NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

A BENCHMARK FRAMEWORK AND SUPPORT FOR AT-SCALE BINARY VULNERABILITY ANALYSIS

by

Kayla N. Afanador

September 2021

Dissertation Supervisor: Cynthia E. Irvine

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2021	**3. REPORT TYPE AND DATES COVERED** Dissertation

4. TITLE AND SUBTITLE A BENCHMARK FRAMEWORK AND SUPPORT FOR AT-SCALE BINARY VULNERABILITY ANALYSIS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Kayla N. Afanador	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Today, software is integrated into nearly every aspect of our lives and so are its vulnerabilities. Exploited software vulnerabilities can have detrimental financial, social, and economic effects. Researchers rely on Vulnerability Analysis Tools and Techniques (VATT) to amplify the vulnerability analysis process. There are hundreds of VATTs on the market, but there is no way to compare their relative efficacy. We developed a framework for the Benchmark for Vulnerability Analysis Tools and Techniques (BVATT). In addition to providing key metrics for quantifying the performance of a particular VATT, the proposed framework ensures that BVATT will facilitate the comparison of different VATTs in a manner that is repeatable, reproducible, fair, verifiable, and relevant.

Additionally, in the past decade, there has been a noteworthy increase of VATTs that leverage machine-learning and data-mining techniques to identify vulnerabilities. Yet, there is no open-source tool to synthesize the extraction, cleaning, and transformation of common features from binary files to be compatible with these techniques. We develop such a tool, and call it BiSECT (Binary Synthesized Extraction, Cleaning, and Transformation). BiSECT reduces the barrier to entry and makes binary vulnerability analysis using data mining and machine learning more accessible to researchers.

14. SUBJECT TERMS software, bugs, vulnerabilities, automated vulnerability analysis, machine learning vulnerability analysis	**15. NUMBER OF PAGES** 211
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A BENCHMARK FRAMEWORK AND SUPPORT FOR AT-SCALE BINARY VULNERABILITY ANALYSIS

Kayla N. Afanador
Civilian, Department of the Navy
BBA, University of Maryland University College, 2012
MS, University of Maryland University College, 2015

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2021

Approved by:

Cynthia E. Irvine
Department of
Computer Science
Dissertation Supervisor and Chair

David L. Alderson
Department of
Operations Research

Robert Beverly
Department of
Computer Science

Peter J. Denning
Department of
Computer Science

Christopher S. Eagle
Department of
Computer Science

Alan B. Shaffer
Department of
Information Sciences

Approved by:

Gurminder Singh
Chair, Department of Computer Science

Michael E. Freeman
Vice Provost of Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Today, software is integrated into nearly every aspect of our lives and so are its vulnerabilities. Exploited software vulnerabilities can have detrimental financial, social, and economic effects. Researchers rely on Vulnerability Analysis Tools and Techniques (VATT) to amplify the vulnerability analysis process. There are hundreds of VATTs on the market, but there is no way to compare their relative efficacy. We developed a framework for the Benchmark for Vulnerability Analysis Tools and Techniques (BVATT). In addition to providing key metrics for quantifying the performance of a particular VATT, the proposed framework ensures that BVATT will facilitate the comparison of different VATTs in a manner that is repeatable, reproducible, fair, verifiable, and relevant.

Additionally, in the past decade, there has been a noteworthy increase of VATTs that leverage machine-learning and data-mining techniques to identify vulnerabilities. Yet, there is no open-source tool to synthesize the extraction, cleaning, and transformation of common features from binary files to be compatible with these techniques. We develop such a tool, and call it BiSECT (Binary Synthesized Extraction, Cleaning, and Transformation). BiSECT reduces the barrier to entry and makes binary vulnerability analysis using data mining and machine learning more accessible to researchers.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AFL	American Fuzzy Lop
AI	Artificial Intelligence
ANOVA	Analysis of Variance
BLSTM	Bidirectional Long Short-Term Memory
BVATT	Benchmark for Vulnerability Analysis Tools and Techniques
CAS	Center for Assured Software
CB	Challenge Binary
CFE	CGC Final Event
CFG	Control Flow Graph
CGC	Cyber Grand Challenge
CRS	Cyber Reasoning System
CVE	Common Vulnerability and Exposures
CWE	Common Weakness Enumeration
DARPA	Defense Advanced Research Projects Agency
DECREE	DARPA Experimental Cyber Research Evaluation Environment
DHS	Department of Homeland Security
DOD	Department of Defense
FECT	Feature Extraction, Cleaning, and Transformation
FN	False Negatives
FNR	False Negative Rate
FP	False Positives
FPR	False Positive Rate
FSM	Finite State Machines

IARPA	Intelligence Advanced Research Projects Activity
ICC	Intraclass Correlation Coefficient
LSTM	Long Short-Term Memory
NSA	National Security Agency
OS	Operating System
OWASP	Open Web Application Security Project
PE	Portable Executable
SARD	Software Assurance Reference Dataset
STONESOUP	Securely Taking On New Executable Software of Uncertain Provenance
TN	True Negative
TP	True Positive
VATT	Vulnerability Analysis Tool or Technique
VPM	Vulnerability Prediction Model

Acknowledgments

To my committee members, your collective knowledge of the history of computing continues to astound me. Your feedback and thorough reviews have enabled this work to meet its potential. I am honored to have been guided by each of you.

To my parents (Matt and Lisa), who never once doubted my abilities. You continually demonstrate faith, love, work ethic, and tenacity—these things have become pillars in my own life because of you. You have provided the foundation upon which I make every accomplishment in my life.

To Otto, who supported me throughout this endeavor even when it took me away from you (and the toddlers). You empowered me to rise to this challenge, celebrated every victory, and stood beside me during every defeat along the way. I am forever grateful. ATD. ATM.

To Leonora and Ottico, when I considered quitting (and I did), it was the thought of you that convinced me to persist. I hope that someday my example helps you to strive for excellence in spite of any adversity that you may face.

To the numerous others who supported me throughout this journey (VADM Jan Tighe, Jeremy, my siblings [Mariah, Scott, and Seana], my Grandfather, the Rat Pack, Dr. Lyn Whitaker). You believed in me, offered constant and candid reviews of my work, cared for my children during the final hour, read everything that I wrote, and encouraged me. This work was made possible because of you. Thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

In God we trust, all others bring data.

William E. Deming

1.1 Context

Software is pervasive. Nearly every electronic device runs embedded software. Our cars are computer networks on wheels run by software. Our ubiquitous smartphones are powered by operating systems that specialize in downloading apps to do common tasks. Apple currently offers over 2M apps in its store [1] and Google over 3M [2]. All our enterprises and all their supply chains are run by software. Government agencies store immense amounts of sensitive information on databases accessed through software. Utility companies manage critical infrastructure with software. Militaries use software for planning, logistics, weapons control, secret communications, and coordination of operations. Without software, we cannot function.

All this software is permeated with bugs. Software is too complex to guarantee that it is bug-free. Criminals pore through software looking for bugs they can use to make the software attack its host and destroy, steal, or poison data. We call bugs that can be exploited in this way "vulnerabilities." Many vulnerabilities are the result of design or implementation mistakes and are unknown to their developers. Attacks on such vulnerabilities are called "zero day" because they can be launched as surprises to the victims, who have "zero days" to develop defenses.

In recent years attacks on computer systems and their databases have exploded exponentially. Nearly everyone is concerned with identity theft. Ransomware has become a billion dollar enterprise [3]. We can no longer escape the nagging anxiety that our data will be lost or stolen in attacks, that our critical infrastructures will be brought down, or that our militaries will be unable to defend us [4], [5].

The pervasiveness of and the need to address software vulnerabilities are the context of this dissertation.

The downside consequences of unmitigated vulnerabilities are as immense as vulnerabilities are pervasive. The magnitude of analysis required to detect these bugs is so immense that even legions of experts cannot keep up with demand. Ultimately, vulnerability analysis must be automated.

Vulnerability Analysis Tool or Techniques (VATTs) are used to augment the vulnerability analysis process. Generally speaking, VATTs include any method (i.e., tool or technique) to identify, analyze, or prioritize vulnerabilities in software. Some VATTs leverage static or dynamic analysis, while others use a combination of the two to achieve their objectives.

The many existing VATTs are isolated efforts, and there is no standardized way to compare their performances. We set out to develop a framework for a dependable benchmark for comparing VATTs, i.e., Benchmark for Vulnerability Analysis Tools and Techniques (BVATT), and this led us to create another new tool, `BiSECT`. Following, we provide the chronicle of our work.

In attempting to identify a possible suite of test cases for BVATT we identified hundreds of thousands test cases in numerous publicly available datasets. We soon discovered that none of these datasets is able to represent the full range and frequency of vulnerabilities seen in the real-world. However, by selecting segments from each existing dataset, we could construct a new suite of problems (i.e., test cases) for BVATT that **is** representative of reality. To this end, we sought an automated method to extract features from each of the compiled test cases (software binaries) and compare those features to evaluate similarities and differences among the test cases. Furthermore, we demonstrated how clustering can be used to group the vulnerabilities exhibiting similar feature sets. By using clustering, we could maximize the diversity and comprehensiveness of BVATT.

As we ventured to cluster our compiled test cases, we were led to the second major challenge addressed by this work— there is no tool that synthesizes the at scale extraction, cleaning, and transformation of features commonly used in data mining and machine learning based vulnerability analysis. We developed such a tool, and called it `BiSECT`. `BiSECT` is available as open source. We designed `BiSECT` to accomplish our original task of providing the

benchmark problems for BVATT. We also found that BiSECT can be used to directly support at scale data mining and machine learning-based vulnerability analysis. BiSECT reduces the barrier to entry and makes binary vulnerability analysis using data mining and machine learning more accessible to security researchers.

The BiSECT system shows that almost anyone can use machine learning and data mining to search for vulnerabilities in their software. To help people learn how to use BiSECT, we prepared several Jupyter notebooks in the BiSECT repository with example problems, walkthroughs, and additional documentation related to BiSECT.

1.2 Motivation

The motivation for the use of tools and techniques to compliment the vulnerability analysis process is simple: the pervasiveness of and the need to address software vulnerabilities. Between 1999 to early 2020 over 150,000 software vulnerabilities were reported by the Common Vulnerabilities and Enumerations (CVE) database [6]. When vulnerabilities are exploited it can have detrimental financial, social, and economic effects. In 2020, the Ponemon Institute reviewed over 500 data breaches (often resulting from exploited vulnerabilities [7]), and found that the cost of a single breach averaged $3.8 million [8]. The Stuxnet and Duqu worms demonstrated how vulnerability exploitation can go beyond financial damage, and disrupt critical operations [9], [10]. In 2018, the Department of Homeland Security (DHS) identified five fundamental pillars upon which the security and resilience of cyberspace in the United States would stand— Pillar II specified the reduction of vulnerabilities to protect critical infrastructure and information systems [11]. Again in 2018, the Department of the Navy established a policy called "CYBERSAFE," that requires nearly all programs of record to identify and reduce vulnerabilities in each system prior to deployment [12]. Despite substantial investments in system security, vulnerabilities persist. In 2020, the Sunburst malware was used to compromise thousands of government systems and an estimated 18,000 firms [13]. Recent data exfiltration [14] and ransomware [15] attacks further illustrate the cybersecurity problem. To reiterate the importance of system security, the Department of Defense (DOD) allocated $9.85 billion to the cybersecurity domain in 2021, with $841 million specifically allocated to cybersecurity-related artificial intelligence activities [16]. Still, the question remains: *how can we identify and reduce software vulnerabilities?* One reduction approach is to begin with the assumption that all software contains vulnerabilities,

and then test each program to find as many vulnerabilities as possible [17].

Software testing techniques can be broadly classified as either *static* or *dynamic*. Simply stated, during static testing or analysis the program under test is not executed, whereas in dynamic analysis, it is. Control and data flow analysis, static symbolic execution, human code reviews and walkthroughs, and basic lexical analysis are examples of static analysis [18]. Debugging, taint analysis, fuzzing, and dynamic symbolic execution are forms of dynamic analysis [19]. Most dynamic analysis techniques require test cases to be either manually or automatically generated. Manually analyzing software and developing test cases requires domain expertise, is expensive, and time consuming [20]. Further, the limited number of specialists able to perform manual vulnerability analysis does not scale to the amount, complexity, and variety of software to be secured [20]–[22]. Researchers and consumers therefore rely on vulnerability analysis tools and techniques to augment the vulnerability analysis process. The importance of VATTs continues to be emphasized.

1.2.1 Motivating a Framework for BVATT

In May 2021, a ransomware attack was successfully launched against the Colonial Pipeline which provides nearly 45% of the fuel in the Eastern United States. Shortly after the attack, the US government issued an order requiring the development of guidance related to the employment of, "automated tools, or comparable processes, that check for known and potential vulnerabilities" [23].

Significant progress has been made to support the vulnerability analysis process using an array of tools and techniques, and while there are a few useful resources [24]–[26], a standard benchmark for VATTs has yet to be adopted by the community.

Researchers have highlighted the impact of the absence of such a benchmark [25], [27], [28]. For example, there is no standard method to assess the relative efficacy of VATTs. We have no comparative metrics of the types of vulnerabilities each tool and technique can or cannot find, how quickly they can do so, or the number of false positives they report. When a consumer wants to employ a VATT they have no standard method of comparing the alternatives to determine which is the optimal choice for their use case. When a developer wants to modify an existing VATT, or enter the market with a new one, no benchmark is available to compare their VATT against existing ones. Consumers and developers therefore

compare VATTs in a disjointed fashion, which can result in misinformation, subjectivity, and inconsistency.

A common comparison approach is to assess the relative performance of different tools and techniques using a collection of test cases containing known vulnerabilities. These test cases are often organized into public or proprietary datasets. Hundreds of thousands of publicly available test cases containing known software flaws and vulnerabilities have been aggregated into datasets, each with its own structure, supported languages, and reporting method. Databases such as the Software Assurance Reference Dataset (SARD) attempt to inject order by providing a consolidated repository of vulnerability datasets and test cases [24]. Unfortunately, even the SARD, which contains 40 datasets and over 170K test cases, is not exhaustive— it excludes datasets such as the Cyber Grand Challenge (CGC) Corpus [26], LAVA-M [25], and Open Web Application Security Project (OWASP) Benchmark Project [29]. Additionally, many of these datasets contain an unrealistic representation of weakness types, i.e., Common Weakness Enumeration (CWE) entries, when compared to known vulnerability instances in the wild, i.e., accepted Common Vulnerability and Exposures (CVE) entries [22]. Consequently, even if a VATT was assessed using *all* 170K of the SARD test cases, the results would still not reflect reality. Furthermore, while the vulnerabilities in some datasets are labeled at a function level, most are labeled at a file level. Labeling vulnerabilities with a finer granularity allows for a more insightful and quantitative assessment of the performance of different VATTs [27], [28], [30], [31]. Finally, while some of the datasets include real-world code (e.g., LAVA-M [25], and STONESOUP [32]) most are comprised of entirely synthetic code. Pure synthetic vulnerabilities are sometimes criticized for being overly simplistic and not representative of vulnerabilities in the real-world [27], [28].

Based on the various limitations of existing datasets, and the absence of a benchmark we surmised that the community is in need of an overarching framework to guide the implementation of BVATT.

1.2.2 Motivating an At-Scale Binary Code Feature Extraction Tool

In addition to conventional static and dynamic techniques, in the past decade researchers have begun leveraging machine learning and data mining techniques to identify, predict, and

prioritize vulnerabilities [27], [28], [33]–[36]. At the time of this writing, two major surveys of these techniques have been published. In 2017, Ghaffarian and Shahriari provided a survey of work that uses conventional machine learning or data mining techniques to identify and analyze software vulnerabilities [27]. Then, in 2020, Lin, et al. surveyed vulnerability detection methods that leverage deep learning and neural networks [28]. From these surveys we make two key observations:

1. The first four steps of the data mining process include data cleaning, integration, selection, and transformation [37]. While machine learning typically begins with the transformation step, it is assumed that certain preprocessing steps, i.e., steps 1-3 in the data mining process, have already been completed [38]. Both the machine learning and data mining processes also assume that the initial dataset has already been created. We find these assumptions to be significant, as the extraction and transformation of even a single feature from a binary file requires domain expertise.
2. Of the 39 publications reviewed in the 2017 survey a total of 3, or roughly 8% [39]–[41], used binaries to support their research (including one sample that used both binary and source code samples), while the remaining 92% relied on source code level samples. In the 2020 survey, a total of 19 works published between 2013 and 2019 were reviewed. In this survey, 4 out of the 19 used compiled code samples to support their work [40], [42]–[44][1], 2 used a mix of binary and source code samples [45], [46], and the remaining 13 used source code samples – 6 out of 19, or roughly 32%, is a noteworthy increase from the mere 8% in the 2017 survey of conventional machine learning and data mining research that used binary samples [27]. Yet, the disparity between vulnerability analysis using source code versus compiled code persists.

We believe that the difference between vulnerability analysis using source code verses compiled code may be correlated to the fact that during the compilation process much of the semantic and syntactic information offered by high-level languages (e.g., C, Java, etc.) is lost, which results in samples that are more difficult to analyze [21], [47], [48]. In turn, this level of difficulty has an inverse relationship with the number of researchers using binary samples to support their work— as observed in the 58 papers reviewed in the two surveys where only 3% leveraged binary samples [27], [28].

[1]Grieco, et al. [40] was referenced in both the 2017 and 2020 survey

Yet, there are situations when the analysis of binary samples is desirable or necessary. For example, a user may need to validate that properties proven by analyzing a program's source code still hold after the program has been compiled, or they simply may not have access to a program's source code [21], [47], [49], [50].

1.2.3 Challenges

The influx and array of VATTs to support the vulnerability analysis process, including those that leverage machine learning and data mining, has resulted in two major challenges that are addressed in our research. We summarize these challenges as follows:

Challenge 1: *There is no comprehensive method to assess the efficacy of VATTs*. We have no comparative metrics of the types of vulnerabilities each tool can or cannot find, how quickly they can do so, or the number of false positives they report. So, when a consumer wants to use a vulnerability analysis tool, they have no method of comparing the alternatives to determine which is a good choice for a specific use case. When a developer wants to modify their tool or enter the market with a new one, no standard method is available to compare their VATT against existing ones. Consequently, consumers and developers compare tools in a disjointed fashion, which can result in misinformation, subjectivity, redundancy, and inconsistency.

Based on the limitations of existing datasets and the absence of a community-accepted benchmark, the community is in need of an overarching framework to guide the implementation of BVATTs. The framework for BVATT should include both the core characteristics of the benchmark, and metrics that can be used to quantitatively compare VATTs.

Challenge 2: *There is no tool to synthesize the at-scale extraction, cleaning, and transformation of features commonly used in data mining and machine learning-based vulnerability analysis* Thus, current research in this area is limited to a niche group of computer scientists with the requisite domain expertise in machine learning and binary vulnerability analysis. This disconnect acts as barrier to both security and machine learning researchers, and ultimately limits advancements in a promising area of vulnerability analysis.

Instead, we need to get to a place where a vulnerability analyst can succinctly extract common features from hundreds of binaries using a single tool. The tool should also clean

and transform those features so that they're compatible with machine learning and data mining techniques.

1.3 Contributions

This study makes several contributions.

1. We synthesize 839 CWEs with over 75,000 CVEs to determine the relative proportions of vulnerability instances and weakness types in the real-world.
2. We analyze four popular software vulnerability datasets, and show that none accurately represents vulnerability instances and weakness types as they occur in the wild.
3. We provide a framework for a benchmark for vulnerability analysis tools and techniques (BVATT).
4. We provide the tool, `BiSECT` (Binary Synthesized Extraction, Cleaning, and Transformation), to extract common features from binaries and transform them into a format compatible with data mining and machine learning techniques.
5. We use the output of `BiSECT` to assess the efficacy of two representation models (and corresponding classifiers), *fastText* and *Doc2Vec*, when they are given the task of labeling potentially vulnerable functions.
6. We examine the impact of using balanced and imbalanced datasets when training the *fastText* and *Doc2Vec* classifiers on binary samples.
7. We use `BiSECT` to prepare $fuzzyInstruction$ sequences in support of binary vulnerability analysis.
8. We use the $fuzzyInstruction$ sequences to train a classifier able to label function-level vulnerabilities in a balanced hybrid-synthetic assembly code base with 96.4% accuracy, 97.8% precision, 94.8% recall, a False Positive Rate (FPR) of 2.1%, False Negative Rate (FNR) of 5.2%, and F1 of 96.3%.
9. We demonstrate how `BiSECT` can be leveraged to identify the most diverse test cases in a test suite.

1.4 Organization

The organization of this dissertation is as follows.

- *Chapter 1* provides an introduction and motivation for the research.
- *Chapter 2* defines terms used throughout this work, provides a brief overview of vulnerability analysis techniques, and describes current methods to compare vulnerability analysis techniques. Chapter 2 concludes with an overview of research related to the use of data mining, machine learning, and deep learning techniques to support the vulnerability analysis process.
- *Chapter 3* addresses **Challenge 1** as specified in 1.2.3, and provides a framework for BVATT. Chapter 3 also identifies the limitations of current datasets, including the misrepresentation of weakness types and vulnerability instances. Chapter 3 identifies what a representative suite of benchmark problems should look like, and finally, sets the stage for `BiSECT`. Contributions 1-3 are detailed in Chapter 3.
- *Chapter 4* addresses **Challenge 2**, and describes a new tool, `BiSECT` (Binary Synthesized Extraction, Cleaning, and Transformation). `BiSECT` was created to support vulnerability research, and provides a user-friendly and repeatable means to synthesize the extraction, cleaning and transformation of common features from binary files in a format compatible with data mining and machine learning techniques. To demonstrate the various utilities of `BiSECT`, two example applications are provided in Chapter 4. Contributions 4-9 are detailed in Chapter 4.
- Finally, *Chapter 5*, presents a summary of our work. Chapter 5 includes a review of the major challenges addressed throughout this research, followed by a discussion of our findings, and finally, with recommendations for future research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

2.1 Introduction

This chapter provides the necessary background to set the stage for the remainder of our research. To summarize, our research is comprised of two parts: a framework for BVATT, and a means to extract common features from binary samples in support of machine learning and data mining based vulnerability research, i.e., the `BiSECT` tool.

In Section 2.2 we precisely define key terms associated with software vulnerability analysis and machine learning as they will be used in this work. These terms are followed by a brief history and overview of key vulnerability analysis techniques in Section 2.3. We review early static analysis techniques in Section 2.3.2 and dynamic analysis techniques in Section 2.3.3. In Section 2.3.4 we review more recent vulnerability analysis techniques that leverage machine learning and data mining. Then, current vulnerability analysis techniques that use deep learning and neural networks are explored in Section 2.3.5. After reviewing vulnerability analysis techniques, we explore current methods to compare VATTs in Section 2.4. Finally, Chapter 2 is concluded with an examination of tools that can be used to extract one or more features from compiled code (i.e., binary) samples.

2.2 Terms

In this section we clarify our use of terms referenced throughout this work that may be ambiguous due to multiple and sometimes vague definitions. These terms include *bug*, *vulnerability*, *data mining*, and *machine learning*. Related terms are also included for clarity. In several cases we introduce a term by attempting to pinpoint its origin. We feel that providing the etymology of the terms helps provide a bit of additional context to the reader. For example, the latter part of our research (Chapter 4) deals heavily with terms and concepts related to Artificial Intelligence (AI), but not necessarily AI as it was defined in the mid-1900s. Where appropriate, we have also included some background information relevant to AI to help set the stage for terms directly relevant to our work (e.g., machine

learning).

2.2.1 Bug

Grace Hopper provided one of the earliest recorded uses of the term *bug* in the context of a computer. In 1947, a moth that was caught between relay contacts caused an issue with the Harvard Mark II computer. Hopper taped the moth to the computer's log book and next to it noted (Figure 2.1), "First actual case of bug being found" [51].

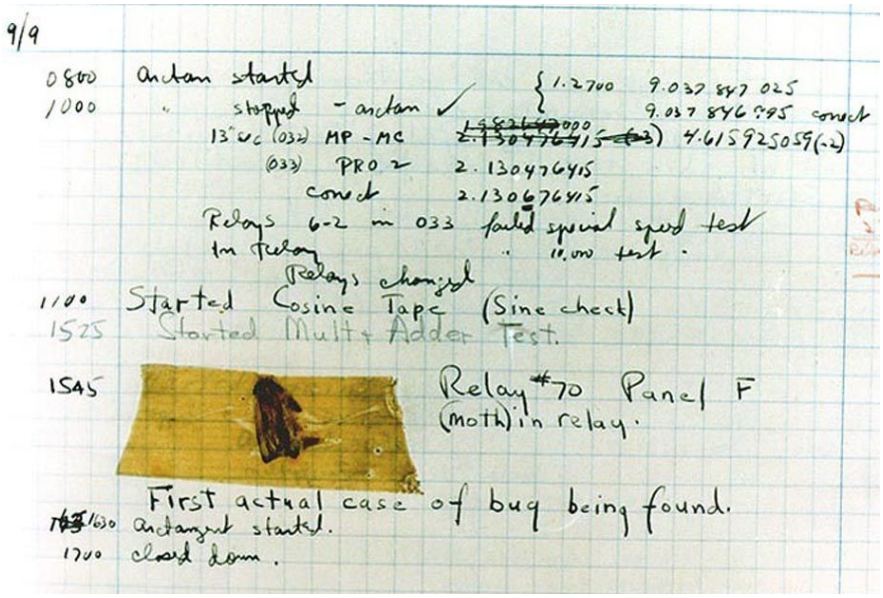


Figure 2.1. Grace Hopper log book with computer bug. Source [51].

However, the term *bug* can be traced back even further. In the 1870s Thomas Edison used it to describe various issues with his inventions. In one example Edison states, "This thing gives out and then that 'Bugs'—*as such little faults and difficulties are called*— show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached" [52]. In, *Etymology of the Computer Bug: History and Folklore, 1987*, Fred Shapiro defines a *bug* as, "a defect in hardware or software" [53]. We draw on these early definitions of the term, and define a **bug** as: a fault or defect in the hardware or software of a machine.

We refer to the following terms in accordance with their definitions provided by Melliar-Smith [54] and Denning [55]:

- A **failure** is an event at which a system violates its specifications
- An **error** is an item of information which, when processed by the normal algorithms of the system, will produce a failure
- A **fault** is a mechanic algorithmic defect which may generate an error
- A **defect** is an imperfection or deficiency

Using these definitions we refer to the term **software bug** as a defect in the software of a machine.

2.2.2 Software Vulnerability

The CVE defines a **vulnerability** as [56],

> a weakness in the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that, when exploited, results in a negative impact to confidentiality, integrity, or availability.

Vulnerabilities therefore are the subset of bugs that:

1. are exploitable; and,
2. when exploited, result in a negative impact to confidentiality, integrity, or availability.

Exploitable in this context simply means able to be used for a particular purpose. A **software vulnerability** is a software bug that, when exploited, results in a negative impact to confidentiality, integrity, or availability. We refer to the following terms in accordance to their definitions in *United States Code, 2006 Edition, Supplement 5, Title 44* [57]:

- **Confidentiality:** Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information.
- **Integrity:** Guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity.
- **Availability:** Ensuring timely and reliable access to and use of information.

2.2.3 Data Mining

Han, Kamber, and Pei describe **data mining** as the process of discovering patterns from large amounts of data [37]. The authors describe data mining as an iterative form of knowledge discovery comprising seven steps:

1. Data Cleaning
2. Data Integration
3. Data Selection
4. Data Transformation
5. Pattern Discovery
6. Pattern Evaluation
7. Knowledge Presentation

Steps one through four in the data mining process are types of **preprocessing**, where data is prepared for mining (i.e., discovery), evaluation, and finally, presentation [37]. These steps begin after an initial dataset has been gathered.

2.2.4 Artificial Intelligence

Much like the term *bug*, the precise origins of AI are difficult to pinpoint. It seems that a number of ideas related to AI surfaced around World War II. For example, in an unpublished 1939 article titled, *Mechanization and the Record*, Vannevar Bush described a machine (later called "memex") that could operate similar to a human brain [58]. The machine combined lower level technologies to achieve higher level functionality. In 1947, Alan Turing also pondered the possibility of intelligent machines [59]. Turing compared such machines to a student who first learned from his master, then added his own work.

Recently, new perspectives on artificial intelligence have been proposed. For example, Peter Denning and Ted Lewis provide a modern hierarchical classification of learning machines (i.e., levels 0-4) capped by AI aspirations (i.e., levels 5 and 6). Their classification, organized by relative learning power, is shown in Figure 2.2. The authors state that the ultimate goal for AI is, "to construct machines that are at least as smart as humans at specific tasks" [60].

The classification proposed by Denning and Lewis illuminates the limitations corresponding to each learning mechanism. The authors distinguish each level according to the functions

that can be learned, and assert that using such a hierarchy none of the current AI machines actually have intelligence [60]. In the remainder of this section we discuss terms related to two levels in the hierarchy specifically relevant to our work: supervised and unsupervised learning.

A Machine-Intelligence Hierarchy	
0	basic automation
1	rule-based systems
2	supervised learning
3	unsupervised learning
4	multiagent interactions
5	creative AI
6	aspirational AI

Figure 2.2. Hierarchy of artificial intelligence machines. Source [60].

Machine Learning

Machine learning is a branch of AI that refers to machines that acquire new functions by being trained from large data sets. Major focus areas in machine learning include: prediction, classification, and clustering tasks [38]. Two common machine learning types (sometimes referred to as "scenarios"): supervised and unsupervised learning, are identified as levels 2 and 3 in the Denning and Lewis hierarchy [60]. In supervised and unsupervised learning the basic objective is to make predictions about an outcome given some input. Two major differences between these learning scenarios are the type of training data that is made available to the machine, and the order in which the *training* and *test* data are used. In this context, the **machine** simply refers to the prediction model that is used, the **training data** are the data used to train the machine, and the **test data** are the data used to evaluate the machine's performance [38]. For the purpose of this research, we focus on these learning scenarios. In **supervised learning** predictions are made about new data based on a set of labeled training data, whereas, in **unsupervised learning** predictions about new data are made based on a set of unlabeled training data [38].

Machine learning scenarios generally include six steps [37], [38]:

1. Data Preprocessing
2. Model Selection
3. Training
4. Model Evaluation
5. Hyperparameter Tuning
6. Prediction

We call out these steps to provide additional context for our work. More specifically, machine learning and data mining are often used in conjunction to one another. For example, machine learning may be used in the data mining "Pattern Discovery" and "Pattern Evaluation" steps, while the data "Cleaning", "Integration", "Selection", and "Transformation" steps from the data mining process may be used to accomplish the machine learning "Data Preprocessing" step. The `BiSECT` tool (detailed in Chapter 4), that is able to perform feature extraction, cleaning, and transformation is relevant to both the machine learning and data mining processes (thanks to their overlap).

Deep Neural Networks

A subset of both AI and machine learning is *deep neural networks* (this term is often used interchangeably with, *deep learning*). In 1944, Warren McCullough and Walter Pitts proposed the first neural network. The McCullough and Pitts neural network was based on the idea that neural events and relationships in the nervous system could be modeled using propositional and symbolic logic [61]. Aggarwal reflects that neural networks were originally developed to, "simulate the human nervous system for machine learning tasks by treating the computational units in a learning model in a manner similar to human neurons" [62].

