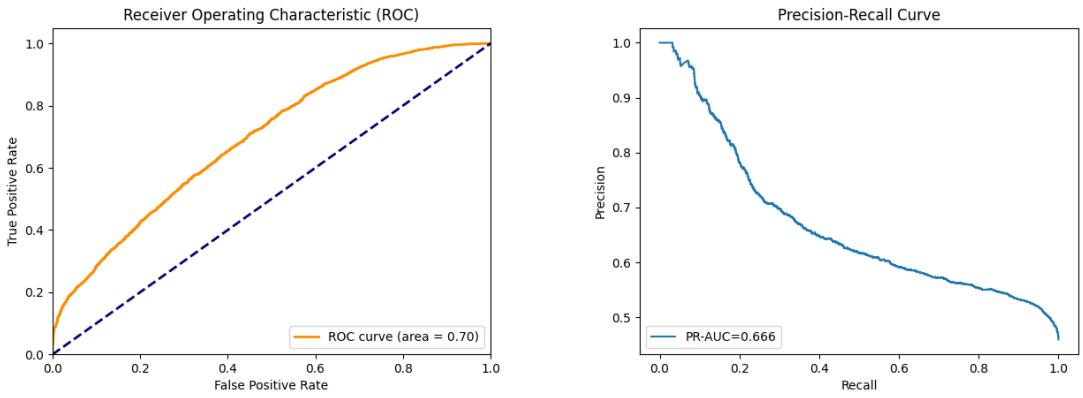

Figure 5.3: Model Performance on Binary Classification Task

The rate of false positives, true positives, false negatives, and true negatives are also displayed in a confusion matrix (Figure 5.4). Confusion matrices are oriented such that the percentage of true negatives is displayed in the upper left corner, false positives in the upper right, false negatives in the bottom left, and true positives in the bottom right. The breakdown of our model is as follows: true positive (49%), true negative (74%), false positive (26%), and false negative (51%). More concretely,

these percentages mean that the model correctly identified 49% of the positive instances in the data set, correctly identified 74% of the negative instances in the data set, incorrectly classified 26% of the negative instances as positive, and incorrectly classified 51% of the positive instances as negative.

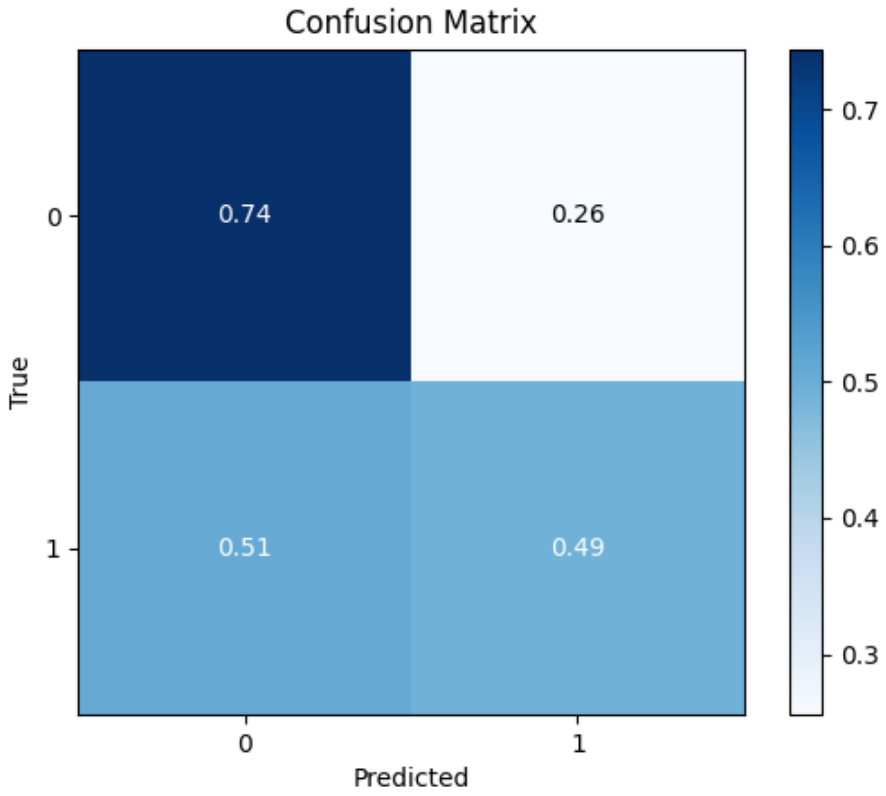

Figure 5.4: Confusion Matrix for CodeBERTa Model

Chapter 6

Discussion

6.1 Empirical Analysis

According to the evaluation statistics provided in Figures 5.1, 5.3, and 5.4, our CodeBERTa model has moderate performance on this binary classification task. Although many of the metrics overlap with one another, examining them holistically paints a clearer picture about the mode's overall effectiveness. While the F1-measure of .55 indicates only marginal improvement compared to random guessing, other metrics present a more positive view of the model's capabilities. For instance, the general accuracy score of 62.88%, the PR-AUC score of 0.666, and the ROC-AUC score of 0.70 all suggest that our model is performing notably better than mere chance.

Delving deeper into the PR-AUC score, ROC-AUC score, and confusion matrix rates is crucial to evaluating the true nature of the model's performance. A PR-AUC score of 0.666 implies that the CodeBERTa model correctly identified a significant proportion of positive instances in the data set, while minimizing the number of false positives. The confusion matrix further emphasizes this point, with a false positive rate of 26%. Furthermore, an ROC-AUC score of .70 indicates that the model can distinguish between positive and negative instances in the dataset, albeit only

moderately. Such results are once again echoed by the confusion matrix. A higher false negative rate (51%) than false positive rate (26%) suggests that the model is more likely to incorrectly classify a positive instance as negative than the alternative, which could be problematic for identifying vulnerabilities in code segments. Therefore, the combination of these metrics conveys that our CodeBERTa model adequately recognizes differences between positive and negative instances but has some weakness in correctly identifying positive instances. Putting this in terms of our defect detection task, the model learns the differences between vulnerable and non-vulnerable code segments but has trouble identifying vulnerable instances outright.

6.2 Comparative Performance

As explained in the data section, the dataset used in this project has not been previously utilized. As such, there is no straightforward way to compare our results with other baselines. To contextualize our results and highlight a typical range of performance for BERT-based models trained on code-specific tasks, Table 6.1 compares the performance metrics reported by similar works to ours. Mashhadi [24], Sequencer [9], and Tufano et al. [50], all trained BERT-based models with the goal of developing an automated program repair system that could generate patch files for code segments.

Table 6.1: Comparative Model Performance - Accuracy (%)

Model	**Accuracy Range**	**Dataset**
Masshadi [24]	19.65%-23.37%	Unique
	68.80%-72.00%	Duplicate
Sequencer [9]	18.00%	Defects4J
	20.00%	CodeRep4
Tufano et al [50]	13.12%-27.33%	CodeRep
	3.33%-10.27%	BFP
CodeBERTa	62.88%	Updated-CVE

Accuracy values for Mashhadi range from 19.65% to 23.27% for unique datasets

(datasets that contain distinct observations or examples that are not found in other dataset) and 68.8% to 72% on duplicate datasets (datasets that are copied with the same observations and fed into the model multiple times) [24]. Sequencer and Tufano et. al report similar accuracy values percentages across the unique datasets used between them [50]. For the Defectes4J and CodRep4 datasets, Sequencer claim an accuracy rate of 18% and 20% [9], and for the BFP and CodRep datasets, Tufano et. al show an accuracy range of 13.12%-27.33% and 3.33-10.27% [50].

Additionally, we assess the efficacy of our model by reference to the F1-score results reported by Pan et al [32]. Pan et. al (Section 3.1) evaluated the results of several BERT-based models, including CodeBERT-PS, CodeBERT-PK, CodeBERT-PT, and the RANDOM model, on two datasets: CVPSC and CPPSC. The average F1-scores reported by these models (listed in Section 3.3) are juxtaposed with ours in Table 6.2 and ranged from 0.397 to 0.631 [32].

Model	**F1-Measure**	**Dataset**
CodeBERT-PS [32]	0.616	CVPSC
CodeBERT-PS [32]	0.570	CCPSC
CodeBERT-PK [32]	0.631	CVPSC
CodeBERT-PK [32]	0.551	CCPSC
CodeBERT-PT [32]	0.565	CVPSC
CodeBERT-PT [32]	0.519	CCPSC
Random [32]	0.397	CVPSC
Random [32]	0.402	CCPSC
CodeBERTa	0.550	Updated-CVE

Table 6.2: F1-Score for Various Models on Different Datasets

Based on these comparisons, it is apparent that the performance of our CodeBERTa model is similar to that to that of other BERT-based models. Our CodeBERTa model achieved an F1-score of 0.55, which falls in the middle of the range of F1-scores for these BERT-based models [32]. While this F1-score does not indicate outstanding model performance, it is consistent with the standard range of performance for CodeBERT and its variations. In terms of accuracy, our value of 62.88%

falls only slightly below the range of Mashaddi's duplicate dataset [24] and otherwise exceeds the accuracy range for Sequencer [9], Tufano et. al [50], and Mashaddi's unique dataset [24]. Therefore, these findings suggest our CodeBERTa model's performance is comparable to and occasionally better than that of other BERT-based models, lending greater validity to our results and indicating that while our model may not achieve the desired level of accuracy, it aligns with the general performance range of NLP models.