In 1958, Frank Rosenblatt published a cornerstone work titled, *Organization of a perceptron*. Rosenblatt's perceptron, depicted in Figure 2.3, modeled a hypothetical nervous system, and was first implemented on the [general purpose] IBM 704 machine [63]. Unlike McCullough and Pitts, whose neural network was based on symbolic logic, this *perceptron* was based on probability theory.

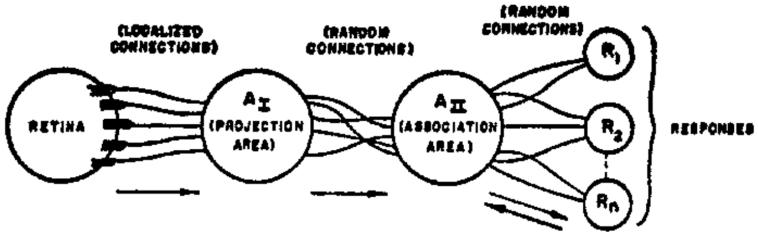


FIG. 1. Organization of a perceptron.

Figure 2.3. Organization of a perceptron. Source: [63].

A **neural network** consists of one or more artificial neurons, which communicate with one another [64]. The output of one artificial neuron is the input to another. Today, a **perceptron** is regarded as the simplest form of a neural network, that is, a neural network that contains a single input layer where a set of inputs is directly mapped to an output using a generalized linear function [62].

Simply stated, **deep learning** models are *deeper* variants of neural networks, that is, they are neural networks with multiple linear or non-linear layers. Deep learning networks can be trained using supervised and unsupervised learning [64]. In the past decade deep learning techniques have made significant strides towards efficiently analyzing complex, noisy, and high-dimensional data— these limitations are often seen in conventional machine learning models [65].

Like both the data mining and machine learning processes, the deep learning process begins with the preprocessing of data [62]. For this reason, much of the functionality provided by the `BiSECT` tool introduced in Chapter 4 is also relevant to deep learning scenarios.

2.3 Vulnerability Analysis Techniques

In this section we explore four major categories of techniques to support the vulnerability analysis process. We first review early static and dynamic analysis techniques, then more recent vulnerability analysis techniques that leverage machine learning, data mining, and deep learning are explored. Throughout this work, we refer to these techniques, and the numerous tools based on them as "VATTs". To date, there is no standard method to compare

VATTs. Consumers and developers currently make comparisons of VATTs by assessing their relative performances using internally developed or publicly available vulnerability datasets [25]. After reviewing the four major categories of VATTs, we examine datasets that can be used to compare them in Section 2.4.

2.3.1 Introduction

In 1936, Alan Turing proved that no algorithm can exist to determine whether an arbitrary program will halt given some input [66]. Turing's proof, describing the Halting Problem, provided a conceptual basis for reasoning about programs. Some may contend that program analysis can be traced back even further to the "Enchantress of Number," Ada Lovelace.

In 1842, Augusta Ada King, Countess of Lovelace (generally referred to as Ada Lovelace), translated an article by Luigi Menabrea that described Charles Babbage's Analytical Engine [20]. In her translation, she included a set of notes that explained the Engine and described its limitations. Ada also provided a step by step symbolic trace (i.e., a program analysis technique) depicting how the Analytical Engine would compute the Bernoulli Numbers [20].

These early examples are presented to both demystify program analysis, which is arguably as old as computing itself, and to set the stage for modern vulnerability analysis. In the remainder of this section we review four overarching types of vulnerability analysis techniques. First, conventional static analysis techniques are introduced, followed by dynamic analysis techniques such as symbolic execution and fuzzing. Then, we review machine learning and data mining techniques, and finally, we explore vulnerability analysis techniques using deep learning.

A Note on Soundness and Completeness

As a result of the Halting Problem [66], and Rice's Theorems [67], we know that most program analysis problems are undecidable in the general case. That is, no such algorithm exists for these types of problems that is both *sound* and *complete*. In the context of vulnerability identification, *soundness* means that if a vulnerability is reported then it is indeed a vulnerability (i.e., there are no false positives), and *completeness* means that all vulnerabilities are found (i.e., there are no false negatives) [68]. Thus, a vulnerability analysis

system that is both sound and complete would be able to report with certainty whether any program is secure, or not— We know that such a system cannot exist. In practice, this means that each vulnerability analysis tool and technique must compromise either soundness, completeness, or both. For example, a VATT with a higher level of completeness may have less soundness– meaning more false positives are reported [68]. Throughout this work we maintain the perspective that the goal is never a perfect solution, rather it is to provide an improved or novel approximate solution.

2.3.2 Static Analysis

Skipping forward over a century past Ada Lovelace's program analysis work, the Lint system was invented in 1978 to statically analyze C source code [69]. During static analysis an abstract representation of the program under test is often created. Such a representation allows the analyst to reason about the program without executing it, and typically includes information about the program's symbols (e.g., variables and types), a control flow graph, and a call graph.

Lint and other first-generation static analysis tools focused on type checking, and applied Unix-like `grep` functionality to determine if potentially unsafe functions, such as `gets`, were found in source code. While having any tool at that time was an improvement over manual analysis, these early static analysis VATTs were criticized for their lack of lexical analysis. The tools could determine the presence of `gets` but they could not determine if it was a function, comment, or variable, and thus they often reported a high number of false positives [69]. For reference, `gets` is used to retrieve a single line of input from standard input, and is known to be vulnerable to buffer overflows [70].

Second generation static analysis tools such as PScan [71], RATS [72], ITS4 [73], Splint [74], and Flawfinder [75] offered a number of improvements including lexical analysis and tokenization to provide a higher level of semantic insight. These tools could differentiate between the various uses of `gets`, yet, there seemed to be a direct correlation between the enhanced functionality and the number of false positives the tools reported. Since static analysis tools do not describe *how* a vulnerability can be exploited, users must manually investigate every potential bug reported to determine if a legitimate vulnerability is present. These problems persisted in the next generation of tools, and the pattern revealed a major

limitation of static analysis, *precision*.

Precision provides a measure of correctness for the positive predictions reported by a static analysis tool, and can be calculated using the following formula (precision is discussed in more detail in Sections 3.2 and 4.5.1),

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

Where, TP is the number of true positives, and FP is the number of false positives.

Listing 2.1 provides a simple example demonstrating the weaknesses of static analysis. Static analysis tools would likely report a False Positives (FP) at the conditional $!b$, even though b will never be null. Static analysis tools may miss the out of bounds memory read and write associated with *my_arr*. Even today, static analysis tools may report the existence of these vulnerabilities, but do not provide information on *how* to trigger them.

Listing 2.1: Simple C example depicting weaknesses of static analysis.

```c
int main(void) {
    int b = 2;
    int my_arr[10];

    if (!b) {               //FP
        my_arr[13] = 37;    //FN
        printf("This does not exist: %d", my_arr[13]); //FN
    }
}
```

2.3.3 Dynamic Analysis

In a personal interview, Peter Denning offered the following parallel between dynamic analysis of vulnerabilities and operating systems— Before the process abstraction was accepted, operating systems were seen as assemblies of modules and interfaces. Serious

problems such as race conditions, deadlocks, and busy waiting, etc. were totally invisible in the modules structure. You could plainly see them, however, as problems in interactions among dynamic processes.

Dynamic analysis provides a similar benefit when identifying software vulnerabilities.

During dynamic analysis a program under test is executed in an emulated or actual environment, providing the analyst with run-time data. Like a race condition, vulnerabilities that may have been hidden during static analysis become glaringly evident during dynamic analysis.

In dynamic vulnerability analysis, execution can be performed either symbolically or using concrete test cases, and thus, dynamic analysis techniques can be further delineated into two categories: *symbolic* and *concrete*. In dynamic symbolic execution a program is executed using symbolic values and multiple control flow paths can be examined, whereas in concrete execution a program is executed using concrete values (i.e., test cases) and a single control flow path is explored [19].

In the next few sections we describe two common concrete-dynamic vulnerability analysis techniques: fuzzing and dynamic taint analysis. This is followed by an overview of dynamic symbolic execution.

Fuzzing

On a stormy night in 1989 Barton Miller was logged into his workstation using a dial-up line. As he typed commands into his terminal the storm scrambled some of the characters, and Miller was intrigued to discover that these spurious characters often caused the programs to crash [76]. Miller and his team leveraged this accidental discovery and developed a systematic way to test programs for bugs, a technique they dubbed, *fuzzing*. The discovery team outlined their fuzzing technique in four steps [76],

1. Construct a program to generate random characters, plus a program to help test interactive utilities.
2. Use these programs to test a large number of utilities on random input strings to see if they crash.

3. Identify the strings (or types of strings) that crash these programs.
4. Identify the cause of the program crashes and categorize the common mistakes that cause these crashes.

Fuzzing techniques have advanced in the past 30 years, yet the original four-step procedure remains largely the same. Step 1 describes the generation of random characters or test cases to test a program. Generating test cases continues to be a necessary step in the fuzzing process, and carefully crafting these cases is critical to the success of the fuzzer. Fuzzers can be broadly classified into three categories: *mutation*, *generation*, and *evolutionary*-based fuzzers. Mutation-based fuzzers create test cases by applying mutations to existing data; generation-based fuzzers use a specification (or similar documentation) which describes the target protocol or file format, to generate test cases; and evolutionary fuzzers learn the desired input format by utilizing feedback from each test case iteration [77], [78]. Even with the careful generation of test cases to guide a fuzzer all fuzzer categories have been criticized for their inability to discover anything other than superficial or shallow bugs [79]–[81]. To mitigate this limitation, coverage-based fuzzers like American Fuzzy Lop (AFL) focus on maximizing the amount of code executed in a target application [82]. Unfortunately, coverage-based fuzzers are still unable to understand the underlying logic of the code they're executing [83]. While they can effectively find vulnerabilities caused by a single input vector, these fuzzers struggle to discover more complex vulnerabilities, or vulnerabilities that require more than one input vector in order to be triggered [83].

Listing 2.2 demonstrates a limitation of fuzzing techniques. In this example, adapted from Stephens et al., a configuration file is parsed which contains a magic number received via in input stream. If the received input contains an incorrect or poorly formatted magic number, the program exits. Otherwise, the program control flow varies based on specified conditions and a memory corruption flaw may occur. The fuzzer will likely get stuck during the first comparison with the magic number and never reach the memory corruption errors [80].

Dynamic Taint Analysis

In 1976, Dorothy Denning explored the use of a lattice model for secure information flow [84]. Denning argued that a formal model was necessary to prevent the unauthorized flow of

Listing 2.2: C example demonstrating a weakness of fuzzing. Adapted from [80]

```c
int main(void) {
    config_t *config = read_config();
    if(config == NULL) {
        puts("Configuration syntax error");
        return 1;
    }

    // magic number check << fuzzer stuck here
    if(config->magic != MAGICNUMBER) {
        puts("Bad magic number");
        return 2;
    }
    initialize(config);

    char *directive = config->directives[0];
    if (!strcmp(directive, "crashstring")) {
        program_bug();
    } else if (!strcmp(directive, "set option")) {
        set_option(config->directives[1]);
    } else {
        good_function();
    }
}
```

information in a computer system. In 1977, Dorothy and Peter Denning proposed a compile-time method, leveraging the lattice model and static analysis, to certify secure information flow in programs [85]. The pioneering work of the Dennings laid the foundation for modern information flow control techniques including dynamic taint analysis and symbolic execution.

Dynamic taint analysis focuses on determining which computations are impacted by tainted sources. Values are considered *tainted* if their computation depends on data derived from

a predefined taint source, otherwise the value is considered *untainted* [19]. Dynamic taint analysis can be used to observe how information flows between untrusted sources and sinks. False positives are called *overtainting*, and occur during dynamic taint analysis when a value that is not a derivative of a taint source is mistakenly marked as tainted. Conversely, false negatives, or *undertainting*, occur when the information flow between an untrusted source and sink is not reported during analysis [19]. To use taint analysis the user must specify a policy that includes information about taint *introduction*, *propagation*, and *checking*. Taint introduction simply refers to how a new taint is introduced to the program, taint propagation specifies the current taint status of data derived from tainted or untainted sources, and taint checking details when to modify the runtime behavior (e.g., halt execution) of the program based on current taint propagation status [19].

Unless otherwise specified, during dynamic taint analysis taint may only be added and is not removed. As the program executes this can lead to an exponential increase in the number of values that are tainted, and a decrease in precision occurs [19]. Additionally, pure taint analysis only reports the presence of tainted values using a single execution stream at a time. Thus, it does not compute control-dependencies that would only be discovered by reasoning about multiple execution paths [19].

Listing 2.3 demonstrates one of the weaknesses of dynamic taint analysis. In this example, there is an implicit control flow between a and b. While there is no direct or explicit flow (e.g. where $a = b$), when the code is executed b implicitly obtains the value of a. This transfer may cause *undertainting* to occur. When $a = false$ and a is tainted, the first branch will be executed, but the second branch will not be executed. In this case, b is not tainted while b depends on a [86].

Dynamic Symbolic Execution

Unlike concrete execution which uses concrete values to test a program, during dynamic symbolic execution a program is executed using *symbolic* values. Symbolic values are used to create a logical formula to represent the execution state of the program. This process allows a program to be reasoned about using many different inputs concurrently [19]. When a program branches the symbolic execution engine explores both paths and records a set of constraints, or path conditions, that must hold for each path to be taken. Path conditions contain information about the constraints on inputs that were used to reach a specific program

Listing 2.3: C example demonstrating a control-flow limitation of taint analysis. Adapted from [86].

```c
int main(void) {
    bool a;
    int usr_val;
    scanf("%d", &usr_val);
    a = usr_val;

    bool b = false;
    bool c = false;

    if (!a) {
        c = true;
    }

    if (!c) {
        b = true;
    }
}
```

state [87]. Then, when the path eventually terminates or a vulnerability is discovered, a constraint solver is invoked to produce a concrete path from the generated symbolic path and corresponding constraints. This path then becomes a concrete test case that can be used to lead execution to a precise point in the program.

Like other testing methodologies symbolic execution faces numerous challenges in practice. Complex memory management, library and system calls, path explosion, and constraint solver limitations are just a few of the reasons that symbolic execution remains a costly testing technique that does not scale well to real-world applications [19]–[21], [80], [88].

For example, program loops and function calls cause a symbolic executor to branch numerous times and can create an explosion in the number of paths and states to explore. This phenomena

is commonly referred to as the *path explosion problem* [19]. The path explosion problem has been shown to be directly correlated to the amount of computational memory and time required to symbolically analyze a program. As the number of states nears infinity during analysis, the ability to exhaustively analyze a program via symbolic execution can quickly become intractable, i.e., no efficient algorithm exists to solve it. The path explosion problem creates a significant limitation for using symbolic execution to discover vulnerabilities in real-world applications [19], [21], [80], [88].

The path explosion problem is demonstrated in Listing 2.4. In this example, from [20], the for loop results in 2^{100} paths for the symbolic execution engine to explore. The path explosion problem is exacerbated in real-world applications where loops and recursive functions have much greater frequency and complexity, and often results in dynamic symbolic execution only discovering shallow bugs. There have been several attempts to mitigate the path explosion problem, but none has provided an absolute solution [88].

In addition to conventional static and dynamic techniques researchers also leverage conventional machine learning and data mining techniques to support vulnerability analysis. Recent work in these areas are discussed in Section 2.3.4.

2.3.4 Machine Learning and Data Mining

In 2017, Ghaffarian and Shahriari surveyed conventional machine learning and data mining techniques used to support vulnerability analysis. They partitioned the works reviewed into four major categories [27]:

1. Vulnerability Prediction Models based on Software Metrics
2. Anomaly Detection Approaches
3. Vulnerable Code Pattern Recognition
4. Miscellaneous

Ghaffarian and Shahriari further delineated the reviewed work into those that make use of program syntax and semantics and those that do not. Sethi provides the following definitions for *syntax* and *semantics*, ”The *syntax* of a language specifies how programs in the language are built up. The *semantics* of the language specifies what programs mean“ [89].

Listing 2.4: C example demonstrating the path explosion problem. Source [20].

```c
int main(void) {

    char buf[32];
    char *data = read_string();

    // Symbolic execution will suffer from path explosion
    int count = 0;

    for (int i = 0; i < 100; i++) {
        if (data[i] == 'Z') {
            count++;
        }
    }

    if (count >= 8 && count <= 16) {
        memcpy(buf, data, count * 20);        // buffer overflow
    }
    return 0;
}
```

Syntax in the context of a program refers to the program's structure, while semantics has to do with the program's meaning. Techniques leveraging program syntax include anomaly detection approaches and vulnerable code pattern recognition. Semantics are often gathered by studying input/output relationships. Techniques leveraging program semantics include vulnerability prediction models based on software metrics, and a number of miscellaneous works that do not fit well into any of the other categories.

In total, the authors surveyed 39 machine learning and data mining approaches related to vulnerability identification and analysis; the results of the survey are summarized in Table 2.1.

Table 2.1. 2017 Survey of machine learning and data mining based vulnerability analysis techniques. Source Code is denoted by SC. Adapted from [27].

Ref.	Year	Granularity	Predictors Used	Techniques Used
Vulnerability Prediction Models based on Software Metrics				
[39]	2010	Binary	Complexity, code churn, coverage, dependency measures, organizational measures	Binary classification using logistic regression
[90]	2010	SC	Complexity, code size, security resources indicator (SRI)	Spearman's rank correlation
[91]	2011	SC	Execution complexity metrics	Binary classification using logistic regression
[92]	2011	SC	Complexity, code churn, and developer activity (CCD) metrics	Binary classification using logistic regression, random forest, naïve bayesian, and bayesian network
[93]	2013	SC	Complexity	Naïve bayesian, bayesian network, IBK, classification via clustering, random tree, logistic, J48 and random forest with some meta classifiers
[94]	2013	SC	Commits (interactive churn, and community dissemination of vulnerable commits)	Data mining
[95]	2014	SC	Developer-activity	Data mining

Continued on the next page

Table 2.1: (Cont.)

Ref.	Year	Granularity	Predictors Used	Techniques Used
[33]	2014	SC	LOC, number of functions, complexity metrics, Halstead's volume, total external calls, fan-in, fan-out, internal functions, external functions, external calls, text mining	Random forest
[96]	2015	SC	Code repository meta-data, code metrics	Data mining
[41]	2015	Mixed	Churn, complexity, coverage, dependency, legacy data, size metrics	Logistic regression, naïve bayesian, recursive partitioning, support vector machine, tree bagging, random forest
[97]	2016	SC	Lines of code, cyclomatic complexity, count path, nesting degree, information flow, calling functions, called-by functions, number of invocations	Welch t-test (for indiv. features), correlation-based, wrapper, and PCA (for feature combinations), classification using logistic regression, naïve bayesian, random forest, and support vector machine
Anomaly Detection Approaches				
[98]	2001	SC	Developer beliefs metrics (API usage pattern)	z-score, data mining— template-based rule extraction
[99]	2005	SC	Software revision history (API usage pattern)	Data mining (association rule mining)
[100]	2005	SC	Explicit programming rules (API usage pattern)	Data mining (frequent closed itemset mining)
[101]	2007	SC	Legal method/function call sequences (API usage pattern)	Data mining (frequent closed itemset mining)

Continued on the next page

Table 2.1: (Cont.)

Ref.	Year	Granularity	Predictors Used	Techniques Used
[102]	2007	SC	Frequent partial function order from API usage pattern	Data mining (frequent partial-order itemset mining)
[103]	2008	SC	Neglected conditions discovered using program dependency graphs	Heuristic maximal frequent subgraph mining algorithm (HMFSM)
[104]	2009	SC	Neglected conditions	Data mining (frequent itemset mining)
[105]	2010	SC	Temporal properties (flow of values) between function calls (API usage pattern)	Data mining (frequent closed itemset mining)
[106]	2013	SC	Missing checks in SC	Bag-of-words, k-nearest neighbors, cosine similarity
Vulnerable Code Pattern Recognition				
[107]	2011	SC	Vulnerability extrapolation of functions using API usage patterns	Supervised classification: cosine similarity of matrix values using principle component analysis (PCA)
[108]	2012	SC	Vulnerability extrapolation using partitioned abstract syntax trees	Supervised classification: latent semantic analysis of matrix using singular value decomposition (SVD)
[109]	2012	SC	Function level input validation code patterns collected using backward static program slices	VPM using naïve bayesian and multi-layer perceptron (MLP)
[110]	2014	SC	Source-code	VPM using bag-of-words, n-gram analysis, naïve bayesian, and random forest

Continued on the next page

Table 2.1: (Cont.)

Ref.	Year	Granularity	Predictors Used	Techniques Used
[111]	2014	SC	Joint data structure including: abstract syntax trees, control flow graphs and program dependence graphs into code property graph (CPG)	Data mining of CPG
[112]	2015	SC	Code property graph (incl. abstract syntax trees, control flow graphs, and program dependence graphs)	Data mining of CPG
[113]	2015	SC	Continuous sequences of tokens in SC	*n*-gram analysis using statistical feature selection and support vector machines
[40]	2016	Binary	Execution traces and function call sequences	Supervised classification— multilayer perceptron (MLP) (logistic regression using hidden layers)
Miscellaneous				
[114]	2007	Binary	Control flow graph	Black box fuzzing using genetic algorithm based on Dynamic Markov Model fitness heuristic
[115]	2012	SC	Textual and syntactical information	Data mining
[116]	2013	SC	Control flow, tracing, variable state	Computational intelligence techniques
[117]	2014	SC	AST, Control-flow information	Data mining, taint analysis
[118]	2014	SC	Application development frameworks (ADF)	Data mining
[119]	2014	SC	Textual and syntactical information	Data Mining

As summarized in Figure 2.4, of the 39 publications reviewed a total of 3, or roughly 8% ([39]–[41]) used compiled samples to support their work (including one "mixed" that used both binary and source code samples), while the remaining 92% relied on source code level samples.

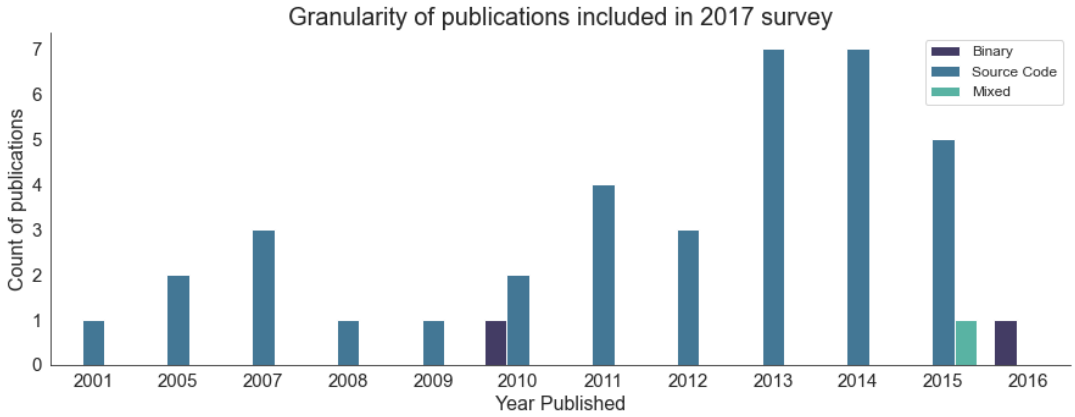


Figure 2.4. Granularity of work reviewed in Ghaffarian and Shahriari 2017 survey. Of the 39 publications reviewed, 8% leveraged binary samples. Adapted from [27].

Figure 2.5 provides a treemap depicting the count of each predictor used in the research reviewed in the survey. The most common predictors included API usage, complexity, and function metrics. We provide a brief overview of the three primary categories that were reviewed in the survey in the following sections. These categories include software metrics, anomaly detection approaches, and vulnerable code pattern recognition.

Software Metrics

In the mid-1960s, the number of "lines of code" was used as a measure of programmer productivity. This is often regarded as the first software metric [120]. The *IEEE Standard Glossary of Software Engineering Technology* defines a **metric** as, "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [121].

The term **software metrics** can be used to describe the collection of various activities related to quantitative measurement in software [120]. Examples of software metrics include lines of code, cyclomatic complexity, bugs per lines of code, program execution time, and cohesion.

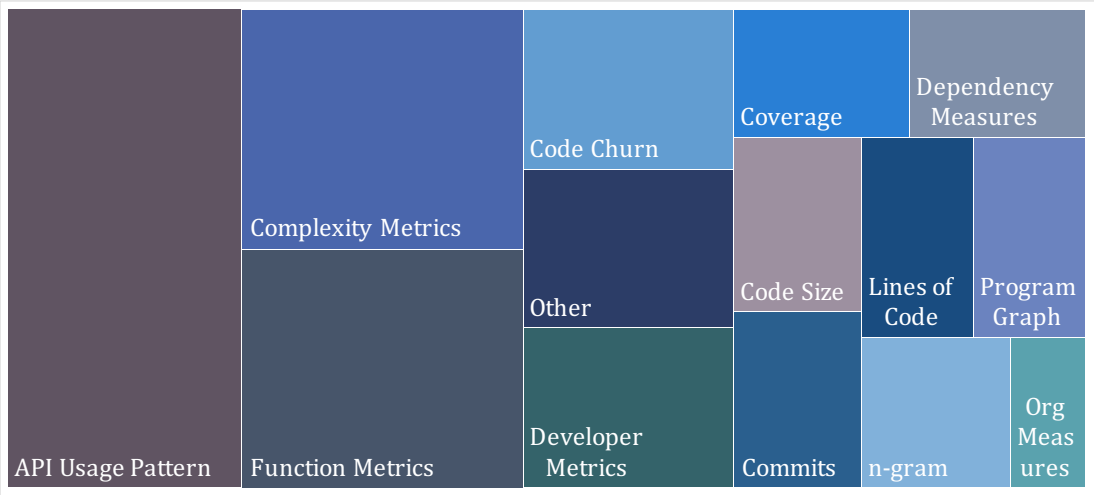


Figure 2.5. Treemap of predictors used in vulnerability analysis techniques that use machine learning or data mining according to a 2017 survey. Adapted from [27].

These metrics (and others) have been used in numerous studies, with varying levels of success, related to vulnerability identification and analysis [27], [33]–[36].

After software metrics have been gathered, learning or classification algorithms are applied to the dataset to create a **Vulnerability Prediction Model (VPM)**, that is, a model used to classify software components as [likely] vulnerable or not vulnerable [27]. This process is described in Figure 2.6 [122].

In addition to software metrics, text mining (also referred to as text analysis) approaches can also be used to create VPMs [123]. Like other software models (e.g., Defect Prediction Models), the primary objective for VPMs is not to identify true vulnerabilities, rather it is to support the testing and code review process by prioritizing testing efforts based on the likelihood that a software component is or is not vulnerable [123]. Software metrics are often readily available or easily obtained throughout the software engineering process for source code, making this type of vulnerability analysis attractive for its convenience. However, VPMs suffer from multiple challenges in practice.

For example, *imbalanced class data*, or datasets with a disproportionate ratio of vulnerable to non-vulnerable code can skew VPM results [27], [124]. Additionally, VPMs based on

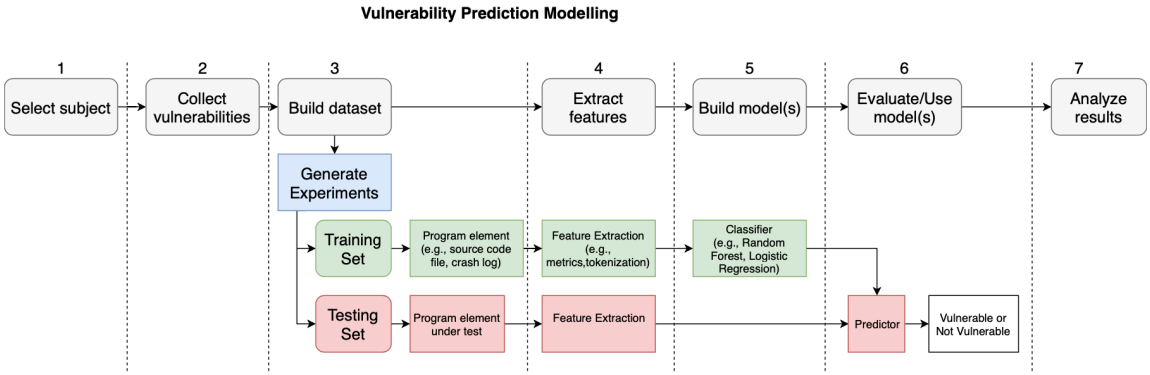


Figure 2.6. Vulnerability Prediction Model (VPM) Pipeline. Source [122].

software metrics are often built for a single project, and have minimal success when applied to another project (this is sometimes referred to as *cross-project VPM, or cross-project prediction*). Researchers have attempted to mitigate this limitation by using feature or attribute dimensionality reduction. Dimensionality reduction techniques can be used to reduce *overfitting*, which occurs when a VPM is too specific to the training dataset that was used to create it [124]. Finally, code metric based VPMs have been criticized for having a lack of semantic information included in the model, however, researchers have begun to apply deep learning techniques to collect rich semantic and syntactic information from source code [125].

Anomaly Detection Approaches

In 1986, Dorothy Denning described an approach to detect security violations using abnormal patterns of a system's usage, i.e., *anomalies* [126]. Denning's work became the basis for early network intrusion detection systems, many of which still leverage anomaly detection approaches today. In the past decade researchers have applied anomaly detection approaches to other areas of security, such as the identification and analysis of vulnerabilities. Similar to the approach developed by Denning, anomaly detection approaches within the context of software vulnerability analysis aim to identify patterns, and deviations or anomalies, from those patterns.

Ghaffarian and Shahriari surveyed a number of anomaly detection approaches, most leveraged data mining techniques to mine API usage and identify patterns in source code, and a few mined graphical representations of the source code (e.g., program dependency graphs) [27]. In both scenarios API rules, legal operations, itemsets, or invocation sequences can be mined to capture normal or expected call patterns. These patterns are abstracted and represented by Finite State Machines (FSM), which are then used to quickly identify anomalies.

As an example, Figure 2.7 describes an anomaly-based vulnerability detection approach proposed by Wasylkowski [101]. In their approach, legal method or function call sequences are mined from the API, and used to develop a model for each method. This model is abstracted to create a general usage model, which in turn is used to identify and flag anomalies to the normal usage pattern [101].