6.3 Limitations and Confounding Variables

This study aims to investigate the feasibility of utilizing ML models to address the challenges posed by conventional fuzzing methods in detecting vulnerabilities. Despite its capabilities in identifying vulnerable-free code and predicting security defects with moderate accuracy, our CodeBERTa model is not without limitations. To start, the dataset employed in training and testing the CodeBERTa model is limited in scope as it only includes code segments written in one programming language. We chose to restrict examples to only one programming language to eliminate the impact of syntax incompatibility across programming languages, which could potentially create confusion in the binary classification and vulnerability detection processes. C programming language was selected as the single programming language because the original dataset was comprised exclusively of C programming examples.

As consequence of limiting the training data to a single programming language, the model may not be able to effectively detect defects in other programming languages or defects that do not commonly in C programming. Furthermore, the training data may not provide a representative sample of real-world software vulnerabilities. Many software systems are built using multiple programming languages, and defects can arise from interactions between parts of the system written that are written in

different languages. By limiting the training data to only one language, the model may not be able to capture the complexities of inter-language interactions, especially when those interactions cause defects. Therefore, it is important to consider the diversity of programming languages and the interplay between them when training a defect detection model. Including examples from multiple programming languages and ensuring that the training data is representative of real-world software development can help to create a more robust and effective model and is something future works ought to explore.

It is also possible that our dataset is comprised of vulnerabilities or patch files that are, for whatever reason, easier to identify. The data collection process was random but the complete dataset was not examined to ensure the examples covered a wide range of vulnerability types, had varying degrees of complexity, or otherwise had some level of diversity. Therefore, it is entirely possible that the examples incorporated in the dataset are simply easier to find and that has contributed to our model's positive evaluation metrics.

A second limitation of the CodeBERTa model concerns its tokenization process, as BERT-based models have a sequence length limit of 512 tokens that leads to the truncation of almost two-thirds of the code segments in the training dataset. This process restricts the model's ability to comprehend the complete context of a vulnerability, or lack thereof, and overcoming this limitation is challenging for various reasons. First, enabling the model architecture to handle sequences longer than 512 tokens usually necessitates training the model from scratch, which incurs considerable computational, time, and resource requirements that exceed those allocated for this project. Additionally, obtaining sufficient labeled data to train the model from scratch on sequences larger than 512 tokens is difficult. Second, modifying the architecture of a pre-trained model (against developer recommendations) introduces a range of formatting and dependency issues that propagate throughout the classes, functions,

and scripts used by the model. While it is possible to modify the pre-trained model's design to 'chunk' input sequences, add adaptive computation, and implement multi-stage modeling, each approach introduces additional complexity, potential for error, and data biases. For instance, chunking requires substantial memory requirements, which can be challenging to handle on regular graphics processing units (GPUs), making this tokenization modification prohibitively expensive for large models such as CodeBERT. Consequently, training large models from scratch to extend input sequence capacity and fine-tuning pre-trained models on long sequences remains an active area of research in natural language processing and other related fields.

Nevertheless, it is vital to note that despite this limitation, the CodeBERTa model still analyzes and examines the complete input sequence from the dataset, even if the sequence surpasses the 512 token limit. Truncation occurs only during the pre-processing phase, indicating that the model still views the code segments from the dataset in their entirety. Its performance on the binary classification task is thus negatively impacted only if the most crucial subtokens (see Section 5.1) are not appropriately chosen.

The t-SNE plot (Figure 5.2) may be impacted by the loss of information that occurs during tokenization, which could lead to a less accurate representation of the data. In particular, certain categories of tokens may be missing from the graph or misclassified, making it challenging to draw reliable conclusions about the relationships between input sequences. Although the t-SNE plot seems to show that vulnerable tokens are concentrated in the center, the absence of clear clusters may be caused by the truncation process rather than a flaw in the model's performance. Therefore, it is important to avoid using this as evidence both against the hypothesis that vulnerable code is distinguishable from non-vulnerable code and against the model's overall effectiveness.

Another possible limitation of our CodeBERTa model is our ML philosophy. Dur-

ing our training process, we adopted a direct feature identification approach – also known as feature learning – rather than extracting a relevant set of features that capture the underlying structure of the data and training the model specifically on these features. This approach uses HuggingFace transformers to identify vulnerabilities directly from the raw data without explicitly being told which features are important for the purposes of classification. Although choosing to employ feature learning through transformers is common throughout ML, it has the possibility to generate biases or confounding variables, particularly in the case of NLP and a training dataset in which the relevant features are extremely nuanced. Examples of possible confounding variables as a result of this approach include the following: data bias, overfitting, and limited control over features.

Data bias occurs when ML algorithms are skewed towards certain features that are overrepresented in the training dataset, leading to poor performance when applied to out-of-sample data and resulting in incorrect predictions. Given that our dataset is restricted to Linux-kernel programs written in C, there may be code features or functions that are more prevalent than others. This can, in turn, cause overfitting and reduce the model's effectiveness. Moreover, the subtle distinctions between vulnerable and non-vulnerable code may also have contributed to this issue (as discussed in Section 6.4), worsening the likelihood of data bias. To mitigate overfitting, we implemented early stopping and a learning rate scheduler; however, feature learning may still have caused overfitting and caused the model to terminate prematurely during training.

Feature learning may have also caused biases due to the algorithm being responsible for selecting the features it considers relevant, as opposed to being fed the relevant features directly. This aspect of feature learning gives developers limited control over the features the model uses and can become problematic when there are features that are important but are not being captured. Without the ability to see which features

are driving the model's predictions, it is challenging to identify whether this limited control affects the model's performance. Therefore, the best we can do is acknowledge the potential for such biases, be mindful of the limitations of this approach, and try to account for them when feasible.

6.4 Code Bugs versus Code Flaws

The ability to distinguish between vulnerable and non-vulnerable code is predicated on the existence of clear and discernible differences between the two. While it is widely accepted that certain kinds of code are more prone to bugs than others, as evidenced by extensive review of vulnerability databases, the results of our study suggest that the differences between vulnerable and non-vulnerable code may be more subtle and nuanced than previously thought. Our findings, therefore, highlight the need for a more in-depth discussion about the difference between code "bugs" and code "flaws" in the context of vulnerability detection.

Vulnerable Code

```
static int uvesafb_setcmap(struct fb_cmap *cmap, struct fb_info *info)
{
  struct uvesafb_pal_entry *entries;
  int shift = 16 - dac_width;
  int i, err = 0;
  if (info->var.bits_per_pixel == 8) {
    if (cmap->start + cmap->len > info->cmap.start +
        info->cmap.len || cmap->start < info->cmap.start)
      return -EINVAL;

    entries = kmalloc(sizeof(*entries) * cmap->len, GFP_KERNEL);
    if (!entries)
      return -ENOMEM;

    for (i = 0; i < cmap->len; i++) {
      entries[i].red   = cmap->red[i]   >> shift;
      entries[i].green = cmap->green[i] >> shift;
      entries[i].blue  = cmap->blue[i]  >> shift;
      entries[i].pad   = 0;
    }
    err = uvesafb_setpalette(entries, cmap->len, cmap->start, info);
    kfree(entries);
  } else {
    /*
     * For modes with bpp > 8, we only set the pseudo palette in
     * the fb_info struct. We rely on uvesafb_setcolreg to do all
     * sanity checking.
     */
    for (i = 0; i < cmap->len; i++) {
      err |= uvesafb_setcolreg(cmap->start + i, cmap->red[i],
             cmap->green[i], cmap->blue[i],
             0, info);
    }
  }
  return err;
}
```

Patch

```
static int uvesafb_setcmap(struct fb_cmap *cmap, struct fb_info *info)
{
  struct uvesafb_pal_entry *entries;
  int shift = 16 - dac_width;
  int i, err = 0;
  if (info->var.bits_per_pixel == 8) {
    if (cmap->start + cmap->len > info->cmap.start +
        info->cmap.len || cmap->start < info->cmap.start)
      return -EINVAL;

    entries = kmalloc_array(cmap->len, sizeof(*entries),
          GFP_KERNEL);
    if (!entries)
      return -ENOMEM;

    for (i = 0; i < cmap->len; i++) {
      entries[i].red   = cmap->red[i]   >> shift;
      entries[i].green = cmap->green[i] >> shift;
      entries[i].blue  = cmap->blue[i]  >> shift;
      entries[i].pad   = 0;
    }
    err = uvesafb_setpalette(entries, cmap->len, cmap->start, info);
    kfree(entries);
  } else {
    /*
     * For modes with bpp > 8, we only set the pseudo palette in
     * the fb_info struct. We rely on uvesafb_setcolreg to do all
     * sanity checking.
     */
    for (i = 0; i < cmap->len; i++) {
      err |= uvesafb_setcolreg(cmap->start + i, cmap->red[i],
             cmap->green[i], cmap->blue[i],
             0, info);
    }
  }
  return err;
}
```