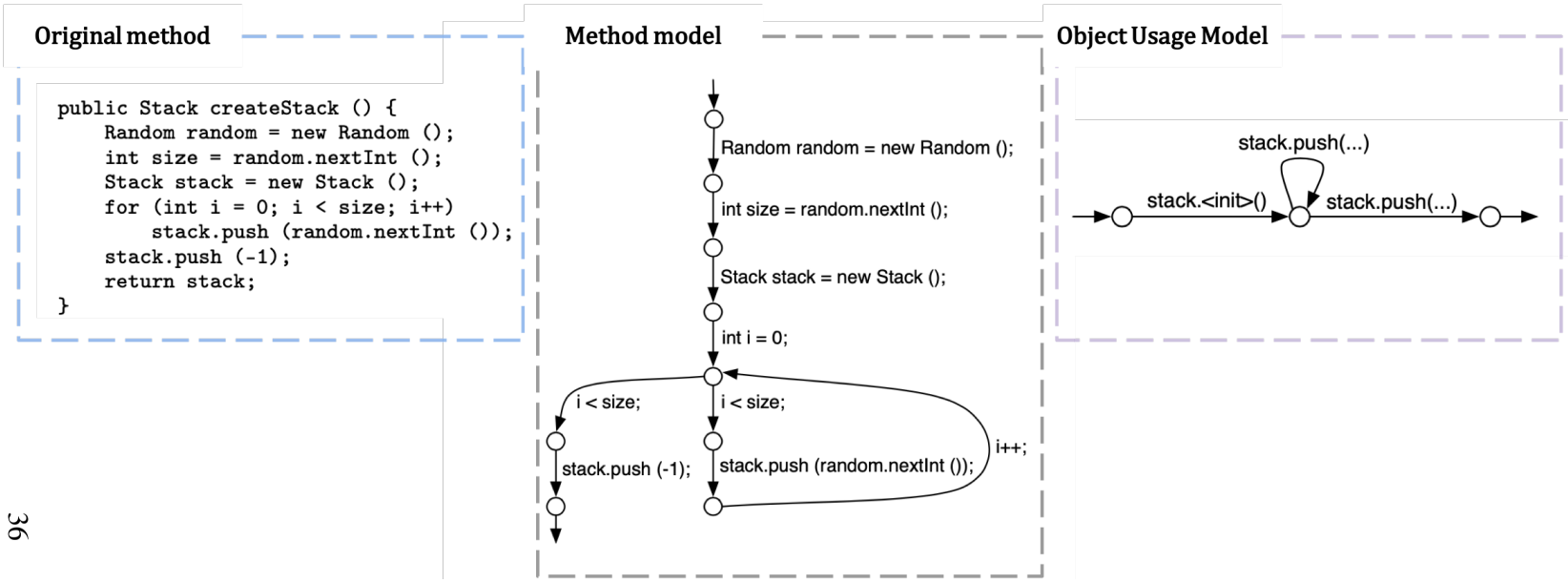


Figure 2.7. Example of a vulnerability identification approach using anomaly detection. Source [101].

Anomaly detection approaches are criticized for having a high false-positive rate, meaning, that they flag non-vulnerable code as vulnerable. This is sometimes attributed to difficulty in determining which anomalies are relevant to security [27].

Code Pattern Recognition

In code pattern recognition, features are extracted from vulnerable code and used to identify patterns [27]. A **feature** is simply a measurable property or characteristic; this term is often used interchangeably with the term **attribute**. In this context, software metrics as they are defined in the previous section are a subset of software features. Approaches to extract features may include traditional static and dynamic analysis, code parsing, and text mining [27].

After the features have been gathered they are transformed into vectors so that they may be easily processed by learning algorithms. All individual vectors are combined to create a single, n-dimensional vector of features called a **feature vector**, where n is equal to the number of features. The feature vector provides a cohesive representation of each sample. Finally, the same features are extracted from a new sample and a comparison is made to determine whether the new sample shares patterns with the known vulnerable code. This classification process is typically accomplished via supervised or unsupervised learning using data or control flow representations of the source code. In this way, vulnerable code pattern recognition techniques are somewhat similar to anomaly detection approaches. In anomaly detection, the objective is to identify anomalies to normal use patterns, whereas, in vulnerable code pattern recognition, the objective is to identify similar patterns.

Current vulnerable code pattern recognition techniques have a number of limitations, such as: high false-positive rates, limited information about the types of vulnerable components (e.g., a CWE or CVE association), and vague reporting (e.g., reporting that a source code file contains a similar pattern to a known vulnerability, but not precisely *where*) [27].

In the next Section 2.3.5 we review a final category of vulnerability analysis– that is, vulnerability identification techniques based on deep learning and neural networks.

2.3.5 Deep Learning and Neural Networks

Neural networks and deep learning models have had widespread impacts on areas such as image and speech recognition, information security, big data processing, cloud computing, internet routing, and forensic science [127]. Researchers have also begun to apply neural networks and deep learning techniques to software vulnerability identification and analysis. A comprehensive survey of notable work in this area was published in 2020, by Lin, et al. [28]. In their survey, the authors review 19 vulnerability detection methods that leverage deep neural networks published between 2013 and 2019. The authors partitioned the surveyed work into four primary categories [28]:

1. Graph-based methods
2. Sequence-based methods
3. Text-based methods
4. Mixed methods

Table 2.2 summarizes the results from the 2020 survey. Of the 19 works reviewed, 5 used compiled binary samples to support their work, 2 used a mix of binary and source code samples, and the remaining 12 leveraged source code samples. This (37%) is a significant increase from the mere 8% of conventional machine learning and data mining research that used binary samples, as discussed in Section 2.3.4. This difference could be a result of the nature of deep learning methods, which often does not require the user to manually extract interesting features. In turn, the automation offered by deep learning may decrease the domain expertise required to analyze vulnerabilities in binary files.

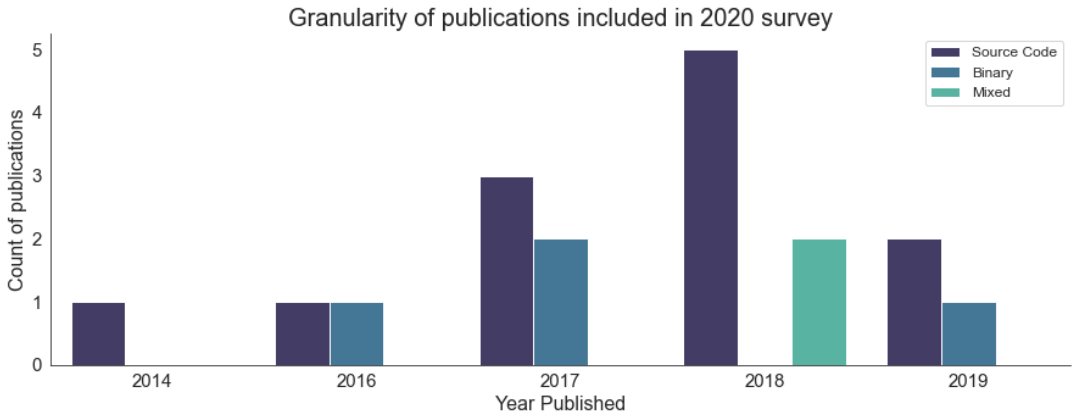


Figure 2.8. Granularity of work reviewed in Lin et al. survey. Of the 19 publications reviewed, 37% leveraged binary samples to support their work. Adapted from [28].

Table 2.2. 2020 Survey of deep learning and neural network based vulnerability analysis techniques. Source Code (SC). Adapted from [28].

Ref.	Year	Granularity	Predictors Used	Techniques Used
Graph-Based Methods				
[128]	2016	SC	Function level AST	Deep Belief Network (DBN)
[129]	2017	SC	Serialized function level AST	BLSTM neural network
[130]	2018	SC	Serialized function level AST	AST tokens encoded tokens by Continuous Bag-of-Words neural embeddings, vulnerability representation generated using sequential deep learning classifier BLSTM
[125]	2017	SC	Function level AST	LSTM
[131]	2018	SC	AST, CGF, and PDG data	Bidirectional Gated Recurrent Unit (BGRU)
Sequence-Based Methods				
[40]	2016	Binary	Execution traces and function call sequences	Supervised classification— multilayer perceptron (MLP) (logistic regression using hidden layers)
[42]	2017	Binary	Function call sequences	Convolution neural network (CNN), LSTM, and CNN-long short term memory (CNN-LSTM)
[31]	2018	SC	Semantically related lines of code (“code gadgets”)	BLSTM
[132]	2019	SC	Semantically related lines of code (“code gadgets”) including control-dependence	Building-block BLSTM

Continued on the next page

Table 2.2: (Cont.)

Ref.	Year	Granularity	Predictors Used	Techniques Used
Text-Based Methods				
[133]	2014	SC	AST	Nearest neighbor, k-means clustering, Tree-based Convolutional Neural Network (TCNN),
[43]	2017	Binary	Instructions	Instruction2Vec to vectorize assembly used as input to text-convolutional neural network model (Text-CNN)
[45]	2018	Mixed	Function level CFG, instructions, opcodes, definitions and variable usage	Convolutional neural network and recurrent neural network used as input to random forest and bag-of-words classifiers
[44]	2019	Binary	Opcode and instruction information	Maximal Divergence Sequential Auto-Encoder
[134]	2017	SC	Complexity	Memory network-based model
[135]	2018	SC	Complexity	Static analysis, and a memory network-based model
Mixed Methods				
[136]	2018	Binary	Android smali (decompiled files of apks) files in apks.	Deep neural network (DNN)
[46]	2018	Mixed	Function level: CFG, use-def matrix, and (IR) opcode vector	Text-CNN
[28]	2019	SC	Function level, sequence based AST	LSTM used as input to random forest classifier

2.3.6 Discussion

From static to dynamic code analysis, conventional machine learning and data mining to deep neural networks, vulnerability analysis techniques have had nearly 100 years to evolve. Yet, there is no panacea for software vulnerabilities, as every tool and technique continues to suffer from at least one major limitation. For example, early static analysis tools, such as Lint and Flawfinder are criticized for having low precision rates and minimal semantic insights [69], [75]. Dynamic analysis techniques such as fuzzing, dynamic symbolic execution, and taint analysis offer increased semantic insights, but are limited by issues including complex memory management and the path explosion problem [19], [21], [79]–[81], [88]. Machine learning-based vulnerability analysis techniques suffer from dataset imbalances, resource-constraints, and brittle classification differences between vulnerable and non-vulnerable code [27], [28], [137]

In an attempt to mitigate the limitations of individual techniques vulnerability researchers have begun to combine techniques. For example, a team at UC Santa Barbara uses a combination of fuzzing and symbolic execution in their tool, Driller [80]; Mayhem uses symbolic execution techniques in conjunction with taint analysis and fuzzing [138]; and [139] proposes another taint-based approach to fuzzing.

In isolation, researchers have explored the limitations corresponding to VATTs, however, to date, there is no standard method to compare VATTs. Consumers and developers currently make comparisons of VATTs by assessing their relative performances using internally developed or publicly available vulnerability datasets [25]. Next, we'll take a deeper look at some of the datasets currently used to compare VATTs.

2.4 Existing Methods to Compare VATTs

2.4.1 Introduction

In this section, we examine a number of vulnerability datasets that can be used to compare VATTs. Each dataset reviewed is publicly available, and contains test cases with at least one known software vulnerability.

2.4.2 Cyber Grand Challenge

The Defense Advanced Research Projects Agency (DARPA) has sponsored multiple Grand Challenges to spur innovation and push the boundaries of the state of the art in autonomous research areas. From 2013 to 2016 DARPA sponsored a $54 million experiment called the CGC. Similar to previous DARPA Grand Challenges, participants in the CGC were entirely autonomous. On August 4, 2016, seven teams competed in the CGC Final Event (CFE). By improving and combining semi-automated capabilities each team built a single autonomous Cyber Reasoning System (CRS) capable of identifying, exploiting, and patching software vulnerabilities [140].

After the final event, DARPA released the entire corpus of binaries that were used throughout the Challenge to the public. The initial release of binaries was only compatible with the custom CGC DARPA Experimental Cyber Research Evaluation Environment (DECREE) Operating System (OS), but researchers at TrailofBits have since ported the binaries to be compatible with common operating environments including Windows, Macintosh, and Linux [141]. TrailofBits contends that in the absence of a true benchmark the extended CGC corpus, dubbed *CB-Multios*, can be used to provide quantitative metrics for VATTs [141]. The CB-Multios corpus includes the source code for 243 test cases. Each test case was written in C code, and is accompanied by the author's description of the one or more known vulnerabilities within that challenge in addition to the associated CWEs. Each test case is also accompanied by a patch for the vulnerabilities. This allows users of the corpus to compile the known vulnerable, and patched versions of the test cases. The CB-Multios corpus is limited by the fact that it only contains **synthetic**, or engineered test cases. However, the CB-Multios corpus was designed to approximate real-world software. The corpus is also limited by the granularity at which vulnerabilities are labeled— at the file versus line or function level. Known vulnerabilities in the CB-Multios corpus are labeled at the file level, however, users can manually query the source code for the presence of `ifdef PATCHED` which will reveal functions where known vulnerabilities reside.

Table 2.3. CB Multi-OS description

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
CB-Multios	C	243	File	888

2.4.3 Juliet C#, C/C++, and Juliet Java

The Juliet C#, C/C++ and Java test suites were developed by the National Security Agency (NSA) Center for Assured Software (CAS) to assess the capabilities and limitations of static analysis tools [142]. Each test case in the suite contains at least one known, synthetic, security flaw. The latest version of the Juliet test suites contain over 100k individual test cases. Criticisms of the Juliet test suite are that it includes non-exploitable bugs, and overly simplistic or redundant source code [30], [137]. Table 2.4 summarizes the Juliet datasets.

Table 2.4. Juliet C#, C/C++ and Juliet Java descriptions

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
Juliet C#	C#	28,942	Function	28,942
Juliet C/C++	C/C++	64,099	Function	64,099
Juliet Java	Java	28,881	Function	28,881

2.4.4 LAVA-M

The LAVA-M (Large-scale Automated Vulnerability Addition) dataset was developed by researchers at New York University, MIT Lincoln Laboratory, and Northeastern University. The dataset contains a corpus of buffer overflow vulnerabilities that were originally developed to evaluate the LAVA bug injection system. Specifically, the LAVA-M corpus is derived from four copies of the GNU coreutils (v. 8.24) programs ($base64$, $md5sum$, $uniq$, and who), and contains a total of 2,265 synthetic vulnerabilities [25]. The bugs in the LAVA-M corpus were injected into the real-world source code using the LAVA tool. While the LAVA-M contains real-world code, it only contains C-based memory corruption vulnerabilities [25]. Table 2.5 summarizes the LAVA-M dataset.

Table 2.5. LAVA-M descriptions

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
LAVA M	C	4	File	2,265

2.4.5 OWASP Benchmark Project

The OWASP Benchmark was designed by OWASP to evaluate automated software vulnerability identification tools [29]. The test suite is presented as a web application, and each test case contains at least one known synthetic Java vulnerability. The OWASP Benchmark is unique in that it also includes a scorecard that describes the number of true and false positives a tool reports, and uses a Youden index to generate a score for each tool [29]. The OWASP Benchmark is limited by the fact that it only contains Java-related vulnerabilities. Table 2.6 summarizes the OWASP Benchmark dataset.

Table 2.6. OWASP Benchmark descriptions

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
OWASP Benchmark	Java	2,740	File	2,740

2.4.6 STONESOUP

The Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) dataset was developed by the Intelligence Advanced Research Projects Activity (IARPA), and contains 7,770 known, synthetic vulnerabilities injected into 16 real-world C and Java programs. Each vulnerability is associated with a known CWE. Table 2.7 summarizes the STONESOUP dataset.

Table 2.7. STONESOUP descriptions

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
STONESOUP	C & Java	16	File	7,770

2.4.7 FLVD

The Function-level Vulnerability Dataset (FLVD) contains 15K non-vulnerable and 118 vulnerable real-world C-functions [137]. Vulnerabilities in this dataset were gathered using published CVE data and labeled using diffing. Each function is saved as an individual .c file, consequently, the test cases are not able to be compiled. Table 2.8 summarizes the Function-level Vulnerability Dataset.

Table 2.8. FLVD dataset

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
FLVD	C	15,426	Function	118

2.4.8 VDISC

The Draper VDISC dataset contains the source code for 1.27M functions mined from open source software [45]. Static analysis was used to confirm the location of each vulnerability. The functions are separated and labeled as vulnerable or not vulnerable. The test cases are not able to be compiled. Table 2.9 summarizes the VDISC Dataset.

Table 2.9. VDISC dataset

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
VDISC	C	1.27M	Function	Unknown

2.4.9 LinuxFlaw

The LinuxFlaw dataset [143] contains 368 real-world memory error Linux vulnerabilities. The complete source code for each vulnerability is provided, so each test case is able to be compiled. Table 2.10 summarizes the LinuxFlaw Dataset.

Table 2.10. LinuxFlaw dataset

Dataset	Language	Total Samples	Label Granularity	Known Vulnerabilities
LinuxFlaw	C	368	File	368

2.4.10 Discussion

In this Section we reviewed eight open source datasets that can be used to assess vulnerability analysis tools. Each dataset contains at least one synthetic or real-world vulnerability. In addition to these datasets, Ferenc et al. provide metrics and labels for 12,125 non-vulnerable

and 1,496 vulnerable JavaScript functions [144]. Unlike the previous datasets, [144] does not contain any actual code, however, a link to the full repository associated with each vulnerability is provided. We made no attempt to validate these links were active.

Table 2.11 provides a high level summary of the datasets we reviewed.

Table 2.11. Open source vulnerability datasets.

Open source vulnerability datasets. Code types: real-world (RW), Synthetic (S), and Hybrid Synthetic (HS). Compilable: Yes (Y), No (N)

Dataset	Language	Type	Label Granularity	Known Vulnerabilities	Compilable
CB-Multios	C	HS	File	888	Y
Juliet C#	C#	S	Function	28,942	Y
Juliet C/C++	C/C++	S	Function	64,099	Y
Juliet Java	Java	S	Function	28,881	Y
LAVA M	C	HS	File	2,265	Y
OWASP Benchmark	Java	S	File	2,740	Y
STONESOUP	C & Java	HS	File	7,770	Y
FLVD	C	RW	Function	118	N
VDISC	C	RW	Function	n/a	N
LinuxFlaw	C	RW	File	368	Y
ferenc	JavaScript	RW	Function	1,496	N

Collectively, these datasets contain over 150K known vulnerabilities and bugs. While the vulnerabilities in some datasets are labeled at the file level, others are labeled at the function level. Providing labels at a finer granularity, e.g., functions or even lines of code, allows for a more insightful and quantitative assessment of the performance of different vulnerability identification techniques [137]. Additionally, these datasets provide a collection of pure synthetic, hybrid-synthetic, and real-world code. Synthetic vulnerabilities are sometimes criticized for being overly simplistic and not representative of vulnerabilities in the real-world, while some of the real-world repositories are limited because they provide source code that cannot be compiled. Many of these datasets have been shown to provide disproportionate number of vulnerability types compared to vulnerability instances that have occurred in the wild in the past decade [22].

Perhaps as a result of these limitations, the community has yet to accept a single dataset or repository as a benchmark for VATTs. Numerous researchers have stressed that we are in need of a standard benchmark to quantitatively assess the efficacy of VATTs [22], [25], [27],

[28]. However, a framework for such a benchmark has yet to be defined.

2.5 Feature Extraction

In Sections 2.3.4 and 2.3.5 we reviewed machine learning, deep learning, and data mining techniques used to support vulnerability analysis. In most of the techniques reviewed an essential step in the process is the extraction of one or more features from a sample. As specified in Section 2.3.4, a **feature** is a measurable property or characteristic of a program. We use this term interchangeably with **attribute**. Thus, we refer to tools (techniques, frameworks, etc.) that can be used to extract one or more features from a program as **feature extraction tools**. The proposed work will be open sourced, so we specifically examine open source feature extraction tools. Additionally, the proposed work extracts features from compiled code, so we emphasize open source feature extraction tools that extract features from binary files. Finally we focus on tools that are used to extract one or more features in support of vulnerability analysis.

We partition the reviewed feature extraction tools into two categories:

1. those that extract one feature (i.e., single feature extraction tools)
2. those that extract more than one feature (i.e., multiple features extraction tools)

To clarify, a single feature extraction tool can be used to extract one feature from a file. This could include a feature such as: strings, external function calls, and instruction sequences. A multiple features extraction tool would extract more than one feature, such as strings and external function calls, or complexity, entropy, and instruction sequences, etc.

2.5.1 Single Feature Extraction Tools

In this Section we review a number of open source tools that can be used to extract a single feature from a compiled code.

Li et al. [31] proposed the use of *code gadgets*, i.e., a number of semantically related lines of source code. The code gadgets are transformed into vectors which are then used to predict vulnerabilities using BLSTM. The authors specifically evaluated vulnerabilities related to stack-based buffer overflows (CWE-121). Ultimately, their design was published as a deep

learning-based vulnerability detection system called, *VulDeePecker*. *VulDeePecker* can be used to extract code gadgets from source code.

Lee et al. [30] proposed *Instruction2vec* which can be used to create a vector representation of input using deep learning. *Instruction2vec* was designed to predict vulnerable functions in binary files using a labeled test suite to train a learning model. The original *Instruction2vec* model was trained using test cases from the Juliet dataset [142] that are related to stack-based buffer overflow (CWE-121) vulnerabilities. *Instruction2vec* is used to identify similarities between individual instructions.

The *Maximal Divergence Sequential Auto-Encoder* was proposed by [44], and leverages the deep learning techniques, Variational Auto-Encoders (VAE) and one-hot encoding to identify vulnerabilities in binary code. The authors represent each binary as a sequence of machine instructions, then apply VAE to encourage the samples to be maximally divergent. Similar to [30], *Maximal Divergence Sequential Auto-Encoder* is based on the vector representation for individual instructions.

Asm2Vec is based on the *Doc2Vec* model [145], and uses control-flow information to provide an assembly code representation learning model [146]. Users can provide pre-parsed basic blocks of assembly code or raw assembly code to *Asm2Vec* for processing. In this regard, *Asm2Vec* does not explicitly extract features from compiled binaries, however, *Asm2Vec* can parse raw assembly code into basic blocks.

2.5.2 Multiple Features Extraction Tools

In addition to single feature extraction tools, we also identified one tool that can be used to extract multiple features from compiled code, *VDiscover*.

VDiscover applies static and dynamic information to extract features from assembly code. Specifically, the structure of the code is approximated using static analysis to extract calls made to the standard C library. Dynamic analysis is then used to extract the execution trace for the program including the arguments for the function calls and the final state of the process if applicable [40]. Finally, a representation model (i.e., *Word2Vec*) is used to transform the extracted features into a feature vector[2].

[2]See Chapter 4 for a more detailed description of *Word2Vec*

2.5.3 Discussion

In this Section we reviewed open source tools that can be used to extract one or more features from binary files in support of vulnerability analysis. The majority of tools including [30], [31], [44], [146] extract a single feature from binary samples. *VDiscover* can be used to extract multiple features, however, *VDiscover* is only compatible with C source code. Many of the reviewed feature extraction tools only look at a single vulnerability type. We observe that all of these tools were made public within the past five years, indicating that this is a relevant and growing body of research. The work presented in Chapter 4 of this dissertation mitigates many of the limitations presented by current feature extraction tools, including the extraction of numerous features from binary files, and the compatibility with an array of vulnerability types.

2.6 Summary

In this chapter we set the stage for the remainder of our research. We provided background relevant to each the development of a framework for BVATT, and a means to extract common features from binary samples in support of machine learning and data mining based vulnerability research.

Looking forward, in Chapter 3 we'll provide a framework, that upon implementation allows users to systematically compare the performance of one VATT versus another in a manner that is repeatable, reproducible, fair, verifiable, and relevant. Then, in Chapter 4 we'll provide the `BiSECT` tool. `BiSECT` allows users to succinctly extract, clean, and transform common features from binary files at scale in support of machine learning and data mining based vulnerability research.

CHAPTER 3:
A Framework for the Benchmark for Vulnerability Analysis Tools and Techniques (BVATT)

[...] when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you can not measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.

Lord Kelvin, *Electrical Units of Measurement.*

3.1 Introduction

In 1883, Lord Kelvin called for a "numerical reckoning" in popular sciences [147]. Kelvin urged that without taking steps towards quantitative measurement we cannot objectively compare the electromagnetic resistance of two coils, or the hardness of a diamond to that of a ruby. The concept of quantitative measurement was hardly novel in the 1800s. Nearly two centuries prior, in 1687, Sir Isaac Newton published, *Philosophiae Naturalis Principia Mathematica*, a groundbreaking work that translated qualitative statements about the physical world into a mathematically based science [148]. These principles provide the fundamental building blocks that allow us to compare the velocity of two objects, and design satellites that successfully orbit celestial bodies.

In computer science, benchmarking suites allow us to objectively and repeatedly compare one multiprocessor design to another, and standard datasets are used to quantitatively assess the relative performance of different algorithms. We can state with confidence that a grammar is ambiguous, and consistently apply the pumping lemma to show that a language is not regular. These building blocks allow us to objectively compare programming languages and architect complex computing systems. In security, databases like CVE and SARD use a consistent syntax and reporting structure for cataloging vulnerabilities [24], [56]. Uniformity in these areas provides a basis for experimentation based on quantitative measurement. Yet,

the facet of security research that focuses on the comparison of VATTs seems to lack the quantitative element exhibited by its peers.

The security community relies on VATTs to support software vulnerability analysis processes. In Chapter 2 we reviewed several vulnerability analysis techniques that we organized into four logical categories: static analysis, dynamic analysis, machine learning and data mining, and finally, techniques using deep learning and neural networks. We discussed strengths and weaknesses of each technique, and provided an overview of methods to compare the relative efficacy of VATTs.

While there are a few useful resources [24]–[26], a standard benchmark to assess the relative efficacy of VATTs has yet to be accepted by the community. Consequently, we have no comparative metrics of the types of vulnerabilities each tool and technique can or cannot find, how quickly they can do so, or the number of false positives they report. When a consumer wants to employ a VATT, they have no standard method of comparing the alternatives to determine which is the optimal choice for their use case. When a developer wants to modify an existing VATT, or enter the market with a new one, no standard benchmark is available to compare their VATT against existing ones. In the absence of such a benchmark, consumers and developers compare vulnerability analysis tools and techniques in a disjointed fashion. This can result in misinformation, subjectivity, redundancy, and inconsistency. One common comparison approach is to assess the relative performance of different tools and techniques using a collection of test cases containing known vulnerabilities. These test cases are often organized into public or proprietary datasets.

Numerous researchers have stressed that we are in need of a standard benchmark to quantitatively assess the efficacy of VATTs [22], [25], [27], [28]. In this Chapter we establish a framework for such a benchmark, which we dub, BVATT. We first summarize a few open source datasets, and their corresponding limitations (also summarized in Chapter 2). This is followed by a discussion of lessons learned from well established benchmarks across multiple domains— from these lessons we surmise that a good benchmark should be repeatable, reproducible, fair, verifiable, and relevant. These five characteristics become the basic building blocks upon which we develop the framework for BVATT. Finally, we extensively investigate what it would require to create a benchmark that is representative of reality (i.e., relevant). To accomplish this, we examine methods to leverage test cases

from existing datasets, and create a benchmark that is representative of weakness types and vulnerability instances as they occur in the wild. Ultimately, this work led us to create the binary feature extraction tool, `BiSECT`. `BiSECT` is briefly introduced at the conclusion of this chapter, and discussed extensively in Chapter 4.

To summarize, in Chapter 3 we make the following contributions:

1. Synthesize 839 CWEs with over 75,000 CVEs to determine the relative proportions of vulnerability instances and weakness types in the wild
2. Analyze four popular software vulnerability datasets, and show that none accurately represents vulnerability instances and weakness types as they occur in the wild
3. Provide a framework for a benchmark for vulnerability analysis tools and techniques (BVATT)

3.1.1 Motivation and Chapter Outline

Hundreds of thousands of publicly available test cases containing known software flaws and vulnerabilities have been aggregated into datasets, each with its own structure, supported languages, and reporting method. Databases such as the SARD attempt to inject order by providing a consolidated repository of vulnerability datasets and test cases [24]. Unfortunately, even the SARD, which contains 40 datasets and over 170K test cases, is not exhaustive— it excludes well known datasets such as the CGC Corpus [26], LAVA-M [25], and OWASP Benchmark Project [29].

Table 3.1 provides a high level overview of the source code language and number of known vulnerabilities in seven open source datasets, the granularity of the datasets vulnerability labels is also included in the table. Additional information about each dataset is provided in Chapter 2. Collectively, these datasets contain over 135K synthetic-known vulnerabilities and bugs.

Limitations of Current Datasets

These datasets suffer from several limitations. We demonstrate that many contain an unrealistic representation of weakness types, i.e., CWE entries, when compared to known vulnerability instances in the wild, i.e., accepted CVE entries [22]. Consequently, even if a

Table 3.1. Combined descriptions of seven open source vulnerability datasets

Dataset	Language	Total Programs	Label Granularity	Known Vulnerabilities
CB-Multios	C	243	File	888
Juliet C#	C#	28,942	Function	28,942
Juliet C/C++	C/C++	64,099	Function	64,099
Juliet Java	Java	28,881	Function	28,881
LAVA M	C	4	File	2,265
OWASP Benchmark	Java	2,740	File	2,740
STONESOUP	C & Java	16	File	7,770

VATT was assessed using all 170K of the SARD test cases, the results would still not reflect reality. This limitation is discussed extensively in Section 3.3.5.

While the vulnerabilities in some datasets are labeled at the file level, others, specifically those included in the Juliet corpus are labeled at the function level. Labeling vulnerabilities with a finer granularity, e.g., function versus file level, allows for a more insightful and quantitative assessment of the performance of different VATTs [27], [28], [30], [31]

Additionally, while some of the datasets include real-world code (e.g., LAVA M, and STONESOUP), all of the vulnerabilities within each of these datasets are entirely synthetic. Synthetic vulnerabilities are sometimes criticized for being overly simplistic and not representative of vulnerabilities in the real-world [27], [28].

In addition to open source datasets, some researchers have created custom datasets to quantify the performance of VATTs. There are several limitations to using custom-developed datasets [25], [27], [28]:

1. Custom datasets are often proprietary, and difficult for other researchers to reproduce and verify
2. It is difficult to discern whether test cases were cherry picked to highlight the strengths (and minimize the weaknesses) of a particular tool or technique
3. Vulnerabilities may be labeled at a different granularity, making it difficult to combine datasets

The limitations of existing datasets are perhaps the reason that none has been accepted as the

standard benchmark for VATTs. Numerous researchers have stressed that we are in need of a standard benchmark to quantitatively assess the efficacy of VATTs, but even a framework for such a benchmark has yet to be established.

Based on the limitations of existing datasets, and the absence of a community-accepted benchmark we determined that the community is in need of an overarching framework to guide the implementation of BVATT. The framework for BVATT should include both the core characteristics of the benchmark, and metrics that can be used to quantitatively compare VATTs.

In Section 3.1.2 we explore lessons learned from computing-related benchmarks, which are used as the foundation for BVATT.

3.1.2 Benchmarks

In the 1960s there was a significant increase in the number of computer vendors and variety of system configurations. *Benchmark problems* became a popular method to compare the speed with which computers accomplished basic data processing functions [149]. Benchmark problems are not strictly bound to assessing computing speed though. In the past decade numerous benchmarks have been published to quantify the efficacy of machine learning and deep learning models. In 2009, the publication of the ImageNet benchmark led to noteworthy developments in computer vision, advances in deep convolutional networks, and ultimately stimulated collaborations among the community [150]. In 2018, MoleculeNet was published and has had similar affects in molecular machine learning research [151].