Figure 6.1: Side-By-Side Vulnerability Comparison

Figure 6.1 illustrates a vulnerability and its corresponding patch from the training dataset. In this example, incorrect usage of the "kmalloc" allocation function allows '`cmap` → `len`' to exceed its maximum value, leading to an integer overflow. A comparison of these two files indicates that, apart from this small six-character change, they are identical among the more than 2,000 lines of code. A similar pattern is present among the 102 defect and patch file pairings added to the original dataset as part of this project, as shown in Figure 6.2. On average, an input code segment is 2,233 lines and the number of lines changed per vulnerability is 9.98. And, in many cases, line alterations consist of minor character additions or deletions similar to the one displayed in Figure 6.1. This considerable difference between the size of an input code segment and the size of a vulnerability patch implies that detecting a defect is often akin to finding a needle in a haystack. As such, it seems that a majority of vulnerability detection is focused more on identifying small bugs within a larger codebase, rather than identifying significant errors in terms of code architecture or style.

Commit ID	Lines Changed	Total Lines
CVE-2002-2443	4	486
CVE-2006-5331	2	1109
CVE-2007-6761	3	370
CVE-2008-7316	3	2582
CVE-2009-1194	15	686
CVE-2009-3111	2	2507
CVE-2009-3627	6	200
CVE-2010-0011	7	530
CVE-2010-1152	154	4661
CVE-2010-1155	1	549
CVE-2010-3696	19	1460
CVE-2010-3697	10	3748
CVE-2010-4159	1	2443
CVE-2010-4250	1	880
CVE-2011-0006	2	491
CVE-2011-0716	2	1846
CVE-2011-0989	5	8035
CVE-2011-0990	4	8044
CVE-2011-1019	7	6334
CVE-2011-1078	1	1088
CVE-2011-1079	1	259
CVE-2011-1160	1	1271
CVE-2011-1759	1	453
CVE-2011-1767	10	1711
CVE-2011-2182	4	1568
CVE-2011-2183	2	2029
CVE-2011-2496	7	528
CVE-2011-2517	2	6901
CVE-2011-2518	1	285
CVE-2011-2521	1	1900
CVE-2011-2707	2	348
CVE-2011-3191	2	6090
CVE-2011-3353	3	2040
CVE-2011-3637	2	879
CVE-2011-4081	4	176
CVE-2011-4097	1	778
CVE-2011-4326	1	1512
CVE-2011-4594	6	3384
CVE-2012-1583	1	381
CVE-2012-2375	1	6527
CVE-2012-2383	7	1451
CVE-2014-1912	2	2877
CVE-2017-7374	8	483
CVE-2017-7487	3	2083
CVE-2017-7541	5	7161
CVE-2017-8062	145	2403
CVE-2017-8063	8	2198
CVE-2017-8064	5	1126
CVE-2018-1000156	4	2574
AVERAGE	**9.98**	**2233.16**

Figure 6.2: CVE Vulnerability and Patch Statistics

Because vulnerability detection tends to involve finding security bugs as opposed to larger code architecture flaws, it is not surprising that the CodeBERTa model struggled to identify positive instances of vulnerabilities. Detecting security vulnerabilities often requires parsing through large amounts of code, understanding the complex interactions between different parts of the code, and making nuanced judgments based on subtle differences between code segments. In contrast, identifying larger errors in code architecture is generally more straightforward and can be done without a deep understanding of the codebase's inner workings. Therefore, one of the challenges with using a model like CodeBERTa for vulnerability detection is that it may struggle to identify subtle vulnerabilities that require a more in-depth analysis of the code. The model's architecture, which has a 512 token limit, may not capture all of the crucial syntactic and semantic features of the input sequence and its surrounding context. As a result, vulnerabilities that require a deeper understanding of the relationships within a code segment may be missed.