Benchmarks have a long history in computing, and have had multiple effects on the community across various sub-domains. Commonly, benchmark problems are used to quantify the performance differences between tools or techniques [149], but they can also encourage innovation, competition, and collaboration [150], [151]. Our hope is that by establishing the framework for BVATT, we can have a similar impact on the vulnerability analysis community. To this end, we first explore desirable characteristics for a benchmark proposed by a number of researchers [27], [28], [152]–[155]. From these we detail the following five fundamental characteristics for BVATT:

- **Repeatable** The same results should be consistently achieved when the benchmark is run with an identical tool under identical conditions
- **Reproducible** The benchmark results should be independently achievable
- **Fair** The benchmark should not be partial to any particular tool or technique
- **Verifiable** There should be confidence that benchmark results are accurate
- **Relevant** The benchmark problems should be representative of reality

In Section 3.3 we detail each of these characteristics for BVATT. In Section 3.2 we examine another key component for BVATT, scoring.

3.2 BVATT Scoring

A primary goal of most VATTs is to correctly identify code as vulnerable or not-vulnerable. In this way, these VATTs can be considered binary classifiers [156]. BVATT can leverage common metrics for binary classification to compare the efficacy of different VATTs that share this goal. The fundamental components for these metrics are included in a standard LxL *confusion matrix*, which shows the predicted and actual classifications, where L is the number of labels [157]. In a binary classification scenario all possible outcomes are represented in a 2x2 matrix, and include: True Positive (TP), False Positive FP, True Negative (TN), and False Negatives (FN), where:

- True Positive (TP): Actual positive, p′, predicted as positive, p.
- False Positive (FP): Actual negative, n′, predicted as p.
- True Negative (TN): n′ predicted as negative, n.
- False Negative (FN): p′ predicted as n

Table 3.2 shows a confusion matrix for binary classification scenarios.

[156] examined 14 metrics that can be used to compare the efficacy of vulnerability detection tools in different binary classification scenarios. The authors analyze and score each metric based on the following criteria: how *understandable* the metric and its formula are to non-statisticians, how *meaningful* the metric is to general users, how *scalable* and cheap the metric is to gather, whether both TP and FP are measured, how much emphasis is placed

Table 3.2. Confusion Matrix for Binary Classification

		Predicted Value	
		p	**n**
Actual Value	**p′**	True Positive	False Negative
	n′	False Positive	True Negative

on TPs, and how compatible the metric is with different tools and small data sets[3]. They conclude that the optimal metrics to compare vulnerability detection tools include: recall, precision, false detection rate, f-measure, F_x-score, and false-positive rate [156]. The formula and description corresponding to each of these metrics is provided in Table 3.3.

Antunes and Vieira demonstrated that different metrics may be more or less suited to certain vulnerability identification scenarios [156]. For example, in a business-critical application scenario where the goal is to identify the VATT that detects the highest number of vulnerabilities, *recall* is the optimal metric. However, in a best-effort scenario where the goal is to identify the VATT that detects the highest number of vulnerabilities while reporting the lowest number of false positives, the *f-measure* is the optimal metric. To support an array of use cases, we propose that BVATT report each of these metrics, in addition to the values reported in the confusion matrix, for each VATT run against the benchmark.

BVATT should also report the metrics by *type* of vulnerability. In order to accomplish this, each vulnerability in BVATT must be labeled with an associated vulnerability type such as the CWE-ID. Reporting the metrics at this granularity will enable users of the benchmark to determine the efficacy of VATTs when tasked with identifying vulnerabilities of a particular type. BVATT should provide each of these metrics as a final scorecard in an easily parsed

[3]The RCCR (repeatable, consistent, comparable ratio) for each metric is also reported. However, each metric received an identical score of 2 in this category, thus the influence of the RCCR on the final score is negated. We exclude the RCCR from our review

Table 3.3. Metrics proposed for BVATT. Adapted from [156].

Metric	Formula	Description
Recall	$\frac{TP}{P} = \frac{TP}{TP+FN}$	Proportion of positive cases correctly classified as positive (aka *true positive rate*)
Precision	$\frac{TP}{TP+FP}$	Proportion of classified positive cases that are correctly classified (aka *true positive accuracy*)
f-Measure	$2 * \frac{prec*recall}{prec+recall} = \frac{TP}{TP+\frac{1}{2}(FP+FN)}$	Represents the harmonic mean of precision and recall. Equivalent to the F1 Score
False Detection Rate	$\frac{FP}{FP+TP}$	Ratio of reported positives incorrectly classified
F_x Score	$(1+x^2) * \frac{\text{prec} * \text{recall}}{(x^2 * \text{prec}) + \text{recall}}$	Weighted average of precision and recall (0=worst, 1=best)
False Positive Rate	$\frac{FP}{N} = \frac{FP}{FP+TN}$	Ratio of negatives incorrectly classified as positives (aka *fall-out*)

format, such as a JavaScript Object Notation (JSON) or comma-separated values (CSV) file.

3.2.1 Scoring Example Implementation

An example scorecard is provided in Figure 3.1. In this hypothetical example, a VATT was tested against 100 vulnerable (denoted by 'P') and 100 not-vulnerable (denoted by 'N') test cases for 5 CWEs, for a total of 1K test cases. The scores for this VATT were high for some CWEs (e.g., 664 and 691), but low for others (e.g., 682).

	CWE	P	N	TP	FP	TN	FN	Recall	Precision	F-M	FDR	F0.5 Score	F1.5 Score	FPR
0	284	100	100	45	12	86	57	0.450	0.789474	0.566038	0.210526	2.8125	2.112500	0.120
1	435	100	100	60	88	45	7	0.600	0.405405	0.558140	0.594595	3.7500	2.816667	0.880
2	664	100	100	90	2	96	12	0.900	0.978261	0.927835	0.021739	5.6250	4.225000	0.020
3	682	100	100	12	58	8	122	0.120	0.171429	0.117647	0.828571	0.7500	0.563333	0.580
4	691	100	100	94	4	86	16	0.940	0.959184	0.903846	0.040816	5.8750	4.412778	0.040
5	Total	500	500	301	164	321	214	0.602	0.647312	0.614286	0.352688	3.7625	2.826056	0.328

Figure 3.1. Example scorecard for a VATT run against BVATT

This scorecard was generated using a CSV and Python code. To support future BVATT maintainers, our score card generation source code has been made publicly available via GitHub[4]. As VATTs continue to evolve, so too may the metrics to quantify the efficacy of those VATTs in different scenarios. For this reason, in addition to the 6 metrics proposed for BVATT (summarized in Table 3.3), the GitHub repository also contains the code to calculate all 14 metrics from [156] to compare the efficacy of VATTs.

3.3 BVATT Characteristics

In this section we examine what is required to achieve the characteristics outlined in Section 3.1.2, and establish a BVATT that is repeatable, reproducible, fair, verifiable, and relevant.

3.3.1 Repeatable

The same results should be consistently achieved when repeating an identical experiment. For example, when evaluating the same version of a VATT against the exact same set of benchmark problems in the exact same environment the same metrics (i.e., results) should be produced. BVATT should be *reliable*, and metrics produced using BVATT should be repeatable. Ensuring reliability not only gives confidence that results produced by BVATT are stable, but it also provides a baseline for future experimentation and the evaluation of new ideas using BVATT [158]. To this end, we propose a more stringent evaluation criteria for repeatability that includes statistical measurements.

Test Retest Reliability

The extent to which measurements can be repeated is often referred to as *test-retest reliability* (also called *internal consistency*) [159]. Given the same VATT, the metrics for tests T_1 and T_2 are said to be correlated. This correlation has many different names including the *test-retest-reliability coefficient*, *Intraclass Correlation Coefficient (ICC)*, *coefficient of stability*, and sometimes simply the *reliability score* [159], [160]. The closer the scores are between T_1 and T_2, the more reliable the test measure is, and the higher the reliability score will be. If the metrics from T_1 and T_2 are perfectly correlated, then there is no measurement error, and the reliability will be equal to 1. Conversely, a reliability score of 0 indicates that

[4]https://github.com/Kayla0x41/BVATT/tree/master/scorecard

there is high measurement error. In this context, *reliability* accounts for the following [159], [160]:

- consistency of a test or measurement,
- the distinguishability of individual measurements,
- and the signal-to-noise ratio in the data

We propose the use of standard statistical methods to ensure that BVATT achieves test-retest reliability. In the following section we explore the use of Analysis of Variance (ANOVA) measurements to quantify reliability between test runs.

Analysis of Variance (ANOVA)

Reliability can be quantified using *variance* in the data. Ronald Fisher first introduced the term **variance** in 1918, which he described as the square of the standard deviation of the data [161]. In 1925, Fisher expanded his original concept to include the *analysis of variance* or ANOVA [162]. ANOVA is used to identify statistical differences among the means, and ultimately to identify relationships and correlations between data. There are several types of ANOVA, but one type, the *repeated-measures ANOVA* (sometimes called the *within-subjects ANOVA* or *ANOVA for correlated samples*) is uniquely suited to identify statistical differences for related, non-independent groups. In this way, the repeated-measures ANOVA is the optimal ANOVA for our use case, as we aim to quantify reliability between test runs for the same VATT.

The repeated-measures ANOVA is statistically equivalent to a **paired T-test** with repeated measures– the only caveat being that in the paired T-test a maximum of two measurements may be compared, whereas there is no limit in repeated-measures ANOVA. A repeated-measures, paired T-test will suffice for our use case as it will allow us to guarantee the reliability of BVATT metrics between two test runs. There are two assumptions in a paired T-test: first, each test run is assumed to have an equal sample size, and second, the variance is assumed to be equal between test runs.

To summarize, both the paired T-test and repeated-measures ANOVA can be used when the same subject (i.e., VATT) is measured more than once for each experimental condition (i.e., test case). Using the repeated-measures paired T-test we can quickly verify the reliability of

experiments run using BVATT.

The **T-test** for test runs T_1 and T_2 is calculated using the following formula:

$$t = \frac{\overline{T}_1 - \overline{T}_2}{s_p \sqrt{\frac{2}{n}}} \tag{3.1}$$

Where,

$$s_p = \sqrt{\frac{s_{T_1}^2 + s_{T_2}^2}{2}} \tag{3.2}$$

Where n is equal to the number of test cases.

We provide an example implementation in Section 3.3.1 to illustrate the power and convenience of using this method to ensure metrics produced by BVATT are reliable.

Paired T-Test Example Implementation

In the following example we consider a VATT, $VATT_1$, and two test runs, Test 1 (T_1) and Test 2 (T_2). In T_1, $VATT_1$ is run against 1,000 test cases spanning 5 CWEs. Of these test cases, 500 contain 1 known vulnerability, and the remaining 500 contain the patched (i.e., non-vulnerable) version of the vulnerable file. The metrics from this experiment are summarized in Column "Test1" in Figure 3.2. A value of "True" indicates that either a TP or TN was recorded, while a value of "False" indicates that the VATT reported either a FP or FN. Then, we repeat the exact experiment, in the same environment, using $VATT_1$ and the same test cases. The metrics from this experiment are summarized in Column "Test2" in Figure 3.2. The results from T_1 and T_2 are identical, the paired T-test from the paired T-test are reported as "nan", as seen in Figure 3.3.

Now, let's say we want to improve the performance of $VATT_1$, so we manipulate various independent variables. For example, we could increase the duration of the test allowing $VATT_1$ more time to correctly identify vulnerabilities. After making this change we rerun the experiment, and record the results from this test in Table T_3.

Using the paired T-test, we can easily assess the statistical impact of the changes made between the tests, T_1 and T_2, and the test after the modifications, T_3. The paired T-test

hypothesizes that the populations have identical variances by default, and produces a *p-value*. If the *p-value* is larger than a predetermined threshold (typically, 0.05 or 0.1) then the hypothesis cannot be rejected, and the average score must be identical. However, if the *p-value* is smaller than the threshold, then the hypothesis can be rejected, and it can be stated that our modifications indeed had an impact on the results. To calculate the repeated measures paired T-test we use the open source Python package, SciPy [163]. Figure 3.4 shows the results from the paired T-test on T_1 and T_3. In this example, $p = 0.07$, thus we reject the null hypothesis, and conclude that the modifications to the environment had a statistical impact on the benchmark results.

	Test Case ID	Test1	Test2
0	t0001	True	True
1	t0002	True	True
2	t0003	False	False
3	t0004	True	True
4	t0005	False	False
...	...	...	...
495	t0496	True	True
496	t0497	True	True
497	t0498	False	False
498	t0499	True	True
499	t0500	False	False

500 rows × 3 columns

Figure 3.2. Sample results from Test1 (T_1) and Test2 (T_2), where $VATT_1$ was run against an identical set of 1K test cases

The paired T-test provides a simple and effective means to ensure that results generated by BVATT are reliable. To support future BVATT maintainers, our code Jupyter notebook implementing the paired T-test has been made publicly available via GitHub[5].

[5]https://github.com/Kayla0x41/BVATT/tree/master/scorecard

```
stats.ttest_rel(test_runs['Test1'],test_runs['Test2'])

Ttest_relResult(statistic=nan, pvalue=nan)
```

Figure 3.3. Paired T-test for identical repeated measures (T_1 and T_2) example calculation using SciPy

```
stats.ttest_rel(test_runs['Test1'],test_runs['Test3'])

Ttest_relResult(statistic=1.7891397773495923, pvalue=0.07419869540784921)
```

Figure 3.4. Paired T-test for different repeated measures (T_1 and T_3) example calculation using SciPy

3.3.2 Reproducible

Reproducibility refers to the independent confirmation of a scientific hypothesis through reproduction by an independent party [164]. In the context of BVATT, reproducibility means that independent parties should be able to reproduce the results achieved by the benchmark. Reproducibility enables independent users of BVATT to replicate results, verify how results may change using different parameters, and re-use or extend experiments [165]. Reproducibility is made up of two primary components [165]:

1. a full description of the experiment including the data used and experiment specification (e.g., steps, duration, underlying code needed, and environment)
2. a complete package of the components, so that it may be executed in various operating environments

If included in BVATT, the first component would provide a step-by-step guide to enables users to replicate the experiment using the second component. To achieve the second component BVATT should be implemented and shared as a Docker container, or other similar technology. A Docker container is a virtual *container* that includes all of the requisite code, runtime information, dependencies, system tools, libraries, settings, etc. needed for a software to run [166]. Using a Docker container will ensure that users are able to download and use BVATT in numerous operating environments.

3.3.3 Verifiable

Verifiability provides confidence that benchmark results are accurate. In order to achieve verifiability the method to identify each vulnerability in the benchmark should be documented. Verifiability in synthetic vulnerability datasets is often accomplished via an author provided description of the vulnerability [29], [142]. Researchers have proposed a number of methods to verify vulnerabilities in real-world code [137], [167]. In the next section we provide an example method to identify vulnerabilities in original and patched executables using binary diffing.

Vulnerability labeling using diffs

One technique to identify vulnerabilities in real-world repositories is to diff the original and patched versions of source code files [137], [167]. During the diff the original function corresponding to any function modified during a patch is deemed vulnerable (the updated function is deemed not vulnerable). We extend this approach and develop a technique to identify vulnerabilities in original and patched executables using binary diffing.

Specifically, we wrote a Python script to leverage the Ghidra, BinExport, and BinDiff APIs, and identify all functions that were modified during a patch [168], [169]. By using the APIs instead of the GUI interface, this method provides the added benefit of batch analysis. Like [137], [167] we assume that if a function was modified during the patch process then the original function was vulnerable. Furthermore, if a function was not involved in any patches, then it is labeled as 'unchanged'. We use *unchanged* instead of 'not vulnerable' for these functions, as we cannot assume that they do not contain an undiscovered vulnerability. The high level diffing process is depicted in Figure 3.5.

To demonstrate the utility of our approach, we provide an example using the CB-Multios corpus, in which we identify and label 888 vulnerable functions, and 136,842 non-vulnerable and unchanged functions [141]. In this corpus, the author of each Challenge Binary (CB) provided a detailed description of the known vulnerabilities included in the sample, including an associated CWE classification [140], [141]. Using GCC on an Ubuntu virtual machine, we compiled 285 known vulnerable test cases or samples from the TrailofBits CB-Multios repository, and 285 patched versions of the samples. Known vulnerabilities in the CB-Multios corpus are labeled at the file level, however, users can manually query the source code for the

presence of `ifdef PATCHED` which will reveal functions that contain known vulnerabilities.

For example, the author of the CB called, *BitBlaster*, indicates that the test case contains vulnerabilities related to CWE-284, *Access of Uninitialized Pointer* and CWE-476, *Null Pointer Dereference*. To increase the granularity of the vulnerability labels and reduce the manual effort required to label the functions we leveraged the differences between the patched and original samples. For reference, it took less than 5 minutes to identify and label all 137K functions as vulnerable or not vulnerable using the proposed technique.

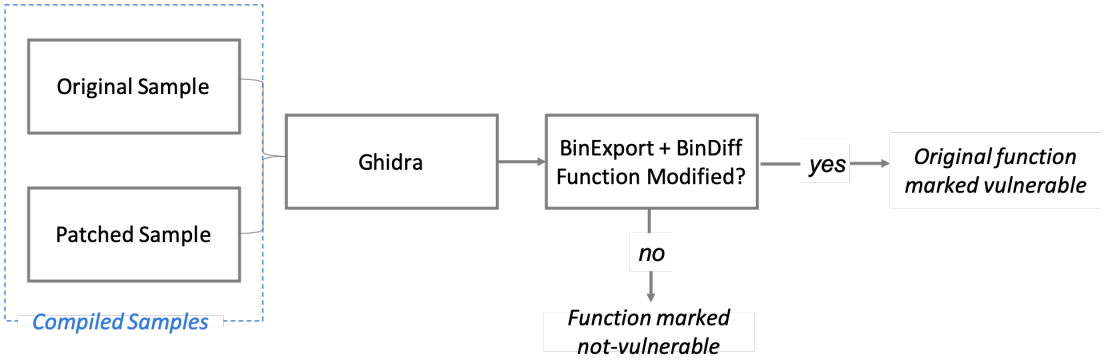


Figure 3.5. High level process to label functions in compiled code as vulnerable or not vulnerable

While the BinExport and BinDiff plugins and associated APIs are well documented for IDA Pro [169], there is limited information available on using the APIs with Ghidra. We have made our code publicly available on GitHub, and provide a Jupyter Notebook that details our full workflow [170].

3.3.4 Fair

In order for BVATT to be fair, that is, not partial to any particular tool or technique, the test cases should be selected independently of any specific VATT. We discuss test case selection methods extensively in Section 3.3.5.

3.3.5 BVATT Relevance

The results from a benchmark test can only be as good as the benchmarks themselves. If the benchmarks are not representative, the results could prove

more harmful than no benchmark at all. [149]

Determining the relevance of problems within a benchmark can involve a number of elements. Kistowski states that from a design perspective relevance involves two dimensions: the breadth of the benchmark's applicability, and the degree to which benchmark problems are relevant in each area of interest [152]. Karl Huppler identifies seven characteristics that determine whether a benchmark problem is relevant. Each problem in a benchmark [154]:

1. must provide a meaningful and understandable metric
2. should stress software features in a way that is similar to customer applications
3. should exercise hardware systems in a way that is similar to customer applications
4. must have longevity
5. must have broad applicability
6. should not misrepresent itself
7. has a target audience that wants the information

John Henning states that benchmark suites should be derived from real-world applications so that designers and consumers can make decisions on the basis of realistic workloads [153]. In order to be relevant benchmark problems should be closely connected to, and representative of reality. In 1965, Joslin discussed the keys to meaningful computer evaluations. He states, "The question of what constitutes good benchmark problems can be answered in one word—representativeness" [171].

In the context of BVATT, we summarize *relevance* as follows: the suite of benchmark problems included in BVATT should be representative of the types of vulnerabilities prevalent in the real-world. In the remainder of this Section, we examine precisely how *reality* and *representativeness* can be determined.

Reality

The industry standard for vulnerabilities in the real-world is provided by the CVE. The CVE is a dictionary of publicly known vulnerability and exposure instances [6]. Each entry in the dictionary describes an instance of a vulnerability, and includes metadata such as a unique identifier (CVE ID), description of the vulnerability, and where applicable, a corresponding

CWE entry[6].

From 1999 to May 2020 the CVE published 160, 544 known vulnerabilities and exposures [6]. To date, the greatest number, 21, 598, of publicly disclosed vulnerabilities and exposures was reported by the CVE in 2018. Over half, 93, 056, of all CVE entries ever recorded, excluding 2020, were published between 2014 to 2019 [6]. Of these, 75, 535 CVEs were accepted[7] by the community— this provides a substantial collection of real vulnerability instances. Figure 3.6 shows the sum of CVEs by year published from 1999 to 2020. The blocks are organized by size, and the size of each block in the figure is proportional to the number of CVEs published in that year. For example, the greatest number of vulnerabilities was published in 2018 (21.60K vulnerabilities).

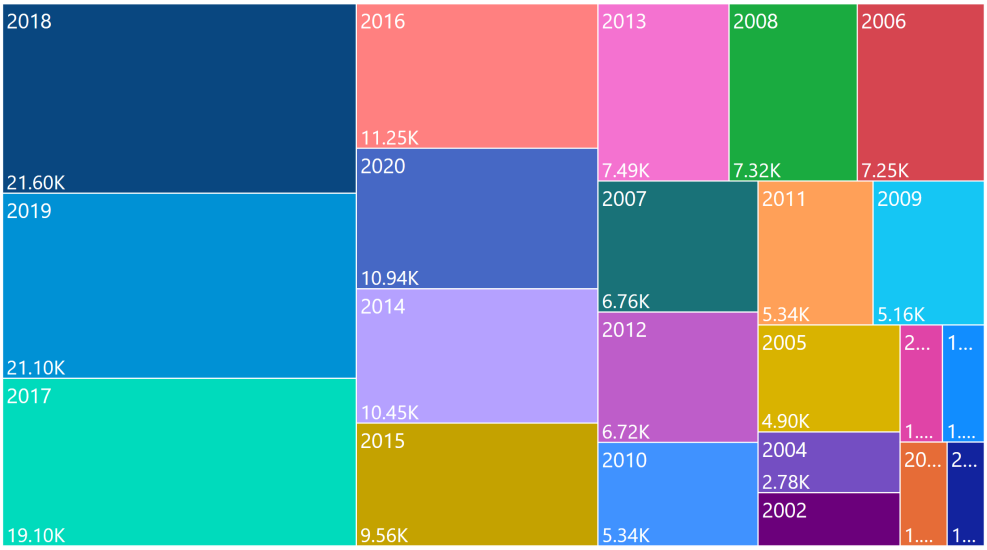


Figure 3.6. Sum of CVE (all statuses) by year since 1999. The size of each block is proportional to the number of CVEs published in that year.

Alone, the CVE provides a dictionary of real and relevant vulnerability instances, however, the CVE does not categorize vulnerabilities by their associated weakness type. To mitigate this deficiency, we use the known correlation between CVE and CWEs to classify each

[6]Many CVE entries published prior to the development of the CWE (2006) do not include a CWE ID

[7]A CVE *record* is an item in the CVE list. Not all CVEs will be 'accepted'. Some may be labeled as 'rejected', however they still show up in the overall CVE listing. A CVE may be rejected if it is discovered to be a duplicate, withdrawn by the original requester, or was incorrectly assigned [6]

CVE by its type. What the CVE provides for vulnerability instances, the CWE provides for weakness *types*. The CWE is a repository of over 1200 hardware and software weaknesses, and provides a common language, identifier, and definition for each weakness type referenced. CWE entries are organized into a number of views to support different objectives. We use view CWE-1000, Research Concepts, which includes a hierarchy of 839 CWE entries. Each CWE in the hierarchy is associated with one of the following abstraction types [172]:

- **Pillar** Weaknesses that are described in the most abstract fashion (10 CWEs).
- **Class** Abstract weakness, typically independent of any specific language or technology (96 CWEs).
- **Base** A more specific type of weakness (441 CWEs).
- **Variant** A weakness that is described at a very low level of detail, typically limited to a specific language or technology (285 CWEs).
- **Composite** A set of weaknesses that must all be present simultaneously in order to produce an exploitable vulnerability (7 CWEs).

Using the abstraction types the weakness hierarchy presented by view CWE-1000 can be organized into ten **rooted trees**. A rooted tree is a tree with a single *root* vertex that is distinguished from all others. Each pillar in the hierarchy is the root node of a rooted tree.

Using BeautifulSoup [173], Pandas [174], and D3 [175] we crawled over 1000 individual pages on the CWE website to create and visualize the tree data structures for each of the ten CWE Pillars. This approach allows us to view the most up-to-date information on CWE relationships and hierarchies. Figure 3.7 shows the overall process to crawl, extract, manipulate, and visualize the CVE and CWE data.

Then, by using CWE-1000 as the root node of a tree we can create a single rooted tree that includes every CWE in the CWE-1000 view. Figure 3.8 depicts the ten CWE pillars and their 839 children as a hierarchical radial dendrogram with root node CWE-1000.

Of the 75,535 community-accepted CVEs published from 2014-2019, 55,128 have an associated CWE. By using the correlations between CVEs and CWEs we classify each of the vulnerability instances (i.e., CVEs) by their weakness type (i.e., CWE). Figure 3.9 shows the sum of known vulnerability instances published from 2014-2019, by type. This visualization shows the relative proportions of CWE in the real-world. The enclosing circles

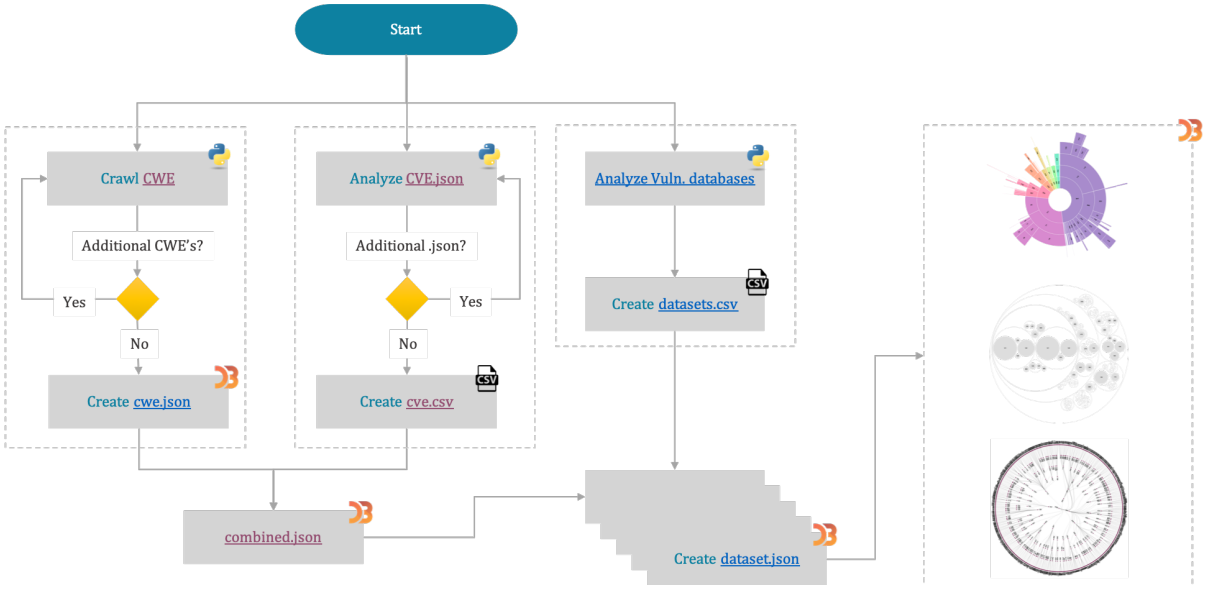


Figure 3.7. The high-level process to crawl, extract, manipulate, and visualize CVE and CWE data

show the cumulative size of each of the ten CWE pillars (i.e., subtrees), while maintaining relationship and hierarchical data. The exterior circle represents root node, CWE-1000.

We repeat this process using four popular vulnerability datasets: Juliet C/C++ (Juliet (C)), Juliet Java (Juliet (J)), the OWASP Benchmark (OWASP Ben.), and the CGC Corpus (CGC). Figure 3.10 shows the sum of vulnerability instances (CVE ID) by type (CWE ID) from 2014-2019 (a), and includes a sum of test cases by type in each dataset (b-e) for comparison. These Sunburst diagrams illustrate the stark contrast between the types of weaknesses in the wild, and those in current vulnerability datasets.

Like the CVE, each test case in the reviewed datasets has a corresponding CWE ID that can be traced to a pillar node. Table 3.4 shows the relative percentages of test cases in each pillar by vulnerability dataset. It shows that none of the vulnerability datasets accurately reflects vulnerabilities as they have occurred in the real-world, i.e., CVEs from 2014-2019. Conversely, we propose a method in Section 3.3.5 to preserve the real proportions of vulnerabilities and weakness types, and ensure that BVATT is representative of reality.

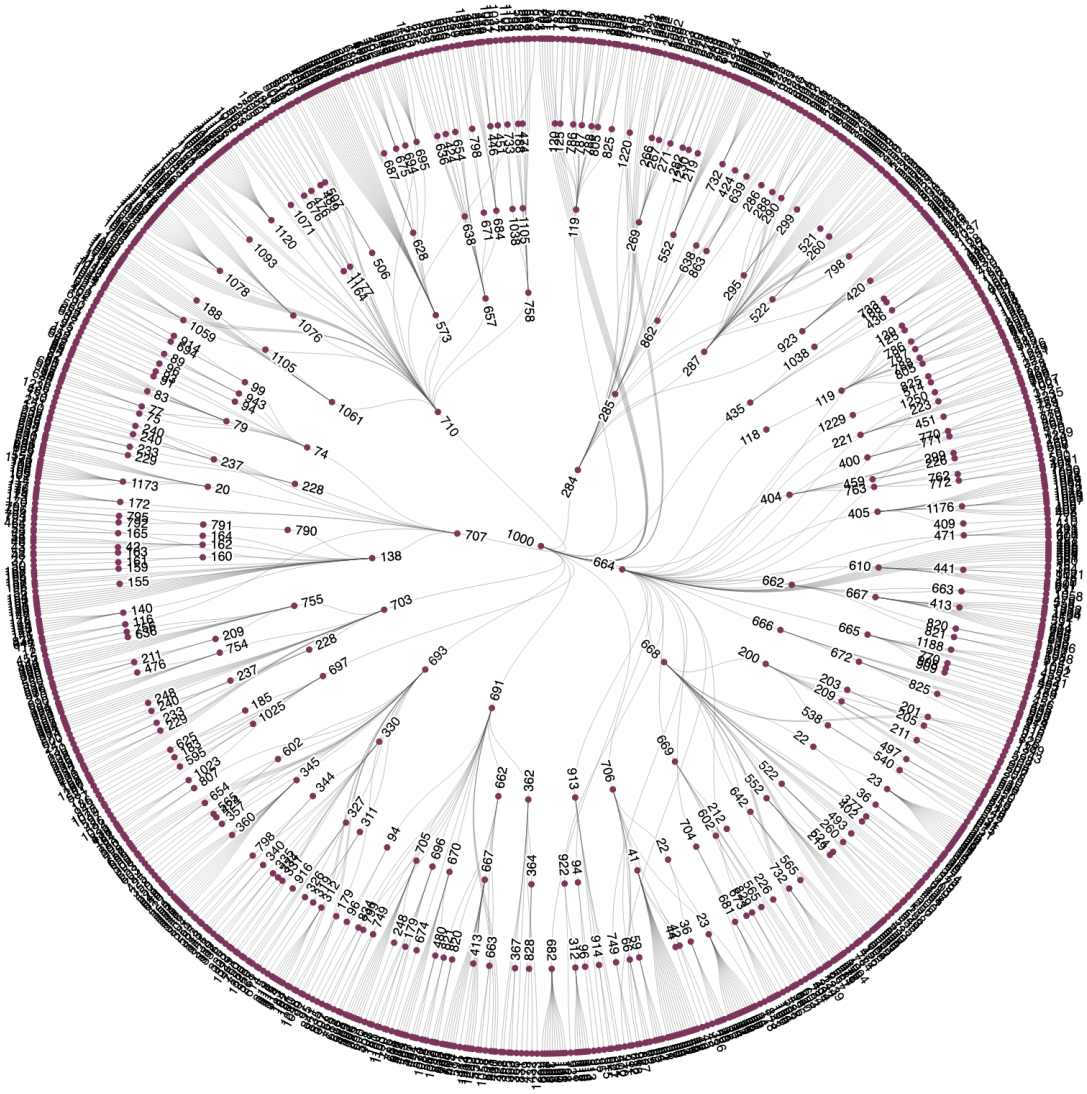


Figure 3.8. CWE view 1000 depicted as a hierarchical radial dendrogram

Representativeness

By leveraging the relationship between CVE and CWEs we can identify a candidate repository of 55, 128 real vulnerabilities that include corresponding type information. From this repository, we could create one benchmark problem for each vulnerability instance, however, to conserve time, limited resources, and availability we identify a subset of vulnerabilities that is representative of the larger set.