One of this study's contributions to the field of vulnerability detection lies in its distinction between code bugs and code flaws. Different approaches are necessary depending on the type of programming error a developer, fuzzer, or ML technique is attempting to detect. While larger errors related to code architecture or style can be more easily resolved through fuzzing or better development practices, smaller programming errors that have the potential to significantly impact the security of a system require a greater level of focus and attention. These errors can be difficult and time-consuming to detect, particularly when they are deeply embedded within the source code.

Despite requiring improvements to identify code bugs more accurately, our CodeBERTa model is at the forefront of the discovery process for these types of vulnerabilities. Its findings suggest that the identification of vulnerabilities in software code may require a more complex and nuanced approach than previously assumed. Still,

ML is an efficient approach for uncovering small programming errors amidst large codebases because it does not require extensive time, resources, or user involvement to do so. Therefore, future vulnerability detection efforts ought to prioritize the development of more advanced and sophisticated ML tools and methodologies to identify these smaller code bugs that might otherwise be missed through manual code reviews, traditional fuzzing methods, or other non-detail-oriented tactics.

Chapter 7

Conclusion

We propose a novel approach to vulnerability discovery that uses ML, specifically Microsoft's bimodal pre-trained model for NLP CodeBERTa, to ameliorate the shortcomings of modern fuzzing techniques. Our model is fine-tuned on an expanded version of the vulnerability dataset curated by Zhou et. al [54] to determine its ability to identify security errors within code segments. Our implementation of CodeBERTa correctly identifies negative instances of vulnerabilities in 79% of the cases and positive instances of vulnerabilities in 49% of the cases, resulting in an overall accuracy score of 62.88% and an F1-score of .55. Comparative analysis demonstrates that these results are on-par with those generated by BERT-based models used in other experiments and, in some cases, considerably better.

Placing these results in context also includes discussing the limitations of our approach, which include token length restrictions and the downsides of our ML philosophy. The short token lengths permitted by BERT-based models prevents the entire context of a code segment from being tokenized , making it more difficult for our model to process the full context of a vulnerability. Consequently, the token limit could negatively impact our model's ability to find security errors. A second element that could reduce the model's overall performance is our use of feature learning. In

feature learning, the model itself learns to extract the most relevant features from the data. In many cases this can lead to better performance – especially in domains where relationships between the features and the target variable are not well understood – but it is also susceptible to biases that are difficult to rectify.

In light of the aforementioned results and limitations, it is important to reiterate that primary goal of this project is not to solve the issue of automated vulnerability detection but instead to improve upon the current state-of-the-art. Vulnerability detection is a massive challenge that has stumped many experts, both within and outside the field of ML. For instance, the latest iteration of ChatGPT, ChatGPT-4, has attempted to tackle cybersecurity subtasks but has faced immense difficulty in identifying vulnerable code due to its limited context window and tendency to "hallucinate." Despite receiving extensive input from cybersecurity experts, ChatGPT-4 still falls short of the benchmark set by existing vulnerability identification tools [29]. This failure indicates the enormity of the issue at hand. Therefore, our project does not seek to solve the problem completely. Rather, it aims to contribute to a potential solution by enhancing current techniques, even if only by a slight margin.

With this goal in mind, the results of our investigation indicate that the employment of ML techniques can enhance the efficacy of vulnerability detection, even though our hypotheses were not entirely validated. The model's ability to identify vulnerable vs. non-vulnerable code with relative accuracy suggest that there are distinct differences between the two, but a more robust version of this model is needed to definitively prove our hypotheses. Future research can explore ways to optimize the model's performance by expanding the dataset and adjusting the model's max token length, and build upon our initial classifications to create a highly accurate vulnerability gradient. Additionally, researchers can investigate how to utilize these findings in real-world software development environments and determine how to integrate these methods into existing software development pipelines.

Overall, our CodeBERTa model has the potential to significantly improve the efficiency of vulnerability detection and reduce the time and resources required for software engineers to search for erroneous code segments, scour stack traces for crashes, or run an entire library through a fuzzer. By leveraging the power of deep learning in this study, we hope to provide insight into the potential benefits of using these technologies for improving software security and demonstrate that machine learning offers a promising avenue for enhancing the efficiency of vulnerability detection techniques. In an era of extreme cybercrime, such efficiency gains are invaluable, and the development of our CodeBERTa methodology may serve as a crucial step in the ongoing fight against cybercriminals.

Bibliography

[1] National vulnerability database. `https://nvd.nist.gov/vuln`. Accessed on March 24, 2023.

[2] Vaibhav Anu, Kazi Zakia Sultana, and Bharath Samanthula. A human error based approach to understanding programmer-induced software vulnerabilities. pages 49–54, 10 2020. doi: 10.1109/ISSREW51248.2020.00036.

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1965–1982, 2019.

[4] William Blum, Mohit Rajpal, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. November 2017. URL `https://www.microsoft.com/en-us/research/publication/not-all-bytes-are-equal-neural-byte-sieve-for-fuzzing/`.

[5] Marcel Bohme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 34(4):96–100, 2017.

[6] Brown, Sara. Machine learning, explained. `https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained`, 2021. Accessed: March 23, 2023.

[7] Shubham Chaturvedi. Masked language modelling with bert. *To-*

wards Data Science, 2019. URL https://towardsdatascience.com/masked-language-modelling-with-bert-7d49793e5d2c.

[8] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018. doi: 10.1109/SP.2018.00046.

[9] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47 (9):1943–1959, 2021. doi: 10.1109/TSE.2019.2940179.

[10] Cloudflare. Cross-site scripting (xss), n.d. URL https://www.cloudflare.com/learning/security/threats/cross-site-scripting/. [Online; accessed 27-March-2023].

[11] Patryk Dabrowski, Robert Gawlik, Przemyslaw Mazur, and Aleksander Twardowski. Better pay attention whilst fuzzing. *International Journal of Information Security*, 20(2):217–231, 2021.

[12] Artem Dinaburg. Fuzzing like it's 1989. https://blog.trailofbits.com/2018/12/31/fuzzing-like-its-1989/, December 31 2018.

[13] Hugging Face. Preprocessing data with transformers. https://huggingface.co/docs/transformers/preprocessing, 2021. Accessed on March 24, 2023.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[15] Hossein Homaei and Hamid Reza Shahriari. Seven years of software vulnera-

bilities: The ebb and flow. *IEEE Security Privacy*, 15(1):58–65, 2017. doi: 10.1109/MSP.2017.15.

[16] IBM. Cost of a data breach 2022: A million-dollar race to detect and respond. https://www.ibm.com/reports/data-breach, 2022. Accessed: March 23, 2023.

[17] Coder's Kitchen. Fuzzing techniques: The ultimate guide. https://www.coderskitchen.com/fuzzing-techniques/, 2023. [Accessed: March 28, 2023].

[18] Kumud Gautami. Understanding CodeBERT. https://indiaai.gov.in/article/understanding-codebert, Aug. 2022. Accessed: March 23, 2023.

[19] Jinyuan Li, Bo Zhao, and Chunfu Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1): 6, 2018. doi: 10.1186/s42400-018-0002-y. URL https://link.springer.com/article/10.1186/s42400-018-0002-y.

[20] Junhao Lin and Lu Lu. Semantic feature learning via dual sequences for defect prediction. *IEEE Access*, 9:13112–13124, 2021. doi: 10.1109/ACCESS.2021.3051957.

[21] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[22] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, 188:111283, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111283. URL https://www.sciencedirect.com/science/article/pii/S0164121222000437.

[23] Serafeim Loukas. What is machine learning: Supervised, unsupervised, semi-supervised and reinforcement learning methods. https://towardsdatascience.com/what-is-machine-learning-a-short-note-on-supervised-unsupervised-semi-supervis June 2020.

[24] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs, 2021.

[25] Microsoft. Microsoft/CodeBERT. https://github.com/microsoft/CodeBERT, 2023. Accessed: March 23, 2023.

[26] Charlie Miller and Zachary N. J. Peterson. Analysis of mutation and generation-based fuzzing. 2007.

[27] MITRE Corporation. Cve - common vulnerabilities and exposures. https://cve.mitre.org/, Accessed: 2023.

[28] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for Use-After-Free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, San Sebastian, October 2020. USENIX Association. ISBN 978-1-939133-18-2. URL https://www.usenix.org/conference/raid2020/presentation/nguyen.

[29] OpenAI. Gpt-4 technical report, 2023.

[30] OWASP. Fuzzing. https://owasp.org/www-community/Fuzzing, accessed 2023. Accessed: March 23, 2023.

[31] Packetlabs. Broken access control, n.d. URL https://www.packetlabs.net/posts/broken-access-control/. [Online; accessed 27-March-2023].

[32] Cong Pan, Minyan Lu, and Biao Xu. An empirical study on software defect prediction using codebert model. *Applied Sciences*, 11(11), 2021. ISSN 2076-3417. doi: 10.3390/app11114793. URL https://www.mdpi.com/2076-3417/11/11/4793.