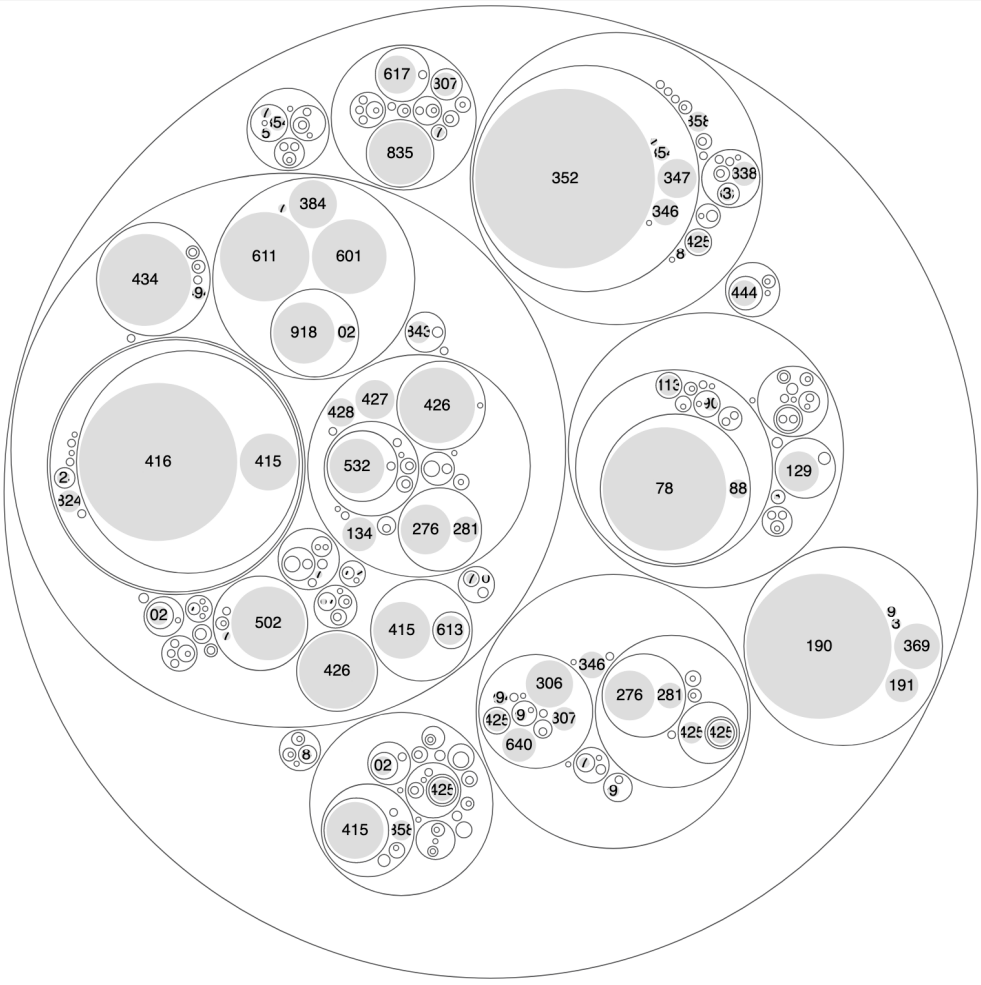
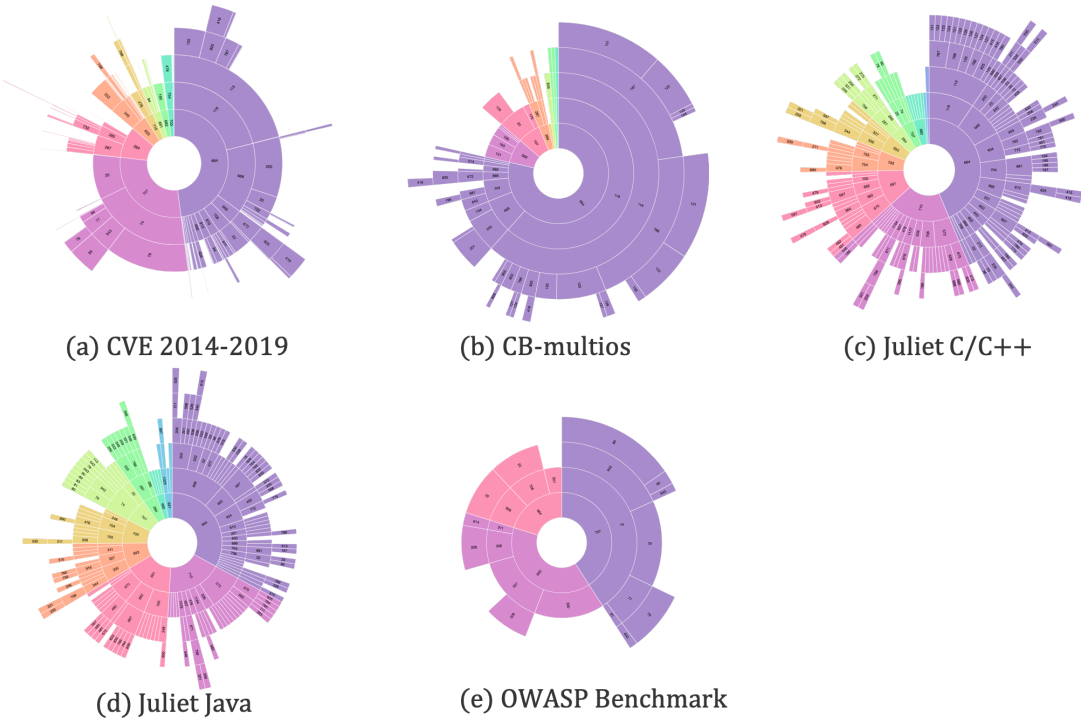


Figure 3.9. Sum of vulnerability instances (CVE) by type (CWE) from 2014-2019

When resources are constrained statisticians take samples of the data to make quantitative inferences about a larger population [176]. A *representative* sample is a type of sample that ensures external validity, or applicability, relative to the overall population of interest [177], [178]. We will use the terms *sample* and *subset* interchangeably. Sampling allows us to learn about the larger population without studying every unit in the population. Our population is defined as the set of known, publicly-accepted vulnerabilities published by the CVE from 2014-2019 that have an associated CWE (as described in the previous Section). We consider two key factors that affect the representativeness of a sample: sample design, and sample



(a) CVE 2014-2019 (b) CB-multios (c) Juliet C/C++

(d) Juliet Java (e) OWASP Benchmark

Figure 3.10. Sunburst diagram comparison of CWE distribution in open source datasets

Table 3.4. Percentage of test cases in each CWE pillar by vulnerability dataset

Pillar	CVE Total	CGC	Juliet (C)	Juliet (J)	OWASP Ben.
CWE-284	10.66%	3.47%	0.95%	0.89%	0.00%
CWE-435	0.07%	0.00%	0.04%	0.00%	0.00%
CWE-664	45.27%	72.25%	68.48%	25.52%	14.38%
CWE-682	2.53%	9.83%	12.66%	34.06%	0.00%
CWE-691	2.57%	0.00%	0.65%	0.35%	0.00%
CWE-693	4.66%	0.00%	0.47%	1.73%	38.03%
CWE-697	0.03%	1.16%	0.02%	0.12%	0.00%
CWE-703	0.30%	1.16%	0.91%	0.34%	0.00%
CWE-707	32.03%	9.25%	10.40%	33.19%	47.59%
CWE-710	1.87%	0.00%	5.18%	3.43%	0.00%

coverage [178], [179].

Sample Design

A.N. Kiaer proposed the "representative sampling" method in the 1800s, and described a **representative sample** as a miniature of the actual population [179]. In the 1900s Sir A.L. Bowley formalized Kiaer's method and introduced two sampling designs for which accuracy measures and equal inclusion probabilities could be computed: *simple* and *stratified* random sampling [178]. The first step to obtaining a sample in either design is to develop a *sampling frame*, i.e., a list of all elements in the population. Our sampling frame includes the 55,128 accepted CVEs with an associated CWE. A high-level overview of our workflow is provided in Figure 3.11. First, we extracted the data published to the CVE from 2010-2019, then we added the corresponding CWE pillar information to each CVE. We then proceeded to take both simple and stratified random samples of the datasets, and finally compared the results to the distribution of the original population to determine which sampling method provided the most accurate results.

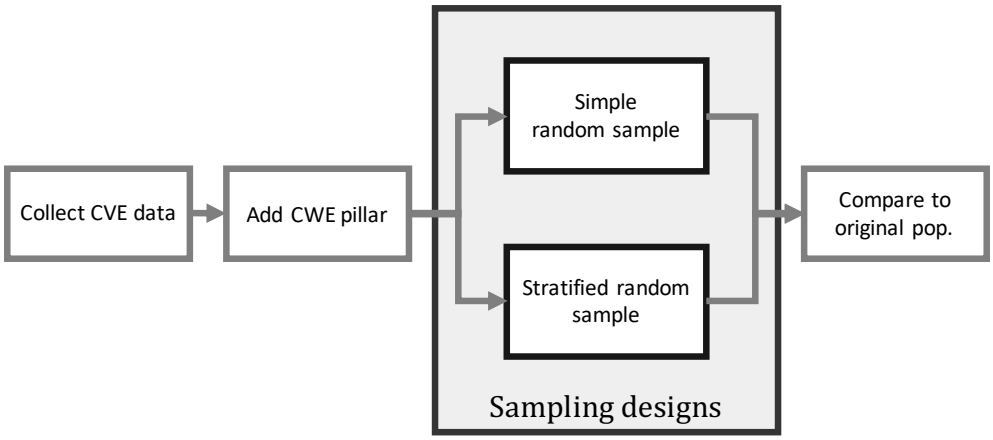


Figure 3.11. High level workflow depicting sample design choice

Ultimately, we determined that the stratified random sample provided the most accurate representation of the original population in the real-world dataset. This process, and the corresponding results are discussed in the next few sections.

Simple Random Sampling A simple random sample is one in which every possible sample of size n from a population of N elements has an equal probability of selection; where

the probability of an element N being selected can be determined using the following formula [179]:

$$\frac{N!}{n!(N-n)!} \tag{3.3}$$

Simple random sampling can be conducted with or without replacement. In the former, each element is replaced after it has been drawn, and thus, eligible for selection in subsequent draws. In simple random sampling without replacement, elements are removed after they have been drawn, and therefore are not eligible for selection in subsequent draws. We perform simple random sampling without replacement of the CVE data, then aggregate the results by their associated CWE pillar. Table 3.5 shows the resulting sample size of each pillar when using simple random sampling without replacement.

Table 3.5. Simple random sample size of each CWE pillar using accepted CVEs published from 2014-2019

Pillar	Total CVEs	Simple Random Sample
CWE-284	5,847	85
CWE-435	40	0
CWE-664	24,957	354
CWE-682	1,397	20
CWE-691	1,419	28
CWE-693	2,571	35
CWE-697	15	0
CWE-703	168	1
CWE-707	17,657	223
CWE-710	1,030	12

Stratified Random Sampling Stratified sampling reduces the variances of estimators (i.e., how far an estimator deviates from an expected value [180]) and improves the efficiency of the sample design. In stratified sampling, additional information about the sampling frame is used to partition the population into L strata, then a sample is selected from each stratum

using simple random sampling [180]. The sample size for each strata is calculated as follows:

$$n_h = (\frac{N_h}{N}) \cdot N \tag{3.4}$$

To determine the appropriate sample size for each strata we apply Cochran's formula. Cochran's original formula is as follows:

$$n_0 = \frac{Z^2 pq}{e^2} \tag{3.5}$$

Where n_0 is the sample size, Z is the Z-value (95%), p is the estimated proportion of the population with the given attribute (0.5, maximum variability), q is $1 - p$, and e is the margin of error.

Unlike random sampling, which may result in the misrepresentation of vulnerability instances and weakness types, stratified sampling allows us to preserve the relative proportions of each pillar, i.e., strata. Table 3.6 shows the sample size of each pillar using proportionate stratified sampling.

Table 3.6. Stratified sample size of each CWE pillar using accepted CVEs published from 2014-2019

Pillar	Total CVEs	Stratified Sample
CWE-284	5,847	245
CWE-435	40	2
CWE-664	24,957	1,042
CWE-682	1,397	58
CWE-691	1,419	59
CWE-693	2,571	107
CWE-697	15	1
CWE-703	168	7
CWE-707	17,657	737
CWE-710	1,030	43

Table 3.7 shows a comparison of sampling methods. In the worst case, stratified random

sampling results in a 0.02% difference between the sample size, and the actual population. On average, there is a 0.00% delta between the stratified random samples, and the actual population. When using simple random sampling there is a 2.61% difference between the sample size and actual population, in the worst case. When using simple random sampling there is an average delta of 0.60%. By taking a stratified random sample of the data we more accurately represent the original distribution of vulnerability instances and weakness types in BVATT.

Table 3.7. Comparison of simple and random sampling methods

Pillar	Total Population %	Random Sample %	Stratified Sample %
CWE-284	10.66	11.21	10.65
CWE-435	0.07	0.00	0.09
CWE-664	45.27	46.70	45.28
CWE-682	2.53	2.64	2.52
CWE-691	2.57	3.69	2.56
CWE-693	4.66	4.62	4.65
CWE-697	0.03	0.00	0.04
CWE-703	0.30	0.13	0.30
CWE-707	32.03	29.42	32.03
CWE-710	1.87	1.58	1.87

The stratified sample reported in Table 3.7 includes only CWE pillar nodes, however we also took a stratified sample including CWEs that are direct descendants of each CWE pillar. For example, pillar node CWE-284 has 27 children that are direct descendants. Each child is a distance of one away from the pillar, and the children's children are a distance of two away. With each level of abstraction, the CWEs become more granular. Table 3.8 shows the distribution of the stratified sample when including nodes one level away from each pillar node.

We also examined the impact of including CWEs a distance of two away from each pillar node when taking the stratified sample. We observed that three out of the ten CWE pillars contain children who are leaf nodes. For example, the maximum depth of Pillar CWE-682, "Incorrect Calculation," is one. Meaning that all children of CWE-682 are leaf nodes, and therefore have no children of their own. As a result, we are unable to achieve a complete

Table 3.8. Stratified sample (SS) of CVEs using CWE pillars and CWEs 1 node away compared to available test cases

Pillar	L1 Child	SS
CWE-284		245
	CWE-269	90
	CWE-285	52
	CWE-287	101
	CWE-346	2
	CWE-923	0
CWE-435		2
	CWE-436	2
CWE-664		1,041
	CWE-118	549
	CWE-400	27
	CWE-404	17
	CWE-405	0
	CWE-471	0
	CWE-610	48
	CWE-665	9
	CWE-666	10
	CWE-668	314
	CWE-669	21
	CWE-673	13
	CWE-704	9
	CWE-706	8
	CWE-749	0
	CWE-913	13
	CWE-922	3
CWE-682		58
	CWE-128	1
	CWE-131	0
	CWE-190	49
	CWE-191	3
	CWE-193	0
	CWE-369	5
CWE-691		245
	CWE-362	18
	CWE-670	4
	CWE-674	2
	CWE-834	11
	CWE-94	25
CWE-693		106
	CWE-179	1
	CWE-184	0
	CWE-311	7
	CWE-326	5
	CWE-327	4
	CWE-330	5
	CWE-345	82
	CWE-358	1
	CWE-424	1
	CWE-602	0
CWE-697		1
	CWE-185	1
CWE-703		7
	CWE-754	3
	CWE-755	4
CWE-707		737
	CWE-116	1
	CWE-172	0
	CWE-20	237
	CWE-74	499
CWE-710		43
	CWE-476	42
	CWE-684	1

stratified sample by using CWEs a distance of two away from each pillar node. The maximum depth we are able use and maintain a complete stratified sample of each pillar is one. Tables 3.9 and 3.10 illustrate this restriction. For each CWE that is a leaf node, “NONE” is printed in the corresponding “L2 Child” column indicating that CWE has no children, and the stratified sample was unable to be completed at that level.

Table 3.9. Stratified sample (SS) of CVEs using CWE pillars and CWEs a distance of two away from each pillar. CWE pillars: 284, 435, 664

Pillar	L1 Child	L2 Child	SS
CWE-284			179
	CWE-269		66
		CWE-250	66
	CWE-285		38
		CWE-552	4
		CWE-732	21
		CWE-862	5
		CWE-863	8
	CWE-287		74
		CWE-261	34
		CWE-290	1
		CWE-294	0
		CWE-295	12
		CWE-306	4
		CWE-307	1
		CWE-521	1
		CWE-522	8
		CWE-640	2
		CWE-798	11
	CWE-346		1
		NONE	
	CWE-923		0
		CWE-297	0
CWE-435			1
	CWE-436		1
		CWE-444	1
		CWE-86	0
CWE-664			768
	CWE-118		405
		CWE-119	405
	CWE-400		20
		CWE-770	20
		CWE-920	0
	CWE-404		12
		CWE-262	2
		CWE-459	0
		CWE-763	0
		CWE-772	10
	CWE-405		0
		CWE-406	0
		CWE-407	0
	CWE-471		0
		CWE-291	0

Pillar	L1 Child	L2 Child	SS
CWE-664			768
	CWE-610		36
		CWE-15	1
		CWE-384	4
		CWE-441	7
		CWE-470	0
		CWE-601	10
		CWE-611	14
	CWE-665		7
		CWE-1188	1
		CWE-454	2
		CWE-770	3
		CWE-908	0
		CWE-909	1
	CWE-666		8
		CWE-415	6
		CWE-672	2
	CWE-668		231
		CWE-134	2
		CWE-200	175
		CWE-22	48
		CWE-427	3
		CWE-428	1
		CWE-642	0
		CWE-8	2
	CWE-669		15
		CWE-212	0
		CWE-434	15
		CWE-494	0
		CWE-829	0
	CWE-673		10
		CWE-426	10
	CWE-704		7
		CWE-588	6
		CWE-681	0
		CWE-843	1
	CWE-706		6
		CWE-178	0
		CWE-22	0
		CWE-59	6
	CWE-749		0
		CWE-618	0
	CWE-913		9
		CWE-502	9
		CWE-94	0
	CWE-922		2
		CWE-312	2

Table 3.10. Stratified sample (SS) of CVEs using CWE pillars and CWEs a distance of two away from each pillar. CWE pillars: 682, 691, 693, 697, 703, 707, 710

Pillar	L1 Child	L2 Child	SS
CWE-682			43
	CWE-128		1
		NONE	-
	CWE-131		0
		NONE	-
	CWE-190		36
		NONE	-
	CWE-191		2
		NONE	-
	CWE-193		0
		NONE	-
	CWE-369		4
		NONE	-
CWE-691			43
	CWE-362		14
		CWE-364	13
		CWE-367	1
	CWE-670		2
		CWE-480	0
		CWE-617	2
	CWE-674		1
		CWE-776	1
	CWE-834		8
		CWE-835	8
	CWE-94		18
		CWE-95	18
CWE-693			80
	CWE-179		1
		CWE-180	1
	CWE-184		0
		NONE	-
	CWE-311		5
		CWE-312	2
		CWE-319	3
	CWE-326		4
		CWE-261	4
	CWE-327		3
		CWE-328	3
		CWE-916	0
	CWE-330		4
		CWE-329	2
		CWE-331	1
		CWE-335	0
		CWE-338	1

Pillar	L1 Child	L2 Child	SS
CWE-693			80
	CWE-345		61
		CWE-346	2
		CWE-347	3
		CWE-352	56
		CWE-924	0
	CWE-358		1
		NONE	-
	CWE-424		1
		CWE-425	1
	CWE-602		0
		CWE-565	0
CWE-697			0
	CWE-185		0
		CWE-186	0
CWE-703			5
	CWE-754		2
		CWE-252	2
		CWE-273	0
		CWE-354	0
	CWE-755		3
		CWE-209	3
CWE-707			543
	CWE-116		0
		CWE-117	0
		CWE-838	0
	CWE-172		0
		CWE-173	0
	CWE-20		175
		CWE-114	172
		CWE-129	3
	CWE-74		368
		CWE-75	15
		CWE-77	42
		CWE-79	273
		CWE-91	1
		CWE-93	2
		CWE-943	65
		CWE-99	0
CWE-710			32
	CWE-476		31
		CWE-690	31
	CWE-684		1
		CWE-451	1

3.3.6 Test Case Selection

The total number of test cases in BVATT corresponding to each CWE pillar should be equal to the stratified sample size for that pillar. We must now determine how to select particular test cases for each strata. Each test case should contain at least one known vulnerability that corresponds to a CWE in one or more strata. In Section 3.3.6 we explore the possibility of reusing test cases from existing vulnerability datasets.

Existing vulnerability datasets have been previously discussed in Sections 2.4 and 3.1.1 of this work. In this section, we summarize some of the strengths and weaknesses of these datasets, and explore the possibility of reusing test cases from the datasets for BVATT.

Table 3.11[8] describes 11 publicly available vulnerability datasets, and provides the: corresponding source code language, type of code (i.e., real-world (RW), Synthetic (S), and Hybrid Synthetic (HS)), granularity of vulnerability labels, and whether the test cases in that dataset are able to be compiled.

Table 3.11. Open source vulnerability datasets.

Open source vulnerability datasets. Code types: real-world (RW), Synthetic (S), and Hybrid Synthetic (HS). Compilable: Yes (Y), No (N)

Dataset	Language	Type	Label Granularity	Known Vulnerabilities	Compilable
CB-Multios	C	HS	File	888	Y
Juliet C#	C#	S	Function	28,942	Y
Juliet C/C++	C/C++	S	Function	64,099	Y
Juliet Java	Java	S	Function	28,881	Y
LAVA M	C	HS	File	2,265	Y
OWASP Benchmark	Java	S	File	2,740	Y
STONESOUP	C & Java	HS	File	7,770	Y
FLVD	C	RW	Function	118	N
VDISC	C	RW	Function	n/a	N
LinuxFlaw	C	RW	File	368	Y
ferenc	JavaScript	RW	Function	1,496	N

Collectively, these datasets contain over 150K known vulnerabilities. A stratified random sample indicates that a grand total of roughly 1.2K test cases are needed for BVATT to be representative of the real-world. We summarize some of the strengths and weaknesses of these datasets as follows:

[8]duplicated from Section 2.4.10 for convenience

1. 82% of these datasets only provide test cases in one source code language.
2. In 6 of the datasets, vulnerabilities are labeled at a file level; while in 5 of the datasets vulnerabilities are labeled at a function level.
3. Collectively, these datasets provide a 4:4:3 ratio of pure synthetic, real-world, and hybrid synthetic code.
4. 8 of the 11 datasets provide code that is able to be compiled.
5. We have demonstrated that most of these datasets provide an unbalanced ratio of vulnerability types [22].

Providing vulnerability labels at a finer granularity, e.g., function level, has been shown to provide a more insightful and quantitative assessment of the performance of different vulnerability identification techniques [137]. Pure synthetic vulnerabilities have been criticized for being overly simplistic and not representative of vulnerabilities in the real-world, while some of the real-world repositories are limited because they provide source code that cannot be compiled. Some VATTs operate on source code, e.g., [75], [88], [181], while others, are compatible with compiled code [80], [83], [137].

Considering each of these observations, we propose that, in addition to providing adequate coverage of CWEs, BVATT be comprised of test cases that offer the following:

1. a mixture of real-world, synthetic, and hybrid synthetic code
2. a mixture of source code languages
3. source code that is able to be compiled
4. both function and file level vulnerability labels

Given these additional requirements for BVATT, we eliminate the FLVD, VDISC, and ferenc datasets from consideration as the test cases within those datasets are not able to be compiled. We now consider how to down-select test cases from the remaining datasets to the number required for BVATT. We reduce the test-suite not only to be reflective of the stratified random sample, but also because the size of a benchmark has been shown to be proportional to the cost, e.g., execution time, and level of effort related to testing and comparing VATTs [182].

Test-Suite Reduction Strategies

There are a number of existing methods to reduce a test-suite (also referred to as *test-suite minimization*, or *test-suite reduction*). Unfortunately, the majority of these methods apply to software testing in a traditional sense— where the focus is on fault detection [183]–[190]. In the case of BVATT, a VATT analyzes a test case containing at least one known vulnerability, attempts to identify the vulnerability, and reports any findings.

Nevertheless, we consider the following test-suite reduction techniques from fault detection-based software detection: identify and eliminate redundant and obsolete test cases [187], use a fuzzy clustering algorithm to identify redundant cases [188], use the ratio of code coverage to test case cost [189], and use a dynamic call tree or calling context tree to identify a subset of test cases that cover the same call tree paths [190]. Techniques that are based on code coverage, obsolete test cases, and call trees are not relevant to our particular use case. However, techniques to prune redundant test cases may be applicable. In the following section we explore two methods to identify and eliminate redundant test cases from a corpus.

Identify and Eliminate Redundant Test Cases

In traditional software testing a redundant test case is one that provides the same coverage measure as another test case, so the removal of a redundant test case will not impact the overall testing coverage [187]. Within the context of test-suite reduction for BVATT, we define a **redundant test case** as a test case with greater than a specified measure of similarity to another test case. Determining whether two programs are equivalent in the general case is a well known undecidable problem[9]. Thus, we specifically focus our efforts on deciding whether the programs exceed a *similarly threshold* based on a set of defined *features*.

Test Suite Reduction Based on Vulnerability Characteristics

Matt Bishop proposed a representation of vulnerabilities based on a set of *characteristics* that he described as the, "conditions that must hold for a vulnerability to exist" [191].

Bishop postulated that the similarity between vulnerabilities could be determined by applying set intersection to basic sets of vulnerability characteristics. In 2002, Sophie Engle refined this definition and described characteristics in terms of preconditions, then applied the

[9]See Section 4.6.1

methodology to identify a set of characteristics for stack-based buffer overflows [192]. Table 3.12 outlines 11 characteristics related to buffer overflow vulnerabilities. For example, the $x{:}jmps$ and $x{:}exes$ characteristics describe the ability to jump (can_jump) to memory on the stack and execute instructions (can_exec) stored on the stack.

Table 3.12. Buffer overflow characteristics

Characteristic	Description	Value
x:buff	len(input) > len(buffer)	n/a
x:jmps	can_jump(stack)	true
x:exes	can_exec(stack)	true
x:rval	may_modify(retnptr)	true
x:fptr	may_modify(funcptr)	true
x:vval	may_modify(flowvar)	true
x:vptr	may_modify(flowptr)	true
x:path	affects_flow(flowvar)	true
x:addr	may_contain(input, addr)	true
x:inst	may_contain(input, inst)	true
x:type	may_contain(input, type(flowvar))	true

Table 3.13. Buffer overflow basic characteristic sets

Type	Basic Characteristic Set
Direct executable buffer overflow	{x:buff, x:addr, x:inst, x:rval, x:jmps, x:exes}
Indirect executable buffer overflow	{x:buff, x:addr, x:fptr, x:jmph, x:exes}
Direct data buffer overflow	{x:buff, x:type, x:vval, x:path}
Indirect data buffer overflow	{x:buff, x:addr, x:vptr, x:path}

Table 3.13 describes the basic characteristic sets for four buffer overflow vulnerability equivalence classes [193]. In an executable buffer overflow, the return address($addr$) or function pointer($fptr$) is modified to either directly or indirectly cause code($exes$) that was placed into a buffer($buff$) to be executed. Alternatively, a data buffer overflow occurs when overflow from the buffer alters the content of another variable on the stack.

We note two weaknesses with this method to determine if test cases are similar. First, the process of reviewing the source code and identifying relevant characteristics is entirely manual. While the effort is likely not significant for a few test cases, we have over 150K test cases to review. The application of a manual method to all cases in our test suite would be a significant undertaking. Second, Bishop and Engle have only established the characteristics for stack and heap based buffer overflow vulnerabilities [192], [194]. Thus, to use their technique, we would need to develop characteristics and vulnerability classes for each other type of vulnerability in the stratified sample. This is a significant undertaking, and thus we choose to explore other, less manual alternatives.

Test Suite Reduction Based on Feature Vectors

In Section 3.3.6 we explored a method to identify and eliminate test cases based on their characteristics, and determined that such a method would require significant manual effort. Instead of using the characteristic method proposed by Bishop and Engle [192], [194], we propose a more automated method, based on the *feature vector* for each test case. By representing each test case as a feature vector, we can plot each vector as a point in an n-dimensional space, where n is equal to the number of distinct features in the vector. We can then calculate the geometric distance between any two points, or in our case, test cases. The closer two points are the more similar the feature vectors are for those samples. Or, in the context of test suite reduction, the farther two points are from one another, the more dissimilar the feature vectors are for those samples.

In order to leverage the proposed technique we require a means to extract features from over 150K test cases. After extraction, the features must be cleaned and transformed into vectors so that they are compatible with machine learning classifiers. The classifiers can be used to cluster the test cases based on how similar they are, and finally, we can identify the least similar set of test cases for BVATT. These requirements provide the basis for the remainder of our work, including the creation of the tool, `BiSECT`. `BiSECT` allows users to extract a number of common features from compiled binaries, and transform them into a format compatible with traditional machine learning and data mining techniques. `BiSECT` and a few example applications of the tool (including using it to facilitate test-suite reduction) are described in Chapter 4.

3.4 Summary

In this chapter, we provided a framework for BVATT. We first examined existing datasets containing known vulnerabilities, and explored their associated limitations. This was followed by lessons learned from existing benchmark implementations. From these, we determined that BVATT should be repeatable, reproducible, fair, verifiable. After exploring each of these characteristics, we investigated what it would require for BVATT to be relevant, that is, representative of reality. To this end, we determined that a stratified random sample of CVEs and corresponding CWEs would provide a perfect representation of known vulnerability instances and types that have occurred in the wild throughout the past decade. Finally, we explored methods to leverage test cases in existing vulnerability datasets for BVATT. In Section 3.3.6 we demonstrated that existing methods to identify and eliminate similar test cases may require significant manual effort. We sought a more automated method, which ultimately led us to create a new tool, `BiSECT`. `BiSECT` allows users to extract a number of common features from compiled binaries, and transform them into a format compatible with traditional machine learning and data mining techniques. In Chapter 4 we give a full introduction of `BiSECT`, and provide a couple of use cases to demonstrate the various utilities of the tool. In Section 4.6 we demonstrate how `BiSECT` can be used to identify and eliminate similar test cases for BVATT.

CHAPTER 4:
BiSECT: Binary Synthesized Extraction, Cleaning, and Transformation

In their early days, computer security and artificial intelligence didn't seem to have much to say to each other. AI researchers were interested in making computers do things that only humans had been able to do, while security researchers aimed to fix the leaks in the plumbing of the computing infrastructure or design infrastructures they deemed leakproof. [...] But the two fields have grown closer over the years

Carl E. Landwehr, *Cybersecurity and Artificial Intelligence*

4.1 Introduction

Today, software is integrated into nearly every aspect of our lives, and so are its vulnerabilities. Between 1999 and early 2020 over 150,000 software vulnerabilities were reported by the Common Vulnerabilities and Enumerations (CVE) database [6]. In 2020, the Ponemon Institute reviewed over 500 data breaches (often resulting from exploited vulnerabilities), and found that the cost of a single breach averaged US$3.8M [8]. Figure 4.1 shows the findings of the Ponemon study by industry. The average costs are measured by direct loss of business, and also account for detection, notification, and post-breach recovery efforts.

The Stuxnet and Duqu worms demonstrated how vulnerability exploitation can go beyond financial damage, and disrupt critical operations [9], [10]. In 2020, the Sunburst malware was used to compromise thousands of government systems and an estimated 18,000 firms [13]. Despite substantial investments in system security, vulnerabilities persist. Recent data exfiltration [14] and ransomware [15] attacks further illustrate the cybersecurity problem. In May 2021, the US government issued an order requiring the development of guidance related to the employment of, "automated tools, or comparable processes, that check for known and potential vulnerabilities" [23].

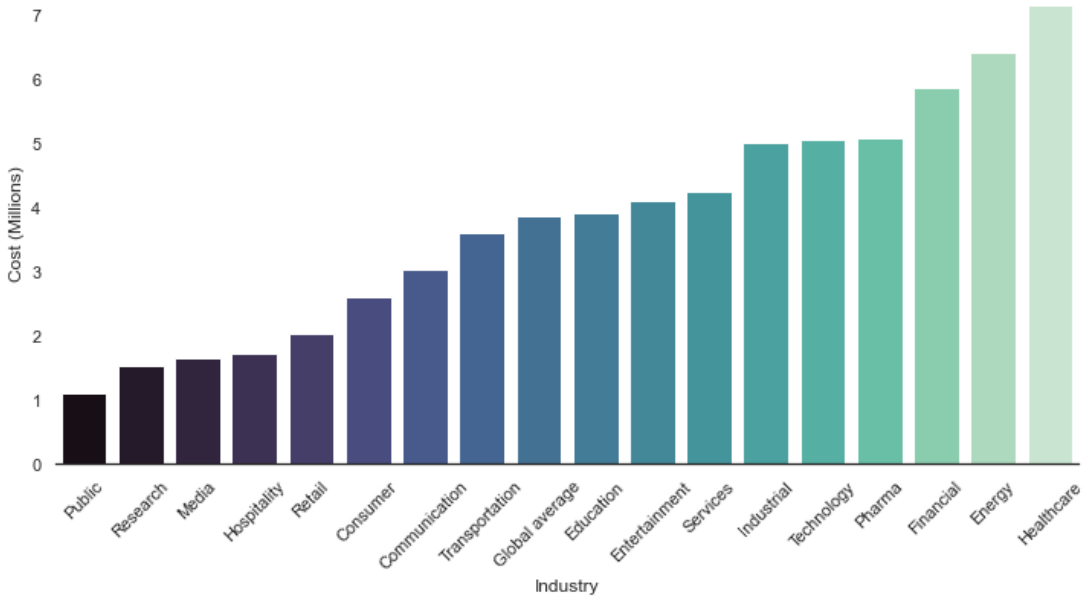


Figure 4.1. Average cost in millions per data breach in 2020. Source: [8].

Yet the question remains: *how can we identify and reduce software vulnerabilities?* One reduction approach is to begin with the assumption that all software contains vulnerabilities, then test each program to find as many vulnerabilities as possible [17]. The pervasiveness and variety of software is disproportionate to the number of individuals able to manually identify and mitigate vulnerabilities in it. *Ergo*, in an attempt to reduce this monumental and seemingly unending problem, researchers have turned to tools and techniques (i.e., VATTs) to amplify the vulnerability analysis process. We discussed a number of VATTs in Chapters 2 and 3. In Chapter 2 we also explored the various limitations of current VATTs. For example, early static analysis tools, such as Lint and Flawfinder are criticized for having low precision rates and minimal semantic insights [69], [75]. Dynamic analysis techniques such as fuzzing, dynamic symbolic execution, and taint analysis offer increased semantic insights, but are limited by issues including complex memory management and the path explosion problem [19], [21], [79]–[81], [88]. In addition to traditional static and dynamic techniques, researchers are also using data mining and machine learning techniques to identify and analyze software vulnerabilities [27], [28]. These techniques have shown encouraging results; however, *there is no tool to synthesize the at scale extraction, cleaning, and transformation*

of features commonly used in data mining and machine learning based vulnerability analysis. Consequently, such advanced techniques are left to a niche group of computer scientists who are experts in both the vulnerability analysis and machine learning domains. This dual requirement acts as barrier to both security and machine learning researchers, and ultimately limits advancements in a promising area of vulnerability analysis. The limitations of machine learning-based software vulnerability analysis are further compounded when the target is compiled code (i.e., binary samples, binaries). According to two major surveys, only 3% of machine learning- and data mining-based vulnerability research leveraged binary samples, while the remaining 97% requires the use of high level source code [27], [28]. We believe that this disparity is due to an additional increase in the domain expertise required to analyze assembly code lifted from the binaries; it lacks the rich semantic information offered by high level languages (e.g., C, Java, etc.).

In this chapter we introduce a new tool, `BiSECT` (Binary Synthesized Extraction, Cleaning, and Transformation). `BiSECT` breaks down the aforementioned barriers by providing a user-friendly, synthesized, and repeatable means to extract and transform common features from compiled code (i.e., samples).

To summarize, we make the following contributions throughout this Chapter:

1. Provide the tool, `BiSECT`, to synthesize the extraction, cleaning, and transformation of common features from compiled binaries to be compatible with data mining and machine learning techniques.
2. Use the output of `BiSECT` to assess the efficacy of two representation models (and corresponding classifiers), *fastText* and *Doc2Vec*, when they are given the task of labeling potentially vulnerable functions.
3. Examine the impact of using balanced and imbalanced datasets when training the *fastText* and *Doc2Vec* classifiers on binary samples.
4. Use `BiSECT` to prepare the $fuzzyInstruction$ sequence in support of binary vulnerability analysis.
5. Use the $fuzzyInstruction$ sequence to train a classifier able to label function-level vulnerabilities in a balanced hybrid-synthetic assembly code base with 96.4% accuracy, 97.8% precision, 94.8% recall, a False Positive Rate of 2.1%, False Negative Rate of 5.2%, and F1 of 96.3%

6. Demonstrate how BiSECT can be leveraged to identify the most dissimilar test cases in a test suite

4.1.1 Motivation and Chapter Outline

In addition to traditional static and dynamic techniques, in the past decade a number of researchers have begun to leverage machine learning and data mining techniques to support vulnerability analysis. The first four steps of the data mining process include data cleaning, integration, selection, and transformation [37]. While machine learning typically begins with the transformation step, it is assumed that certain preprocessing steps, i.e., the first few steps in the data mining process, have already been completed [38]. Both the machine learning and data mining processes also assume that the initial dataset has already been created. We find these assumptions to be significant, as the extraction and transformation of even a single feature from a binary file requires domain expertise. By **feature**, we simply mean a measurable property or characteristic of a program.

To demonstrate how nontrivial the extraction and transformation of features can quickly become, we provide two examples. The first, in Section 4.2.3, details methods to identify the McCabe cyclomatic complexity for each function in a binary file. The second, in Section 4.2.4, details the identification and extraction of mnemonic n-grams from a binary. After demonstrating the domain expertise required to manually extract features such as these, we discuss techniques to automate the process using Python and Ghidra. In Section 4.3 we explore the process to clean, transform, and finally, to amalgamate the data into a feature vector that is compatible with conventional machine learning and data mining techniques.

The domain expertise required to complete the feature extraction, cleaning, and transformation steps results in a barrier to entry that in turn inhibits advancements in the application of machine learning and data mining techniques to analyze software vulnerabilities. To reduce the domain expertise required to perform these steps, we propose a new tool, BiSECT. BiSECT enables researchers to synthesize the extraction of a number of common features from compiled binaries, and transform them into a format compatible with traditional machine learning and data mining techniques. At its core BiSECT is a collection of stand-alone Python scripts, Jupyter notebooks [195], and scripts intended to be used with the open source software reverse engineering (SRE) suite, Ghidra [168]. We designed BiSECT to be

user friendly, however, to further support researchers, BiSECT is accompanied by a Jupyter notebook that provides detailed documentation, and uses simple examples to illustrate some of the key functionality provided by the tool.

To demonstrate the utility of BiSECT, two example applications are described in Sections 4.5 and 4.6. In the first application, provided in Section 4.5, we use BiSECT to extract, clean, and transform roughly 5M disassembled x86 functions from 64K compiled C \ C++ programs. We then use the output of BiSECT to compare the efficacy of two representation models and corresponding classifiers, *fastText* and *Doc2Vec*, when they are given the task of labeling potentially vulnerable functions. In the second application, provided in Section 4.6, we use BiSECT to identify similar test cases for BVATT.

Throughout the next few sections we explore the key functionalities provided by BiSECT. To add clarity to our discussion we reference Figure 4.2 numerous times. In each major section we repeat the figure with the current area of exploration in color, and the remainder of the figure in light gray.

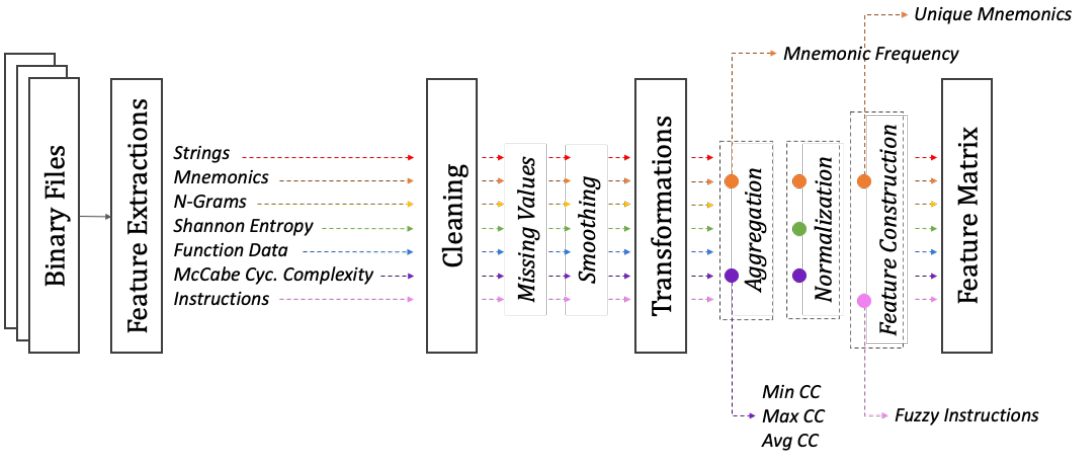


Figure 4.2. The complete high-level workflow for BiSECT

Malware Analysis

Like vulnerabilities, the amount and variety of malware is disproportionate to the number of researchers able to analyze it. The malware community uses an array of tools and techniques to amplify the malware analysis process— Some of these involve machine learning and data

mining. Common feature types used to support machine learning-based malware detection and analysis include: byte sequences, permissions, intents, activities, services, providers, information flow, APIs/system calls, opcodes, network, file system, CPU registers, Portable Executable (PE) file characteristics, and strings or other cleartext information [196], [197]. A few of these features, including: opcodes, information flow, strings, and system calls, overlap with features that have demonstrated applicability to the vulnerability analysis community [27], [137].

Significant research has been published, and numerous techniques and features proposed to support static and dynamic machine learning-based malware detection and analysis[10]. However, there are limited open source tools available to support the synthesized extraction, cleaning, and transformation of multiple features from binary files in support of this work. The open source tools that *are* available primarily focus on the extraction, cleaning, and or transformation of 1-2 features and do not provide general support. For example, DroidAPIMiner can be used to extract critical API calls from Android applications. The tool also extracts API package level and parameter information, in addition to permission information [198]. Jackstab transforms machine code into an intermediate language to perform data flow analysis of the control flow graph for a binary. The tool produces the full disassembly (with all reachable instructions), a Control Flow Graph (CFG) in the intermediate language, and a CFG of the assembly instructions [199]. KiloGram facilitates the extraction of the, "the top-k most frequent n-grams for large values of k and n" [200]. Manalyze parses and statically analyzes PE files for information such as: compiler version, whether the file is packed, languages detected, header information, dynamically linked libraries and functions, and debug information [201].

In 2008, Cavalca and Goldoni proposed HIVE (now referred to as "TheHive Project"), an open source honeynet to provide, "rapid comprehension and detailed data analysis," of malware samples [202]. TheHive Project includes the Cortex toolset. Cortex allows users to analyze "observables" or features, i.e., measurable property or event, at scale by querying a single tool. To the best of our knowledge TheHive project, specifically the Cortex component, provides the most comprehensive toolset that enables the extraction, cleaning and transformation of features from malware samples. Cortex currently facilitates the extraction of observables such as: IP and email addresses, URLs, domain names, files or

[10]See [196] for a recent review of such work.

hashes. Unfortunately, most of the features that have demonstrated utility in vulnerability analysis are not extracted by TheHive Project at the time of this writing.

The opportunity to develop a toolset to synthesize the extraction, transformation, and cleaning of features in support of binary vulnerability analysis persists, and provides a fundamental motivation for the creation of `BiSECT`. Like Cortex, `BiSECT` reduces the manual effort required to extract common features from binary files by synthesizing and standardizing the output from multiple tools.

4.2 Feature Extraction

In this section, we explore one of the core functionalities provided by `BiSECT`, feature extraction, as depicted in Figure 4.3.

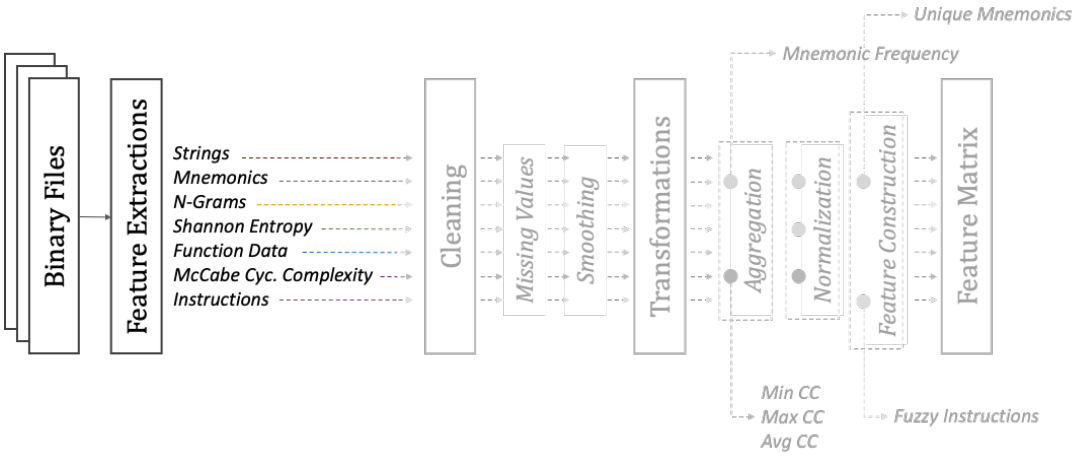


Figure 4.3. `BiSECT` Feature extraction component

Figure 4.4 provides a treemap of commonly leveraged predictors used in data mining and machine learning vulnerability analysis techniques, according to 58 works reviewed in two major surveys [27], [137]. Throughout the surveys 15 types of features were repeatedly used as predictors for vulnerabilities. These features are categorized by two branches of the same tree— if the tree is all machine learning and data mining methods currently used to identify vulnerabilities in software, one branch (shown in green on the treemap) includes “graph-based” features. All of the other features on the second branch (shown in

shades of brown) are considered "code-based". In general, the most prevalent predictors used throughout these surveys include the following: abstract syntax trees (AST) or other graphical representations of the program, API usage patterns, complexity metrics, and function metrics. Combined, these predictors were leveraged in 50% of the studies reviewed in the surveys. These predictors provide the basis for the features extracted by `BiSECT`.

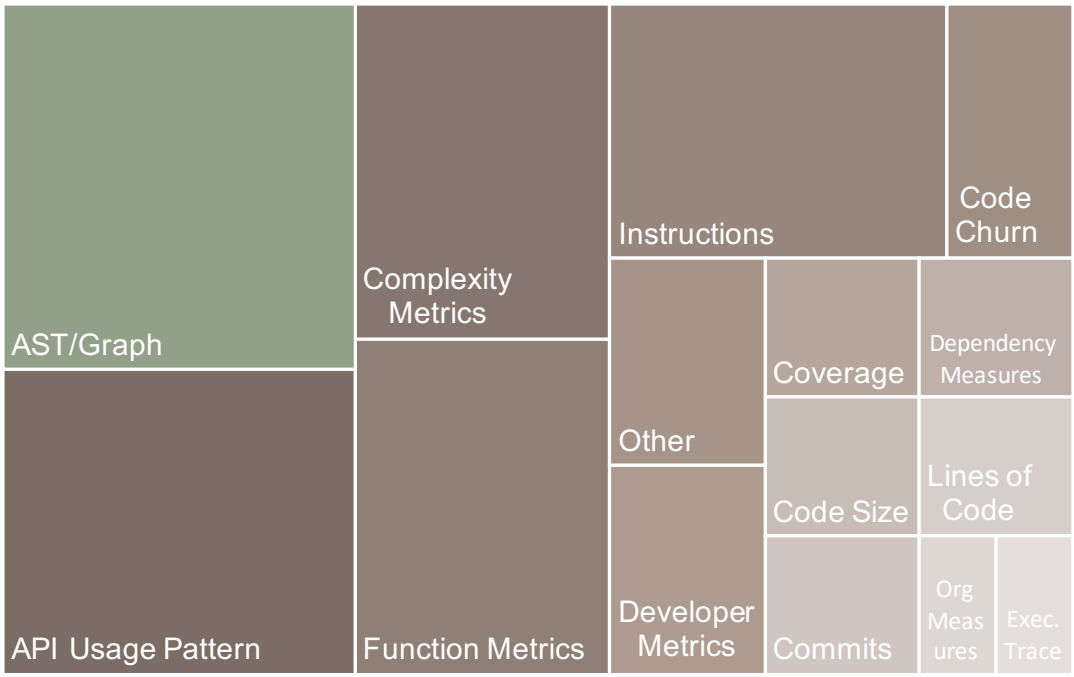


Figure 4.4. Treemap of most commonly used predictors in 58 publications, to support vulnerability analysis using data mining and machine learning. Adapted from [27], [137].

4.2.1 Feature Engineering

The term **feature engineering** has an array of definitions across the machine learning, data science, and data mining disciplines. Some definitions generalize the term in such a way that there is clear overlap with `BiSECT` [203], while others highlight clear delineations [204], [205]. Due to this ambiguity, we have decided to remove the term **feature engineering** from our work altogether, and instead focus on providing a clear description of the `BiSECT` tool.

4.2.2 Compiled Code Features

In 2017, Ghaffarian and Shahriari provided a survey of work that uses machine learning or data mining techniques to identify and analyze software vulnerabilities [27]. Of the 39 publications reviewed in the survey a total of 3, or roughly 8% [39]–[41] used compiled binaries to support their research (including one sample that used both binary and source code samples), the remaining 92% relied on source code level samples. Researchers have also begun to apply neural networks and deep learning techniques to software vulnerability identification and analysis. A comprehensive survey of notable work in this area was published in 2020, by Lin, et al. [28]. In their survey, the authors review 19 vulnerability detection methods that leverage deep neural networks published between 2013 and 2019. Of the 19 works reviewed, 5 used compiled code samples to support their work [40], [42]–[44][11], two used a mix of binary and source code samples [45], [46], and the remaining 13 used source code samples – 7 out of 19, or roughly 37%, is a significant increase from the mere 8% in the 2017 survey of conventional machine learning and data mining research that used binary samples [27]. Yet, the disparity between vulnerability analysis using source code verses compiled code persists.

During the compilation process much of the semantic and syntactic information offered by high-level languages (e.g., C, Java, etc.) is lost, resulting in samples that are significantly more difficult to analyze [21], [47], [48]. This level of difficulty is inversely related to the number of researchers using binary samples to support vulnerability research (using machine learning and data mining techniques). This relationship can be observed in the 58 papers reviewed from the two surveys where only 3% of the work reviewed leveraged compiled code samples [27], [28]. Yet, there are situations when the analysis of binary samples is desirable or necessary, for example, a user may need to validate that properties proven by analyzing a program's source code still hold after the program has been compiled, or simply may not have access to a program's source code [21], [47], [49], [50]. To support and encourage research in binary code vulnerability analysis, `BiSECT` was designed to extract features specifically from binary samples.

While some features (e.g., file size) can be extracted from a raw binary file, most features rely of the presence of some level of semantic or syntactic information, and thus, the binary must be disassembled prior to analysis. We note that many of these features could also be derived

[11] Grieco, et al. [40] was referenced in both the 2017 and 2020 survey

by writing a custom parser, however such methods are beyond the scope of this research.[12] Before reviewing the specific features able to be extracted and curated by `BiSECT`, we first introduce various *feature types*.

Feature Types

An initial assumption in the machine learning and data mining processes is that a dataset has already been created. Datasets are typically comprised of rows and columns. Rows equate to **objects** or individual entities, e.g., each binary sample could be an object in our dataset. The columns in the dataset correspond to one or more features or attributes associated with each object, where each feature has an associated *type*. Figure 4.5 illustrates the location of objects, features, and vectors in a dataset.

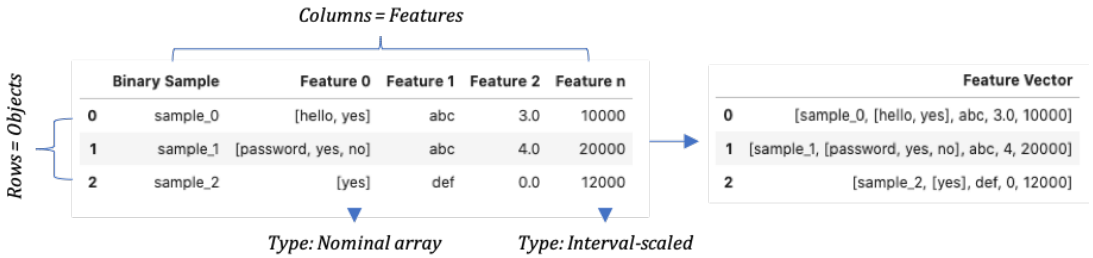


Figure 4.5. Objects, features, and feature vectors

The **type** of a feature is the set of possible values for that feature. We describe a number of feature types in Table 4.1 [37]. At a high level, feature types are either qualitative or quantitative. The former includes features that are described without using an actual quantity, while in the latter, integers or real values are used to represent the feature. Qualitative features can be nominal, binary, or ordinal. Quantitative features may be interval or ratio-scaled. Some features (e.g., ordinals) may have a meaningful order, while others, such as nominal and symmetric binary features, have no meaningful order.

`BiSECT` Features and Feature Types

Table 4.2 provides the features that can be extracted with `BiSECT`. Each feature was specifically selected based on its relevance to vulnerability analysis of binary files, demonstrated

[12]To the reader interested in custom parsers, we suggest the open source parser generator, ANTLR https://www.antlr.org/

Table 4.1. Description of commonly used feature types.

Feature Type	Ordered	Description
Qualitative		Feature described without using actual quantity
Nominal, *Nom*	Unordered	symbols or names of things (e.g., enumerations)
Binary, *Bin*		Feature with 2 states: 0 or 1 (0 typically means that the feature is absent, 1 that it is present)
Symmetric, Sym	Unordered	Both states equally valuable / no preference for a specific outcome
Asymmetric, Asy	Ordered	One state more important or desirable than the other. Convention is that the rarest outcome is coded as a 1, and the other by 0
Boolean, Bool	Unordered	Binary feature with two states: True and False
Ordinal, *Ord*	Ordered	possible values that have a meaningful order or ranking among them, but the magnitude between successive values is not known
Quantitative		
Numeric		Measurable quantity (integer or real value)
Interval-scaled, IS	Ordered	measured on a scale of equal-size units. Values may be positive, 0, or negative.
Ratio-scaled, RS	Ordered	an ordered, numeric feature containing an inherent zero-point (i.e., a multiple or ratio of another value)

usefulness in previous studies, and potential for use in future studies. Table 4.2 also indicates the granularity of each feature extracted. For example, the *Cyc. Complexity*, i.e., the McCabe cyclomatic complexity, is extracted at the function level for each sample, while the *Max Complexity* is aggregated at the file level. The "Related Research" column provides references to publications that leverage the corresponding feature.

Feature Extraction Techniques

To facilitate the synthesized extraction of features from binary files a myriad of tools and techniques can be used. Determining which tool or technique to apply largely depends on the specific feature being extracted. `BiSECT` synthesizes the feature extraction process so that users no longer have to write custom scripts, or use multiple tools to create commonly used features.

Table 4.2. Description of all raw and constructed features extracted by BiSECT

Feature	Related Research	Granularity
Strings	[206]–[208]	Func & File
Mnemonics	[209]–[211]	Func & File
Mnemonic Frequency		Func & File
Unique Mnemonics		Func & File
***N*-Grams**	[33], [110], [113], [212]	Func & File
Shannon Entropy	[213]–[216]	File
Function Data	[217]	File
Total Functions		File
Internal Functions		File
External Functions		File
Cyc. Complexity	[91]–[93], [181], [218]–[221]	Func
Max Complexity		File
Min Complexity		File
Avg Complexity		File
Instructions	[30], [31], [44], [181]	Func & File
FuzzyInstructions		Func

BiSECT relies solely on publicly available tools (namely, Ghidra), and custom developed Python scripts. BiSECT has been made completely open source[13], and is available on GitHub[14]. BiSECT can be run from the command line, or in a more interactive way via Jupyter. In fact, the entire BiSECT workflow is documented in a Jupyter Notebook. We hope is that these resources will be leveraged by researchers as both a learning and research aid. A synopsis of the technique that was used to extract each feature is provided in Table 4.3.

4.2.3 Feature Example: Cyclomatic Complexity

In 1976, Thomas McCabe introduced the concept of *cyclomatic complexity* [222]. The **McCabe cyclomatic complexity** is a software metric that identifies the maximum number of linearly independent paths in a program [222]. *Linearly independent* paths have at minimum one edge that is not shared by any other path [222]. The formula for McCabe cyclomatic

[13]The link will become active after the final dissertation has been approved!
[14]https://github.com/Kayla0x41/BiSECT

Table 4.3. Summary of the tool or technique used to generate each feature from binary samples

Feature	Custom Python	Ghidra Headless (Python)
Strings		•
Mnemonics		•
Mnemonic Frequency	•	
Unique Mnemonics	•	
N-Grams	•	•
Shannon Entropy	•	
Total Functions		•
Internal Functions		•
External Functions		•
Cyclomatic Complexity	•	•
Max Complexity	•	
Min Complexity	•	
Avg Complexity	•	
Instructions		•
fuzzyInstruction	•	

complexity (CYC) can be given in the context of a *flow graph*, or a directed graph containing the program's basic blocks. If program control can pass between two basic blocks in the graph there exists an edge between them, where the direction of the edge indicates the direction in which control *flows*. Mathematically, McCabe cyclomatic complexity can be expressed as follows [222]:

$$CYC = E - N + 2 * P \tag{4.1}$$

Where E is number of edges, N is the number of nodes, and P is the number of connected components.

Cyclomatic Complexity Extraction

Consider the example program in Listing 4.1 to calculate the Fibonacci sequence for 10 numbers. When the $main()$ function is called, a sequence of conditional statements begins.

Using the flow graph depicted in Figure 4.6 we can visualize the control flow for the program, and calculate the McCabe computational complexity for the $main()$ function. The number of edges E in the function is 7, the number of nodes N is 6 and the number of connected components P is 1, thus the CYC is 3.

$$CYC = 7 - 6 + 2 * 1$$

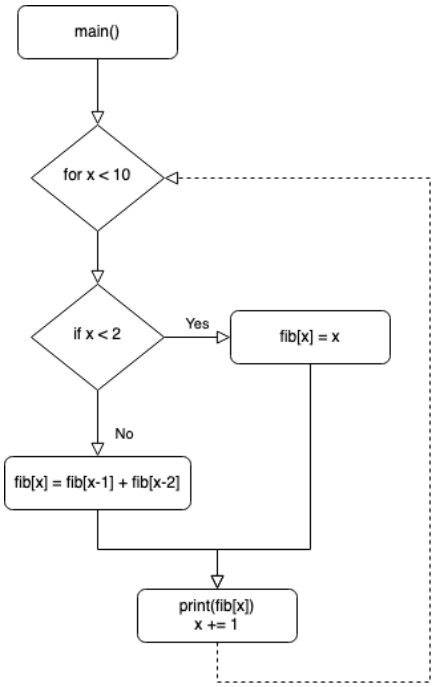


Figure 4.6. Control flow graph (CFG) for Fibonacci example code

The example in Listing 4.1 is trivial. Programs, even those that are synthetic, do not typically consist of a single function containing a few conditional statements. Moreover, the example provided here demonstrates how to calculate the McCabe cyclomatic complexity given the source code for a function. An additional level of complexity[15] is added when determining the McCabe complexity for functions in compiled code.

[15]pun intended

```c
int main(int argc, const char* argv[]) {
    int size = 10;
    int fib[size];
    int x;

    fib[0] = 0;
    fib[1] = 1;
    printf("Calculating Fibonacci for %d terms: \n", size);

    for (x = 0; x < size; x++) {
        if (x <= 1) {
            fib[x] = x;
        } else {
            fib[x] = fib[x-1] + fib[x-2];
        }
        printf("\t%d\n", fib[x]);
    }
    return 0;
}
```

Listing 4.1: Fibonacci example C code

To calculate the cyclomatic complexity beginning with compiled code, the binary must first be disassembled. The assembly can then be parsed at a function level, and the cyclomatic complexity calculated for each function by interpreting the assembly instructions.

A subset of the x86-64 assembly for the previous example is provided in Figure 4.7. The code was compiled using GCC version 9.3 with debug symbols included:

```
$ gcc -g -c fibonacci_example.c -o fibonacci_example.exe
```

The assembly was generated using the following objdump command:

```
$ objdump -d -S --no-show-raw-insn fibonacci_example.exe
```

For clarity, the assembly in Figure 4.7 has been partitioned into sections, and the C-code corresponding to each section is provided for a quick side by side comparison. The loss of the high level code's rich semantic and syntactic information during compilation is clearly evident. As a result of this loss, additional expertise is required to interpret the assembly and calculate cyclomatic complexity.