[33] Kate Pearce, Tiffany Zhan, Aneesh Komanduri, and Justin Zhan. A comparative study of transformer-based language models on extractive question answering, 2021.

[34] Zhongxuan Peng, Yingfei Shi, Jingxuan Wang, Yan Liu, Kai Ma, and Qingkai Zeng. Guided fuzzing for network protocols. *Scientific Reports*, 12(1):5731, 2022.

[35] Richard D. Pethia. Information technology—essential but vulnerable: how prepared are we for attacks? http://www.cert.org/congressional_testimony/Pethia_testimony_Sep26.html, 2001. [Online; accessed 27-March-2023].

[36] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing, 2019.

[37] Arash Sabbaghi and Mohammad Reza Keyvanpour. A systematic review of search strategies in dynamic symbolic execution. *Computer Standards Interfaces*, 72:103444, 2020. ISSN 0920-5489. doi: https://doi.org/10.1016/j.csi.2020.103444. URL https://www.sciencedirect.com/science/article/pii/S0920548919300066.

[38] Tanujay Saha, Tamjid Al Rahat, Najwa Aaraj, Yuan Tian, and Niraj K. Jha. Ml-feed: Machine learning framework for efficient exploit detection. In *2022 IEEE 4th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, pages 140–149, 2022. doi: 10.1109/TPS-ISA56441.2022.00027.

[39] Iqbal H. Sarker. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *Sn Computer Science*, 2, 2021.

[40] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a sat solver from single-bit supervision, 2019.

[41] Ahmad Shafique, Muhammad Usman Ashraf, Muhammad Awais, Adnan Sana, and Muhammad Ali Raza. A comprehensive survey of fuzzing techniques in software testing. *IEEE Access*, 7:123788–123820, 2019.

[42] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing, 2019.

[43] Shiqi Shen, Soundarya Ramesh, Shweta Shinde, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: The feasibility of an inductive approach to symbolic execution, 2018.

[44] Yuxin Song, Xiaohong Zhang, Zhi Wang, Chao Yang, Hongyu Yu, Yuan Cheng, Chen Qian, and Yang Liu. Cmfuzz: Context-aware adaptive mutation for fuzzers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 2023–2037, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370865. doi: 10.1145/3427228.3427243. URL `https://doi.org/10.1145/3427228.3427243`.

[45] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486, 2007. doi: 10.1109/ACSAC.2007.27.

[46] Steve Morgan. Cybercrime To Cost The World $10.5 Trillion Annually By 2025. `https://cybersecurityventures.com/`

hackerpocalypse-cybercrime-report-2016/, Nov. 2020. Accessed: March 23, 2023.

[47] Mario Calín Sánchez, Juan Manuel Carrillo de Gea, José Luis Fernández-Alemán, Jesús Garceran, and Ambrosio Toval. Software vulnerabilities overview: A descriptive study. *Tsinghua Science and Technology*, 25(2):270–280, 2020. doi: 10.26599/TST.2019.9010003.

[48] The MITRE Corporation. Cwe top 25 most dangerous software weaknesses - 2022. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html#cwe_top_25_with_scoring_metrics, 2022. [Online; accessed 27-March-2023].

[49] TheSecMaster. What is a buffer overflow attack and how to prevent it?, 2019. URL https://thesecmaster.com/what-is-a-buffer-overflow-attack-and-how-to-prevent-it/. [Online; accessed 27-March-2023].

[50] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 832–837, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3240732. URL https://doi.org/10.1145/3238147.3240732.

[51] Yuhui Wang, Peng Jia, Li Liu, Chuang Huang, and Zhiqiang Liu. A systematic review of fuzzing based on machine learning techniques. *PLoS ONE*, 15(8): e0237749, 2020. doi: 10.1371/journal.pone.0237749. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0237749. Received: May 31, 2020; Accepted: August 1, 2020; Published: August 18, 2020.

[52] WhiteSource. The State of Open Source Security Vulnerabilities: WhiteSource Annual Report 2020. https://www.mend.io/wp-content/media/2020/03/Annual_Report_2020_12.03.20.pdf, 2020. Accessed: March 23, 2023.

[53] L. Zhang, L. Tong, X. Hou, and X. Wang. Deep learning: A review of deep learning models for unsupervised and semi-supervised learning. *IEEE Access*, 7: 128833–128850, 2019. doi: 10.1109/ACCESS.2019.2933452.

[54] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019.