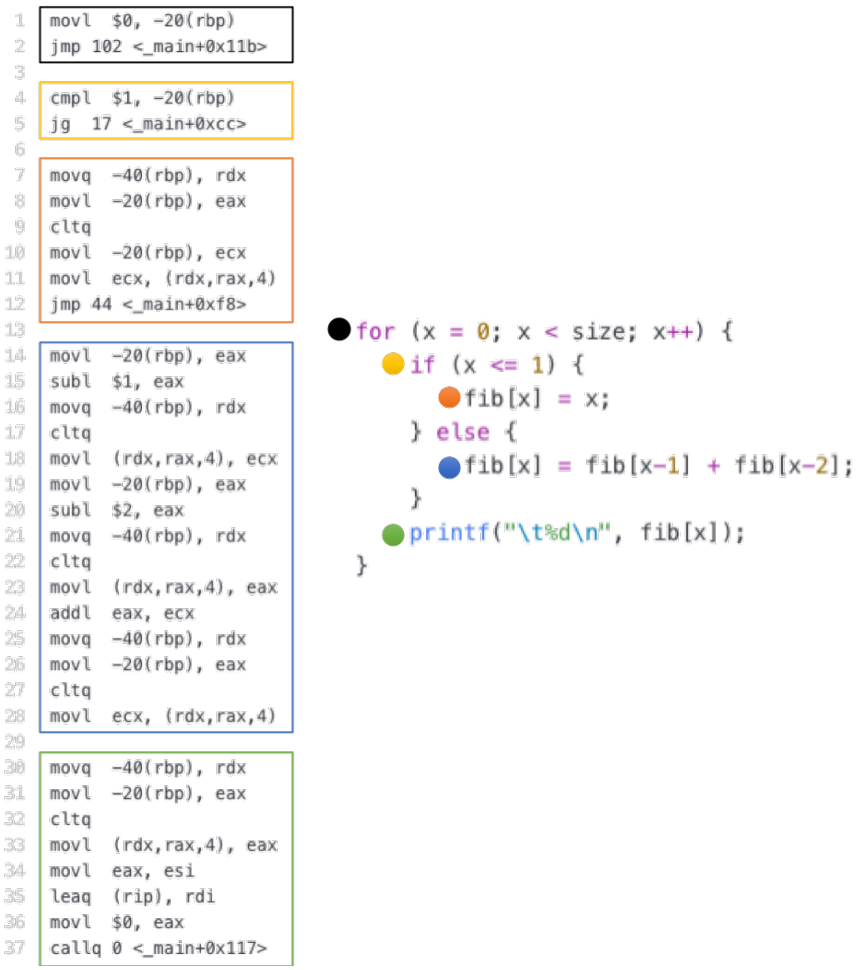


Figure 4.7. Assembly vs. C-Code Fibonacci example snippet

To demonstrate how nontrivial the extraction of this feature rapidly becomes we extracted the total real (i.e, non-stub) functions for 47K compiled binaries using Ghidra. Each binary is a synthetic test case from either the CB-Multios corpus [141] or Juliet C corpus [26]. Each sample was compiled using GCC on an Ubuntu version 21.04 virtual machine. Figure

4.8 depicts the total number of real functions in all samples by their associated CWE pillar, capturing a total of 5,123,148 functions. Figure 4.9 describes the total function count in terms of their McCabe cyclomatic complexity. Manual identification of the function-level cyclomatic complexity for compiled code is clearly insurmountable.

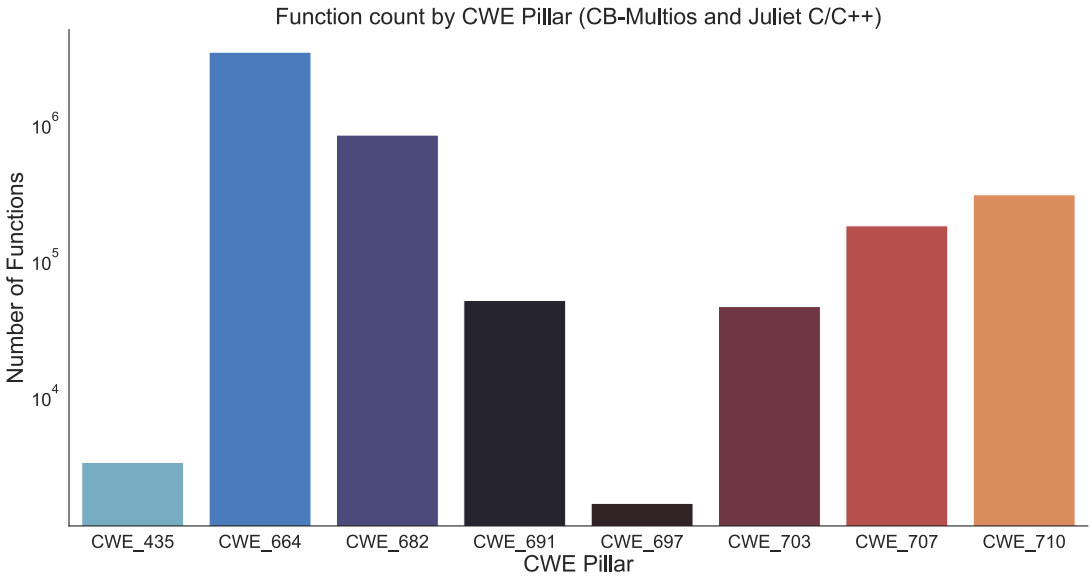


Figure 4.8. Total function count by CWE Pillar

Cyclomatic Complexity Vulnerability Research

A number of researchers have explored the correlation between cyclomatic complexity and software security. The below timeline highlights some of this work. A common theme in these works is that higher complexity levels may be associated with security risks such as vulnerabilities. In 2008, Yonghee Shin and Laurie Williams explore whether cyclomatic complexity can be used to predict vulnerabilities [218]. The authors build upon their previous study in 2008, when they examine the correlation of various complexity metrics to known vulnerable components from the JavaScript Engine in the Mozilla application framework [219]. The work is continued in 2011 when Shin, et. al. discuss the utility of complexity, code churn, and developer activity as predictors of vulnerable code locations [92];

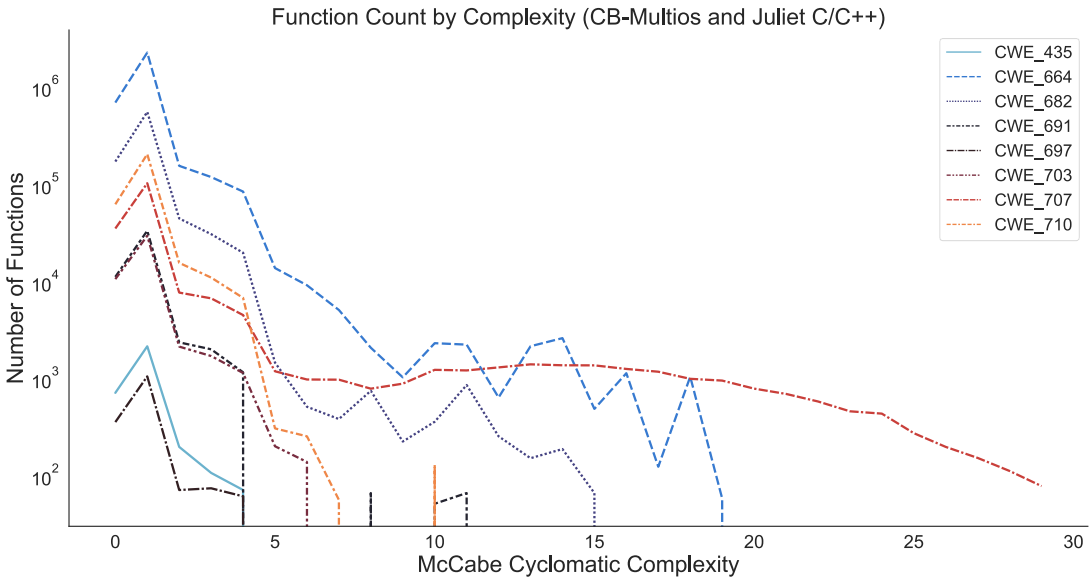


Figure 4.9. Total function count by cyclomatic complexity

and again in 2011, when Shin and Williams examine complexity metrics specifically collected during program execution [91].

Other notable work includes the following: a 2011 study on the use of complexity and other software metrics as early indicators of vulnerabilities [220], a 2015 study on a method to improve the effectiveness of fuzzing techniques using computational complexity [221], and a framework called LEOPARD, proposed in 2020, to identify potentially vulnerable functions using metrics such as complexity [181].

Each of these studies explores the efficacy of using cyclomatic complexity to enhance various aspects of vulnerability research. With the exception of the LEOPARD framework in [181], none are accompanied by a tool to support other researchers or reproduce findings. The LEOPARD framework operates on source code. LEOPARD first uses complexity metrics to bin program functions, then uses other vulnerability metrics to prioritize functions for further analysis based on how potentially vulnerable they are. `BiSECT` differs from LEOPARD in two primary ways: first, `BiSECT` extracts features from binary files. Second, the final result of

LEOPARD is a list of functions that has been prioritized based on their associated potential vulnerabilities. The final result from `BiSECT` is a dataset of features that has been curated to support data mining and machine learning algorithms.

TIMELINE 1: *Cyclomatic Complexity Related Vulnerability Research*

- 2008 — *An empirical model to predict security vulnerabilities using code complexity metrics*, [218]
- 2008 — *Is Complexity Really the Enemy of Software Security?* [219]
- 2011 — *Using complexity, coupling, and cohesion metrics as early indicators of vuln.* [220]
- 2011 — *Evaluating Complexity [...] Metrics as Indicators of Software. vuln.*, [92]
- 2011 — *[...] on the use of execution complexity metrics as indicators of software vuln.* [91]
- 2013 — *Using complexity metrics to improve software security* [93]
- 2015 — *Improving Fuzzing Using Software Complexity Metrics* [221]
- 2020 — *LEOPARD: Identifying Vulnerable Code [...] through Program Metrics* [181]

4.2.4 Feature Example: N-Grams

An *N*-**gram** is a contiguous sequence of items of length *N* in a given sample [223]. As an example, consider the following sentence:

> The quick brown fox jumps over the lazy dog

This sentence can be broken up into many different 'grams' depending on the level of granularity, Table 4.4 provides a few examples of n-grams of a few word and letter grams of different lengths.

In the English language the granularity of *N*-grams can include characters, words, sentences, and even clauses. Once the sample has been partitioned at the chosen level, it is said to have been **tokenized** [223]. An *N*-**gram token** is simply a single unit or gram from a tokenized sample. As an example, in Table 4.4, the first token in the word-level 3-grams (i.e., *trigram*) is "the quick brown." In this way the term *N-gram token* is somewhat similar to a **lexical token**, or a single unit defined in the specification of a programming language [89]. There

Table 4.4. Example n-grams of varying granularity levels

Granularity	**Size (N)**	**Result**
Word	3	the quick brown | quick brown fox | brown fox jumps | fox jumps over | jumps over the | over the lazy | the lazy dog
Word	4	the quick brown fox | quick brown fox jumps | brown fox jumps over | fox jumps over the | jumps over the lazy | over the lazy dog
Letter	5	t h e _ q | h e _ q u | e _ q u i |_ q u i c | q u i c k | u i c k _ | i c k _ b | c k _ b r

are five types of lexical tokens (at the source code level) in most programming languages: keywords, identifiers, operators, separators, and literals. We can use these lexical tokens as logical starting place to determining appropriate N-gram tokens in a binary sample.

N-Grams Extraction

Similar to computational complexity, to extract N-grams from compiled code the binary must first be disassembled[16]. N-grams from the disassembled binary samples will be in the form of assembly code. Thus, assembly keywords may include things such as instructions, mnemonics or opcodes (e.g., MOV, XOR, ADD); literals may include string constants (e.g., "hello", "all your N-grams are belong to me", etc.).

Consider again the Fibonacci example discussed in the previous section— Figure 4.7 provides a subset of the assembly code for the sample. Figure 4.10 outlines the first 20 mnemonic N-grams, where N = 2, 3, or 4, extracted from the full sample. In this short sample a total of 86 2-grams (i.e., bigrams), 87 3-grams (i.e., trigrams), and 88 4-grams can be extracted.

To demonstrate how nontrivial the extraction of this feature rapidly becomes, we extracted all 2, 3, and 4 mnemonic-grams from 47K compiled binaries using Ghidra. For consistency, these are the same samples used in Cyclomatic Complexity, Section 4.2.3. Figure 4.11 shows

[16] N-grams could also be discovered by writing a custom parser

	Two Grams	Three Grams	Four Grams
0	(PUSH, MOV)	(PUSH, MOV, SUB)	[PUSH, MOV, SUB, MOV]
1	(MOV, SUB)	(MOV, SUB, MOV)	[MOV, SUB, MOV, MOV]
2	(SUB, MOV)	(SUB, MOV, MOV)	[SUB, MOV, MOV, MOV]
3	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
4	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
5	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
6	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
7	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
8	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
9	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
10	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, MOV]
11	(MOV, MOV)	(MOV, MOV, MOV)	[MOV, MOV, MOV, LEA]
12	(MOV, MOV)	(MOV, MOV, LEA)	[MOV, MOV, LEA, AND]
13	(MOV, LEA)	(MOV, LEA, AND)	[MOV, LEA, AND, MOV]
14	(LEA, AND)	(LEA, AND, MOV)	[LEA, AND, MOV, MOV]
15	(AND, MOV)	(AND, MOV, MOV)	[AND, MOV, MOV, CALL]
16	(MOV, MOV)	(MOV, MOV, CALL)	[MOV, MOV, CALL, SUB]
17	(MOV, CALL)	(MOV, CALL, SUB)	[MOV, CALL, SUB, MOV]
18	(CALL, SUB)	(CALL, SUB, MOV)	[CALL, SUB, MOV, MOV]
19	(SUB, MOV)	(SUB, MOV, MOV)	[SUB, MOV, MOV, MOV]

Figure 4.10. First 20 2,3,4-grams from the full Fibonacci.c example

the total number of 2,3 and 4-grams in the 47K samples by associated CWE pillar, capturing a total of 2.5 billion N-grams. Like the function-level McCabe cyclomatic complexity, identification of N-grams in binary code requires domain expertise, and manual efforts do not scale to the number of grams possible in any given sample.

N-Grams Vulnerability Research

The concept of an N-gram was first introduced in 1948, by C. E. Shannon, in *A Mathematical Theory of Communication* [224]. Shannon used preceding N-grams to predict the next letter or word in a text [224]. Since the mid-1900's N-grams have been used to support a variety of research areas including: natural language processing (e.g., speech recognition, machine translation, and predictive text input), communication theory, computational linguistics,

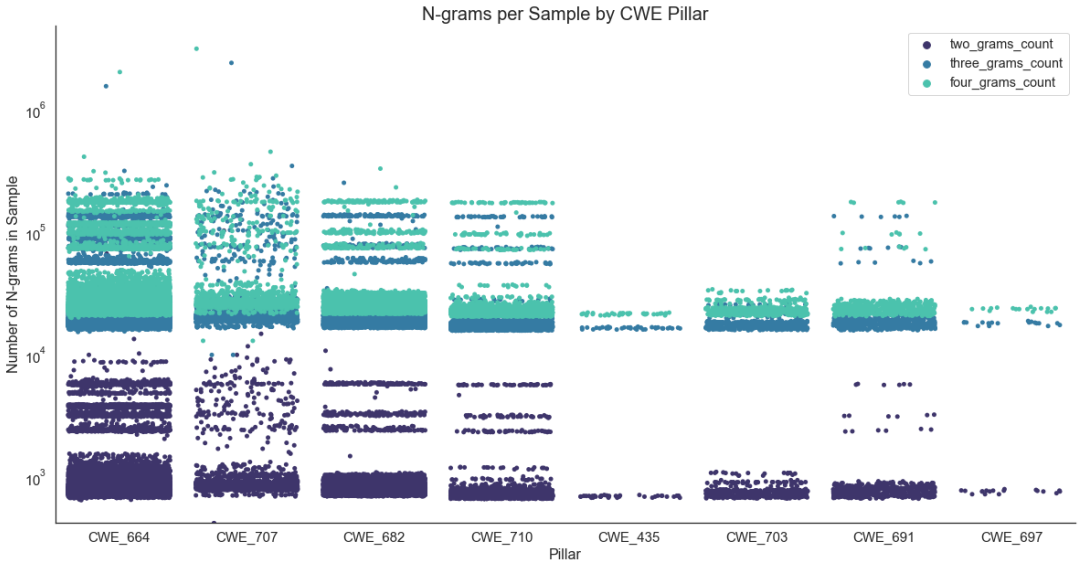


Figure 4.11. Count of N-grams in 47K samples by associated CWE Pillar

computational biology, and data compression. In the system security domain N-grams have been used to identify malicious code, determine malware authorship, and to detect spam [225]–[230].

In the last decade N-grams have also been used to facilitate vulnerability research. While not specifically examining vulnerabilities[17], [212] presents a technique to use N-grams and rank event handlers for faults in graphical user interfaces. In 2014, Scandariato et al. [110] used a text mining bag-of-words approach leveraging N-grams to predict which components of source code are likely to be vulnerable. Also in 2014, Walden et al. [33] examined the strengths and weaknesses of using text mining and software metrics such as N-grams as vulnerability predictors. Pang et al. [113] used a combination of N-grams and feature selection algorithms to predict vulnerable components in Java class files.

[17]See Chapter 1 for a description of the differences between faults and software vulnerabilities

TIMELINE 2: *Vulnerability research using N-grams*

2011 *GUI Software Fault Localization Using N-gram Analysis*, [212]
2014 *Predicting Vuln. Sw. Components via Text Mining*, [110]
2014 *Predicting Vuln. Components: Sw. Metrics vs Text Mining*, [33]
2015 *Predicting Vuln. Sw. Components through N-Gram Analysis [...]*, [113]
2016 *Toward Large-Scale Vulnerability Discovery using Machine Learning*, [40]

4.3 Data Cleaning and Transformation

After each feature has been extracted the data must be cleaned, consolidated, and transformed into a format that is compatible with machine learning algorithms and data mining techniques. The cleaning and transformation processes may involve the following steps [37]:

1. **Missing Values** missing or incomplete values are removed, ignored, or replaced
2. **Smoothing** noise and outliers removed from data using binning, regression, or other outlier analysis (e.g., clustering)
3. **Numerics** non-numeric data is transformed into a numeric representation
4. **Aggregation** summary and aggregation operations are applied to the data
5. **Normalization/Scaling** features are normalized or scaled to fall within a smaller range (e.g., 0.0 to 1.0)
6. **Discretization** raw values of numeric data are binned by being transformed into interval ranges or conceptual labels
7. **Feature Construction** new features are constructed from existing features

In Section 4.2 we provided an overview of the primary features directly extracted from binary files by `BiSECT`. In this section we explore two additional functionalities provided by `BiSECT`: data cleaning, and transformations. Figure 4.12 indicates where these steps reside in the overall `BiSECT` workflow.

As indicated in Figure 4.12, some steps, such as cleaning missing values or smoothing data to remove outliers can be completed regardless of the features selected for analysis. Other steps, such as normalizing or aggregating data, and constructing new features may or may not be completed depending on the initial feature set and the user's end goals.

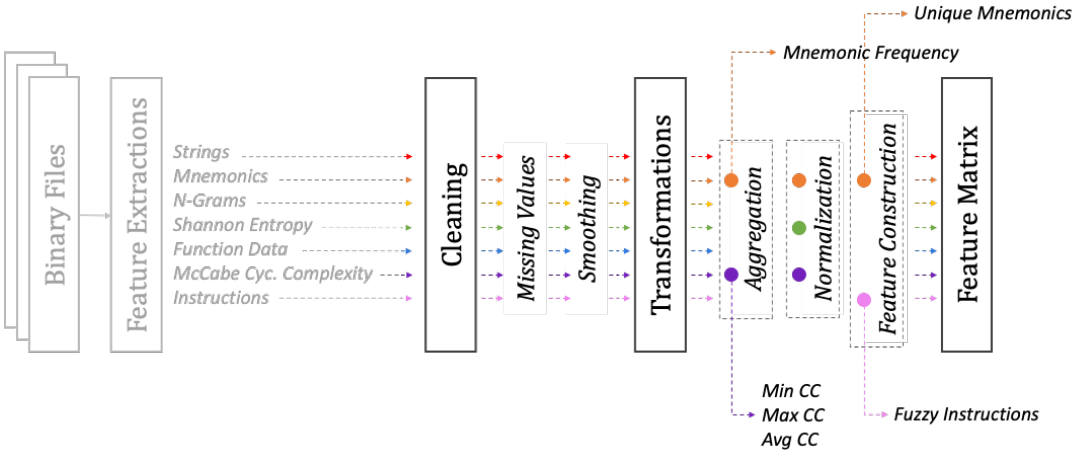


Figure 4.12. `BiSECT` data cleaning and transformation steps

In the following sections we will describe the process to clean and transform features using `BiSECT`.

4.3.1 Data Cleaning

When using real-world data objects, such as binary files, some files may have missing, inconsistent, or incomplete attributes. Data cleaning typically involves two steps, dealing with missing values and smoothing outliers. Missing or incomplete values must first be identified, then either replaced, dropped, or ignored. It is important to note that missing values may not be the result of an error. For example, consider a feature, *strings*, that specifies the set of all unique strings in a binary sample. If a sample has no strings, this feature will appear to be missing for that data object.

Missing Values

During the **missing values** step, the user may choose to ignore the missing values, perform a manual replacement, replace missing values with a global value, such as 'unknown' or 'null', or replace them with some other statistically determined measurement, e.g., a measure of central tendency, or the most probable value for that feature [37]. Each of these alternatives has drawbacks. For example, replacing missing values with another statistically determined value introduces bias into the dataset. This occurs because the replacement value may not

be accurate. While manually replacing missing values may be accurate, it can also be time consuming. Replacing values with a global variable is simplistic; however, the analyst must be careful not to perceive connections between samples with the same replaced value.

For simplicity, `BiSECT` provides three options to the user: drop all data objects (i.e., samples) that contain a missing value for any feature, replace all missing values with a user-provided replacement value (with a default of 0), and ignore the missing values. The user should select the option that is best suited to their intended use case.

Smoothing

After handling missing values, the next step is to **smooth** the dataset by reducing noise. **Noise** essentially equates to variance in a measured variable [37]. Conventional methods to reduce noise include binning, regression, and outlier analysis.

In **binning**, the objects are grouped into *bins* typically with neighboring objects, and smoothed by replacing each value with the mean, median, or boundary for that bin. When reducing noise using **regression** (e.g., via linear or multi-linear regression), the values are replaced by a line of best fit. Noise in a dataset can also be reduced using various forms of **outlier** analysis. Clustering, for example, can be used to quickly determine which data objects fall outside of the primary cluster or clusters.

Since smoothing may or may not be appropriate, `BiSECT` does not perform any smoothing by default. Additionally, standard smoothing techniques are not applicable to features that are comprised of a set of text data (e.g., a set of unique strings in the sample). `BiSECT` provides an optional function, $smooth()$, that uses outlier analysis to perform smoothing. Users may apply $smooth()$ to individual features, such as cyclomatic complexity, as appropriate for their use case. An example of the cyclomatic complexity distribution before and after smoothing is provided in Figure 4.13.

4.3.2 Data Transformation

After the dataset has been extracted and cleaned, the next steps involve performing one or more transformations of the data. In a typical machine learning or data mining workflow feature transformations include the aggregation and or normalization of features, and the construction of new features.

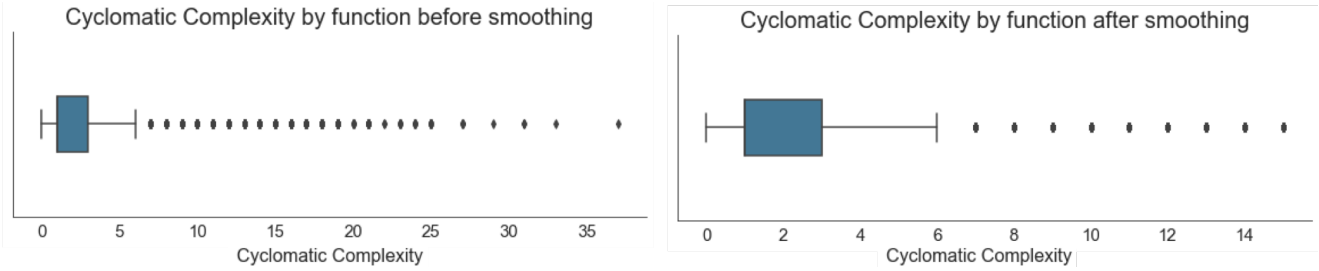


Figure 4.13. Cyclomatic complexity distribution before (left) and after (right) applying the `BiSECT` *smooth*() function

Aggregation

During the **aggregation** step, summary and aggregation operations are applied to the data to create new features. `BiSECT` can be used to extract the McCabe cyclomatic complexity for every function in every sample analyzed. Out of the box, `BiSECT` provides three aggregation features related to cyclomatic complexity: minimum, maximum, and average cyclomatic complexity. Each of these features are provided at the file level; Figure 4.14 provides an example outcome of the application of the *aggregation*() function. Aggregating the cyclomatic complexity in this manner could provide quantitative data to answer questions such as, what is the maximum or minimum cyclomatic complexity for every file analyzed? Or perhaps more insightful, what is the distribution of maximum cyclomatic complexity across all samples analyzed, and does it differ for known vulnerable versus known benign samples?

Normalization

During **normalization**, features are scaled to fit within a standardized or smaller range. Normalizing the feature values forces all features to have an equal weight, and can prevent numeric features with large ranges (e.g., entropy) from outweighing features with small ranges (e.g., cyclomatic complexity). Treating all features equally, can also reduce the domain expertise required to distinguish the importance of one feature from another. Common normalization techniques include scaling (e.g., min-max, linear, or log scaling), clipping, and using a z-score.

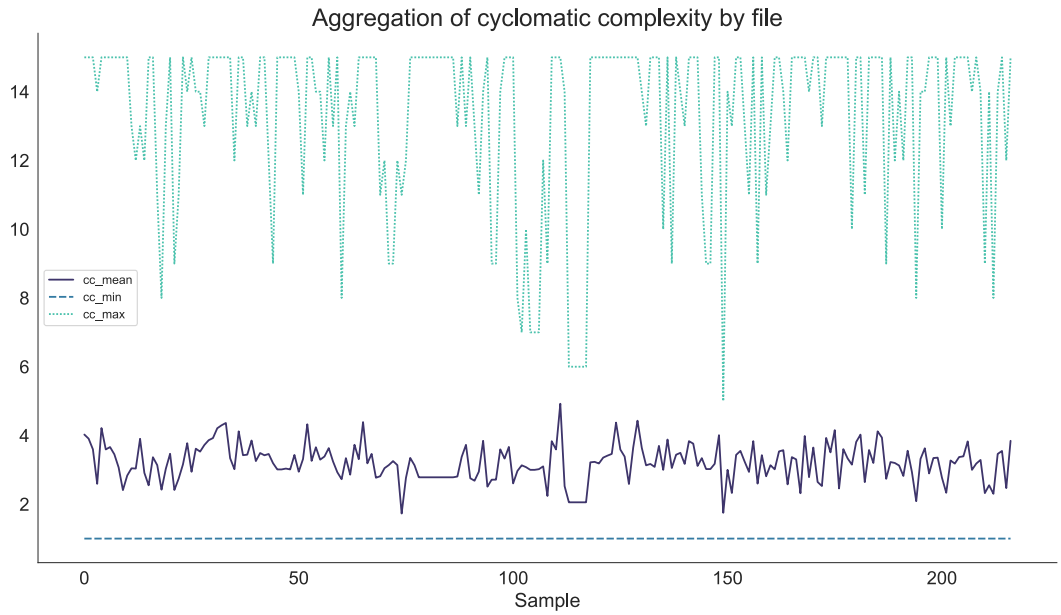


Figure 4.14. Example outcome of `BiSECT` *aggregation*() function

`BiSECT` provides the user with the option of normalizing features using the function, *normalize*(). *normalize*() applies min-max scaling using the following formula,

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{4.2}$$

Where x is the original value, and x' is the normalized value.

Feature Construction

During the **feature construction** phase, new features are created from existing features. The most noteworthy feature constructed by `BiSECT` (specifically for X86 binaries) is *fuzzyInstruction*. By default, `BiSECT` extracts the function sequence for each disassembled function in a binary sample. To prepare the instruction sequences to be compatible with machine learning algorithms, we perform a number of transformations on the original feature.

First, we tokenize each instruction sequence into individual *words*, i.e., operands, mnemonics, etc. Then, convert all characters to lowercase, and remove all punctuation and assembly size qualifiers such as $dword$ and $byte$. Finally, a number of standardizations are performed. For example, each memory address is converted to *addr*, and all remaining numeric constants are converted to *num*. The resulting instruction sequence loosely resembles X86, and thus we refer to these as $fuzzyInstruction$ sequences. For example, the following transformation would occur:

"SUB RSP,0x10" —> "sub rsp num"

An example of the process to construct the $fuzzyInstruction$ feature is provide in Section 4.5.2.

4.3.3 Feature Extension and General Workflow

We designed `BiSECT` with extensibility in mind. All features extracted by `BiSECT` are written to CSV files. We then leverage the Python Pandas library and Jupyter to perform many of the core cleaning and transformation steps offered by `BiSECT`. In this regard `BiSECT` is only limited by features that can be output to a Pandas-compatible format (e.g., CSV, JSON, XLSX, etc.). Finally, each of the individuals are combined into a single feature matrix, or dataset. The final matrix completes the `BiSECT` workflow as depicted in Figure 4.2.

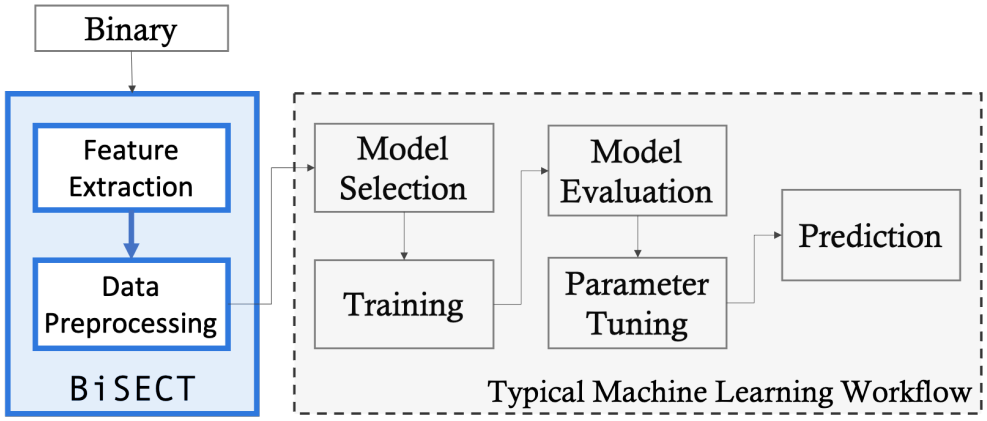


Figure 4.15. `BiSECT` as a precursor to the typical machine learning workflow

BiSECT has a number of use cases, however, each feature was specifically selected for its applicability and potential for use in data mining and machine learning scenarios as described in Section 4.2. Figure 4.15 depicts BiSECT as a precursor to a typical machine learning workflow. Features are extracted and preprocessed using BiSECT, then the typical machine learning workflow resumes. In Section 4.5 we demonstrate the ease with which one or more of the features generated by BiSECT can be used to train a machine learning classifier.

4.3.4 BiSECT Limitations

BiSECT suffers from several limitations in practice. For many of the features extracted by BiSECT Ghidra is first used to disassemble and analyze the binary file. In these instances BiSECT will inherently adopt any of Ghidra's limitations, of which several have been documented [231]. Additionally, BiSECT uses static analysis to extract features, and thus, samples analyzed by BiSECT are not executed. BiSECT makes no attempt to decompress, decrypt, or deobfuscate the files it analyzes. When working with binaries of these types, the features extracted by BiSECT may be sparse, null, or even misleading. For example, the use of opaque predicates may result in the obfuscation of the program's semantics [232], [233]. Learning models trained using code that has not been obfuscated may have a difficult time properly predicting the labels for obfuscated code and vice versa.

4.4 Related Work

We partition the reviewed work into two categories: open source tools that can be used to extract, clean, and transform one or more features from binaries, and deep learning techniques to identify vulnerabilities in binaries. See Section 4.1.1 for a discussion on malware feature extraction, cleaning, and transformation tools.

4.4.1 Feature Extraction, Cleaning, and Transformation (FECT) Tools

We refer to a tool that can be used to *extract, clean, and transform* one or more features from a binary file in support of vulnerability analysis as a Feature Extraction, Cleaning, and Transformation (FECT) tool. For example, the proposed tool, BiSECT, is a FECT tool. However, with this criteria, the *Strings* utility is not a FECT tool as it does not clean or transform any feature (i.e., strings) that it extracts [234]. Since the proposed work is open

source, we specifically examine open source FECT tools. We were unable to identify any open source FECT tools that meet the specified criteria. However, we identified numerous tools that accomplish one or two of the steps (e.g., extraction, cleaning and transformation, extraction and transformation, etc.). For example, the *Strings* utility can be used to extract UNICODE or ASCII characters from files, and *Objdump* can be used to extract information about one or more object files [235]. We also identified a couple of tools that do not explicitly extract features from binary files, however they can be used to clean and transform assembly code into features. Each of these tools assumes that the binary has already been disassembled. For example, *Asm2Vec* can be used to convert assembly code into basic blocks [146]. *VDiscover* applies static and dynamic information to extract features from assembly code [40]. In *VDiscover*, the structure of the assembly is approximated using static analysis to extract calls made to the standard C library. Dynamic analysis is then used to extract the execution trace for the program including the arguments for the function calls and the final state of the process if applicable [40]. `BiSECT` differs from the related work in three key ways. First, `BiSECT` is the only tool identified that strictly meets the criteria for a FECT tool (extraction, cleaning and transformation). `BiSECT` leverages Ghidra to extract features from binary samples directly. Second, `BiSECT` extracts and transforms additional features not extracted by the reviewed work (e.g., function-level instruction sequences, function-level cyclomatic complexity, etc.). Finally, `BiSECT` is the *only* tool that synthesizes the extraction, cleaning, and transformation of numerous features that are commonly used in binary vulnerability research.

4.4.2 Vulnerability Identification Using Deep Learning

In Section 4.5 we demonstrate one use case for `BiSECT` by providing an example application. In this example, the output of `BiSECT` is used to train two deep learning classification models and label functions as either vulnerable or not vulnerable. The application of deep learning techniques to identify potential vulnerabilities in software has gained momentum in recent years. A comprehensive survey of notable work in this area was published in 2020, by Lin, et al. [28]. Of the techniques reviewed by Lin et. al., we identified three publications related to work presented here. Two of these leverage binary code and one leverages source code to support their work.

Li et al. [31] proposed the use of *code gadgets*, or a number of semantically related lines of source code. The code gadgets are transformed into vectors which are then used to predict vulnerabilities using Bidirectional LSTM (BLSTM), the authors specifically evaluated vulnerabilities related to stack-based buffer overflows (CWE-121). Ultimately, their design was published as a deep learning-based vulnerability detection system called, *VulDeePecker*. Both *VulDeePecker* and our work use a sequence of instructions as the basis for training a classifier, however, *VulDeePecker* uses source code and was designed specifically to identify buffer overflow vulnerabilities.

Lee et al. [30] proposed *Instruction2vec*, based on *Word2Vec* [236]. Similar to *Word2Vec*, *Instruction2vec* creates a vector representation of some input using deep learning. Like the proposed work *Instruction2vec* was designed to predict vulnerable functions in binary files using a labeled test suite to train a learning model. However, *Instruction2vec* was trained using only test cases from the Juliet dataset [142] that are related to stack-based buffer overflow (CWE-121) vulnerabilities. In contrast, our work uses all C and C++ test cases in the Juliet suite, and all test cases from the CB-Multios repository [141]— collectively, these test cases span over 50 CWE types. Finally, *Instruction2vec* is used to identify similarity between individual instructions, whereas our work utilizes the entire instruction sequence in each function to train the classifier.

Le et al. [44] proposed the *Maximal Divergence Sequential Auto-Encoder* which leverages the deep learning technique, Variational Auto-Encoders (VAE) with one-hot encoding to identify vulnerabilities in binary code. The authors represent each binary as a sequence of machine instructions, then apply VAE to encourage the samples to be maximally divergent. Similar to [30], *Maximal Divergence Sequential Auto-Encoder* is based on the vector representation for individual instructions, and therefore differs from our work for the same reasons.

In addition to the work included in the survey, there are a few additional works that use features extracted from binary files to create deep learning models. *PalmTree* is based on the BERT model [237], and provides a pre-trained assembly language model for generating instruction embeddings [238]. *Asm2Vec* is based on the *Doc2vec* [145] model, and uses control flow information (including functions, basic blocks, and control flow graphs) to provide an assembly code representation learning model [146]. We demonstrate that features extracted with `BiSECT` can be used directly with existing representation models to achieve

favorable results.

4.5 BiSECT to Support At-Scale Vulnerability Identification in Binaries

Recent advances in the application of deep learning to speech recognition and machine translation have demonstrated the high potential ability of such algorithms to understand semantic relationships in natural languages [137]. This success has motivated a number of researchers to explore the applicability of machine learning models to software security and vulnerability research. A survey of data mining and conventional machine learning techniques to support the identification and analysis of software vulnerabilities was published in 2017 [27]. In this survey the authors reviewed 39 works published between 2001 and 2015. In 2020, Lin et al. conducted a survey of 19 works, published between 2013 and 2019, that utilize deep learning and neural networks to support vulnerability research [28]. These efforts provide the basis for the following BiSECT example application, *using BiSECT to Support at Scale Vulnerability Identification in Binaries*.

4.5.1 Methodology

In the following section we present an overview of the proposed methodology for using BiSECT in conjunction with existing sentence classification techniques to identify potentially vulnerable functions in compiled code.

Overview

The basic hypothesis for this application is similar to that of *Word2Vec* [239] and *Doc2Vec* [240]— if the $fuzzyInstruction$ sequences for two functions have a similar context, the meaning of these two functions also has a high-level of similarity. In turn, this similarity can be used to identify potentially vulnerable functions, and ultimately to prioritize vulnerability analysis. We compare the efficacy of two classification models: *Doc2Vec* and *fastText*, when given the task of labeling functions as likely not vulnerable or potentially vulnerable. To accomplish this task we conduct two experiments: Experiment 1, which uses pure synthetic code, and Experiment 2, which uses hybrid-synthetic code. In these

experiments we use BiSECT to extract the $fuzzyInstruction$ sequence for roughly 5M disassembled functions from 64K compiled C \ C++ programs.

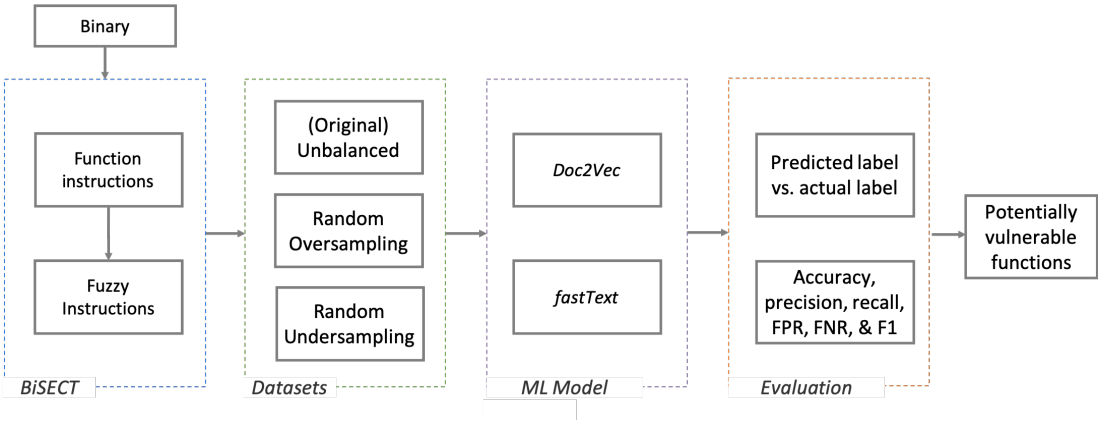


Figure 4.16. BiSECT Workflow in Example Application 1

The high level workflow for this example application is illustrated in Figure 4.16. First, BiSECT is used to extract the original instruction sequence from the compiled code (i.e., samples), which in turn is used to create the $fuzzyInstruction$ sequence— The result is an original dataset containing the $fuzzyInstruction$ sequence for all functions in all samples. We demonstrate that the original dataset suffers from class imbalance, and subsequently balance it using two different techniques: random oversampling and random undersampling. Each dataset is split into training (80%) and test (20%) sets. *Doc2Vec* and *fastText* are then used with the training sets to develop a vector representation for the $fuzzyInstruction$ sequences. These embeddings are ultimately used to classify whether the $fuzzyInstruction$ sequences in the test sets are [potentially] vulnerable or [likely] not vulnerable. We compare the actual labels with the predicted labels for each function in the test set to generate six evaluation metrics: accuracy, precision, recall, False Positive Rate (FPR), False Negative Rate (FNR), and F1. Finally, we examine the efficacy of the two classification methods using each dataset to identify potentially vulnerable functions.

Sentence Classification Techniques

In 2013 Mikolov et. al., introduced a novel Natural Language Processing (NLP) technique called, *Word2Vec*. *Word2Vec* uses a neural network to represent words in vector space, and ultimately to learn associations between individual words using a large corpus of text [239].

Word2Vec provides the basis for many sentence classification techniques including both *Doc2Vec* and *fastText*.

Mikolov et. al., expanded their word-embedding technique in 2014 to include variable-length texts. Their updated technique, *Doc2Vec* is based on the concept of a *Paragraph Vector*, that is, "an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts, such as sentences, paragraphs, and documents" [240].

The original *Doc2Vec* paper has been sited over 7,400 times, and the algorithm has recent success in cyber-relevant applications such as filtering malicious sourcecode [241], and the extraction of cyber threat-related information from public sources [242].

In *Doc2Vec*, individual word vectors, or in our case, mnemonics, contribute to the prediction task about the next word in the sentence, i.e., $fuzzyInstruction$ in a function. In this regard, the paragraph vectors in *Doc2Vec* inadvertently capture semantic information as they perform the prediction task [240]. Each paragraph and each individual word is mapped to a unique vector. For example, if there a N functions in the corpus, and M unique mnemonics in the vocabulary, our task is to learn the vector for each function mapped to p dimensions, and for each word mapped to q dimensions. The resulting model will have a total of $Nxp + Mxq$ parameters. Figure 4.17 describes the *Doc2Vec* framework using mnemonics as words, and $fuzzyInstruction$ sequences as paragraphs, where D refers to a *document*, and W refers to a *word*, which in our case equate to $fuzzyInstruction$ sequence and *mnemonics*.

Like *Doc2Vec*, *fastText* was also inspired by *Word2Vec*. *fastText* was developed and published by Facebook Research in 2016 [243]. *fastText* can also be used to learn word representations and perform text classification for variable-length texts. When using *fastText* the $fuzzyInstruction$ sequence for each function is represented as a bag of n-grams, which preserve information about the order of each mnemonic in the instruction set. The average of the mnemonic embeddings is taken, and subsequently the function representation is fed into a linear classifier which in turn determines the appropriate label for that function. The overall process is very similar to the *Doc2Vec* process outlined in Figure 4.17.

The primary differences between the two models are that a vector corresponding to a $fuzzyInstruction$ sequence in *fastText* is developed using a linear bag of words that is computed by averaging the mnemonic vectors for each mnemonic in the sequence. The

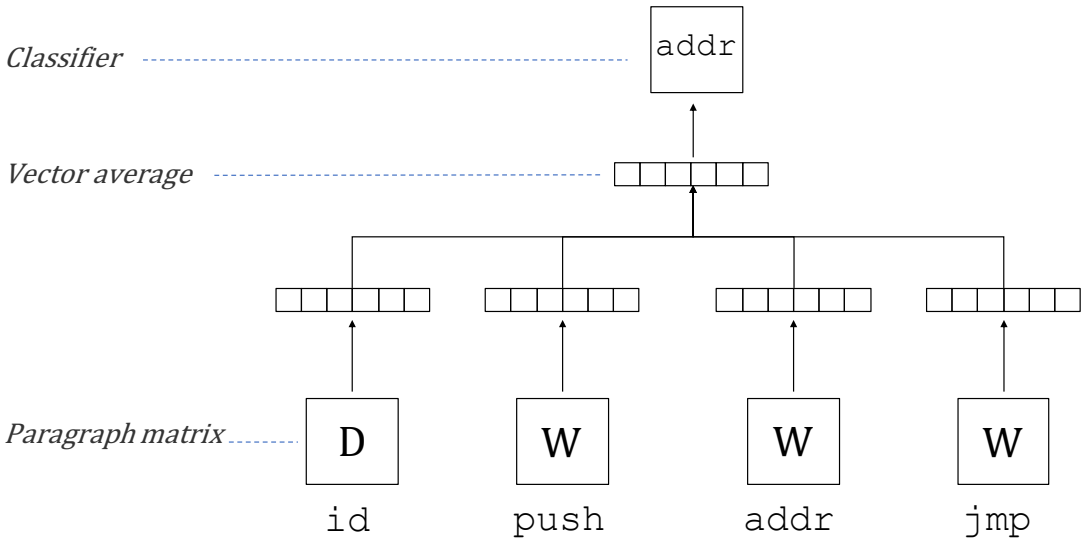


Figure 4.17. Framework for vectorizing $fuzzyInstruction$ sequences using *Doc2Vec*. Where D refers to a *document*, and W refers to a *word*, which in our case equate to $fuzzyInstruction$ sequence and *mnemonic*

vector for the *Doc2Vec* model is developed by concatenating the word and paragraph vectors for each paragraph (created using stochastic gradient descent with backpropogation [240]). Moreover, *Doc2Vec* feeds whole, individual words into a neural network, while *fastText* first decomposes each word into one or more N-grams. This technique provides *fastText* with the advantage that rare words, or at least some of their components will be represented by the model [243].

When using representation models like *fastText* and *Doc2Vec* each word is represented as a vector. The goal when using these models is that semantically similar words will have similar vector representations [240]. In both the *fastText* and *Word2Vec* representation models the vectors for individual mnemonics and operands contribute to the final vector for each $fuzzyInstruction$ sequence. The relationships between vectors can be plotted to reveal relationships between them. It is important to note that the vector representations are directly linked to the representation model— different models will result in different relationships. For example, Figure 4.18 shows a dendrogram of the relationships between mnemonics and operands learned using *Word2Vec*, while Figure 4.19 shows the resulting dendrogram using *fastText*'s skipgram algorithm. These models were trained using identical training data, but

with different underlying techniques. In both cases, the model resulted in close relationships between semantically related mnemonics like jnz (jump if not zero) and jz (jump if zero), cmp (compare) and sub (subtract).

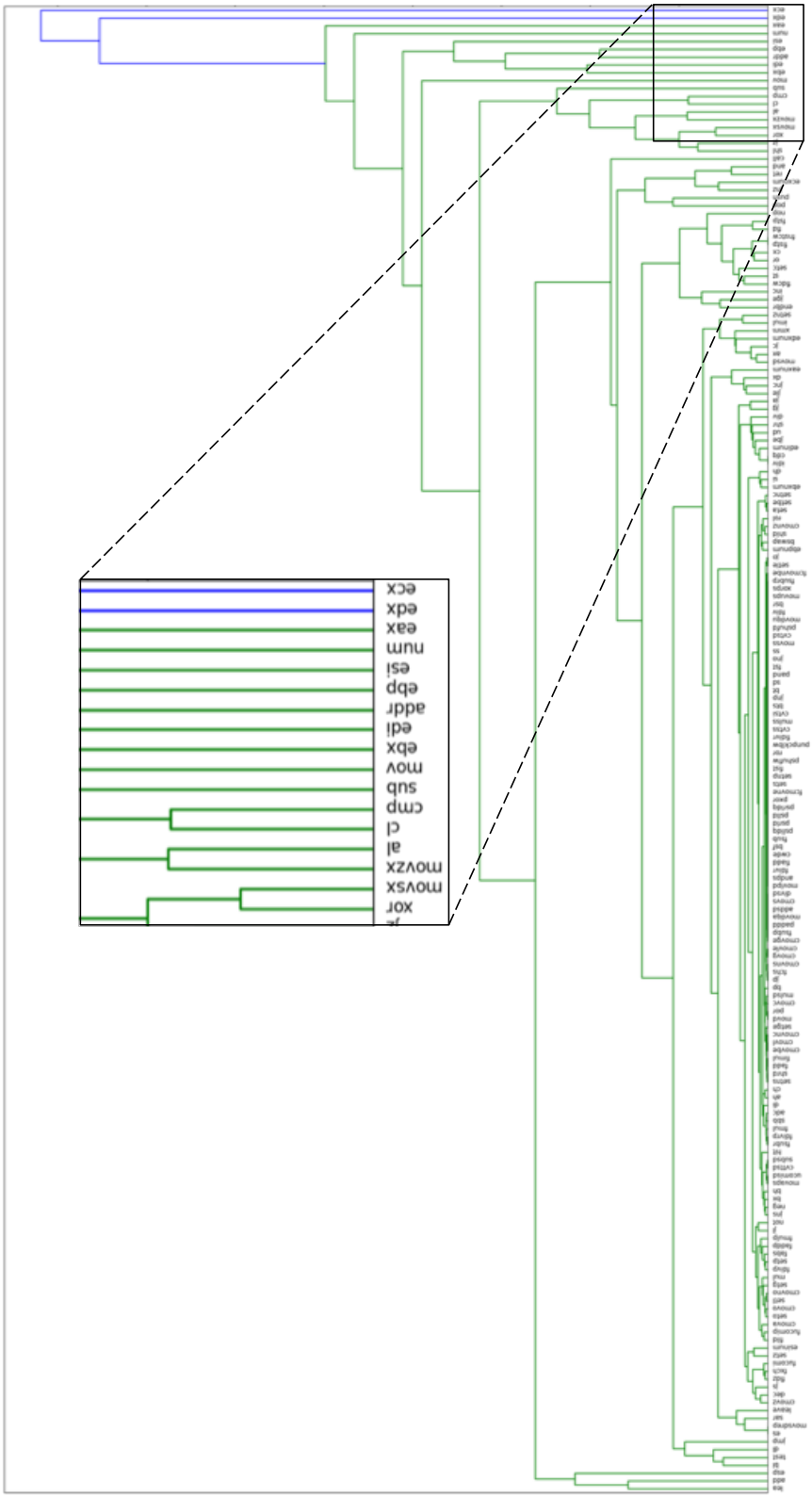


Figure 4.18. Dendrogram of x86 mnemonics embedded using *Word2Vec*

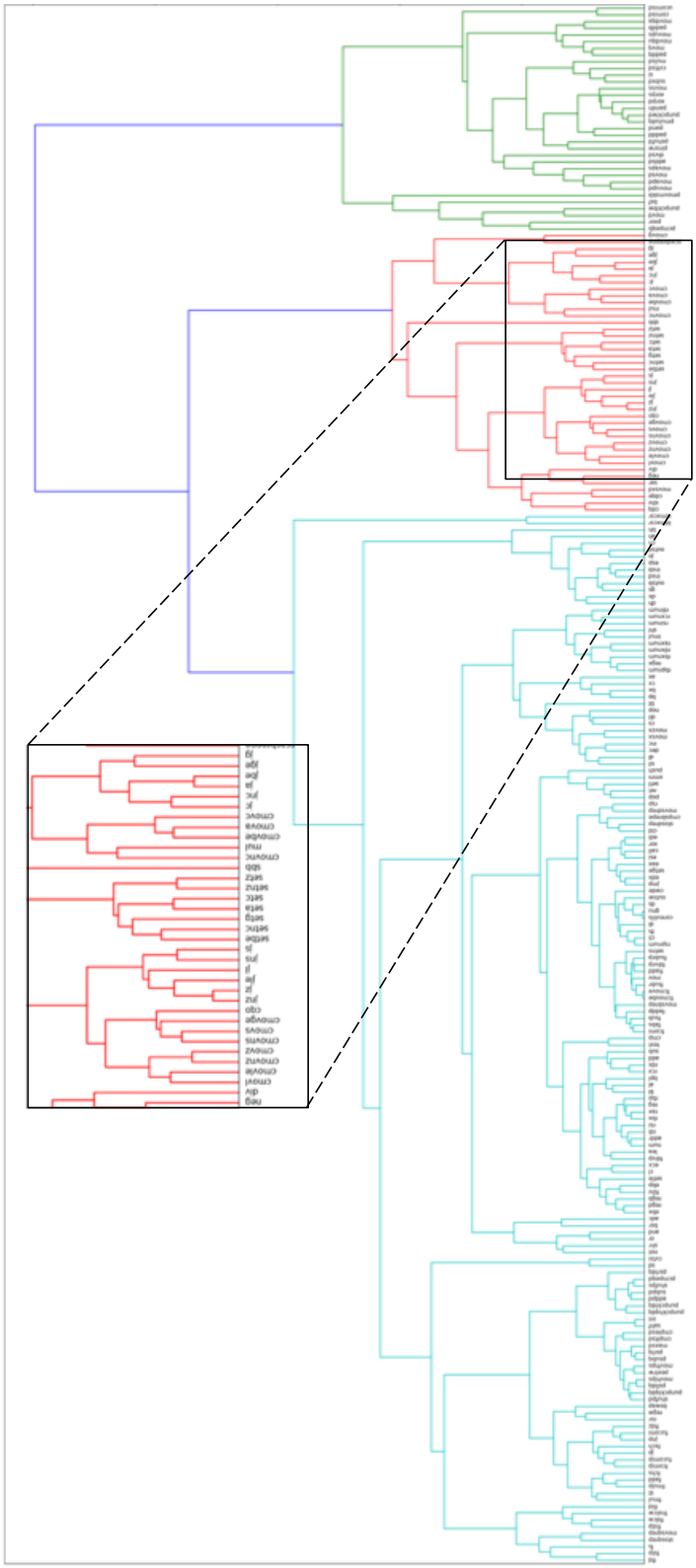


Figure 4.19. Dendrogram of x86 mnemonics embedded using the *fastText* skipgram algorithm

The vectors can also be used to create a scatter plot, where each node represents an individual mnemonic or operand. For example, Figure 4.20 depicts, as a scatter plot, a subset of x86 mnemonics and operands. The vector representation for each node was created using *fastText*. As expected, related mnemonics and operands such as those involved in floating-point or string instructions are grouped together. Additionally, mnemonics or operands related to logic, arithmetic, and control transfer instructions are tightly clustered.

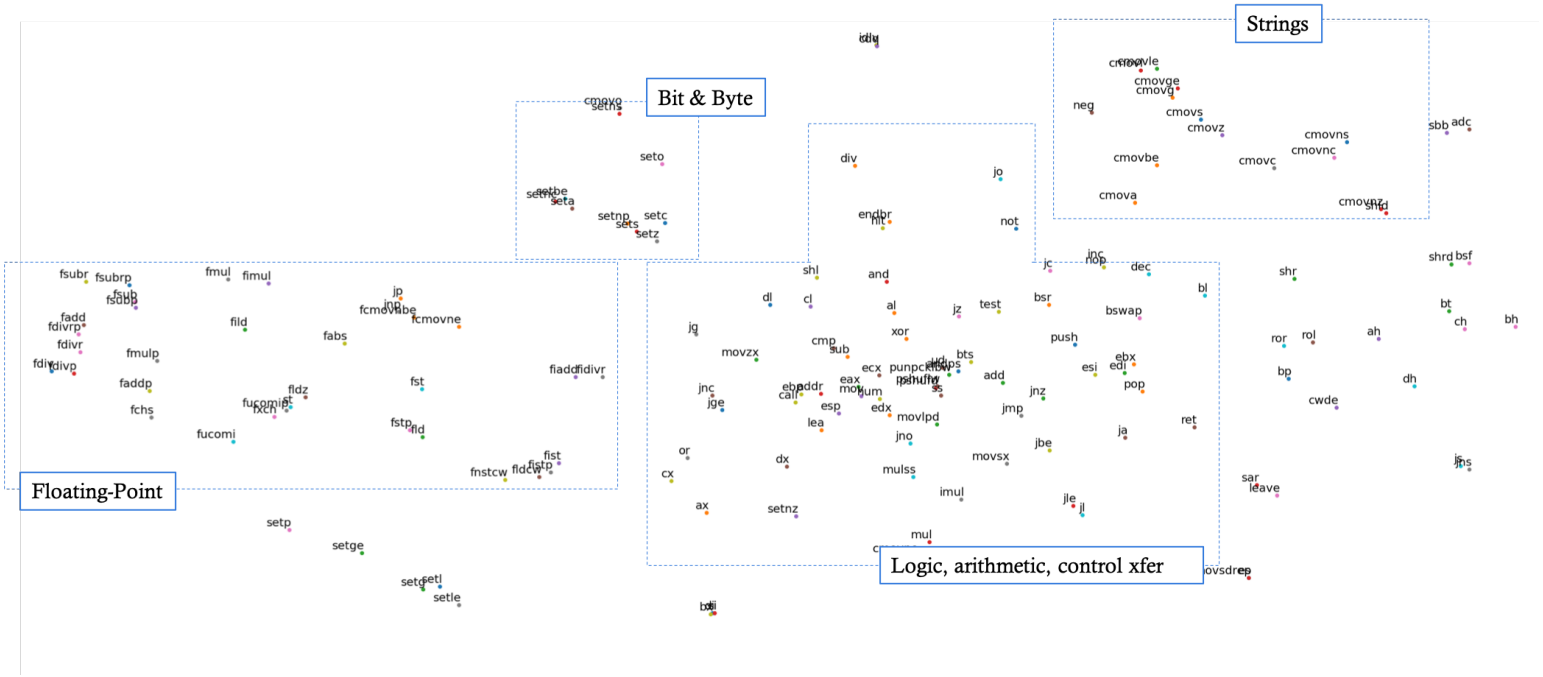


Figure 4.20. Scatterplot depicting x86 mnemonics embedded using *fastText*

Evaluation Metrics

To evaluate the performance of each model and corresponding dataset, we first created a confusion matrix for each test run. The confusion matrix can be used to display the *true positives* (tp), *false positives* (fp), *true negatives* (tn), and *false negatives* (fn). For example, the confusion matrix[18] for the *fastText* test using random oversampling is displayed as a heatmap in Figure 4.21. The larger the value, the darker the corresponding color in the heatmap, and vice versa. Throughout this work, known vulnerable functions are labeled *True*, all other functions are labeled *False*.

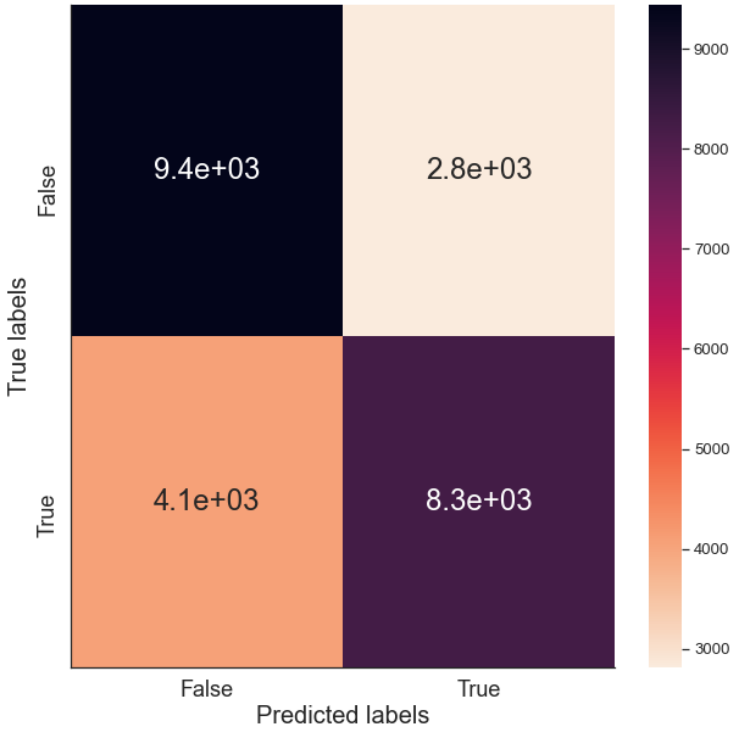


Figure 4.21. Confusion matrix (depicted as a heatmap) for the *fastText* model using Juliet corpus with random oversampling

[18]In the confusion matrices, *e* indicates that the number is written in scientific notation, e.g., 4.1e+03 is equal to 4,100.

Information included in the confusion matrix is subsequently used to quantify the accuracy, precision, recall, False Positive Rate (FPR), False Negative Rate (FNR), and F1-measure. These metrics are recorded for every test run in each experiment. **Accuracy** measures the number of correct predictions made (tp and tn). **Precision** provides a measure of correctness for the positive predictions, and **recall** measures the *true positive* rate, i.e., how many true vulnerabilities are predicted from the total vulnerabilities in the dataset. Finally, the *F1-measure* accounts for both precision and recall. The six evaluation metrics are calculated as follows,

$$\text{Accuracy} = \frac{tp + tn}{tp + fp + fn + tn} \tag{4.3}$$

$$\text{Precision} = \frac{tp}{tp + fp} \tag{4.4}$$

$$\text{Recall} = \frac{tp}{tp + fn} \tag{4.5}$$

$$\text{False Positive Rate (FPR)} = \frac{fp}{fp + tn} \tag{4.6}$$

$$\text{False Negative Rate (FNR)} = \frac{fn}{tp + fn} \tag{4.7}$$

$$\text{F1-measure} = \frac{2 * precision * recall}{precision + recall} \tag{4.8}$$

For reference, a *sound* technique would provide a False Negative Rate (FNR) of 0, and a *complete* technique would provide a False Positive Rate (FPR) of 0.

Class Imbalance

The imbalance of vulnerable to not vulnerable functions, known generally in machine learning as *class imbalance* presents a recurring challenge to vulnerability prediction models [27],

[137]. Imbalanced class data can be to the detriment of learning algorithms, as the learner learns to recognize the majority class, in our case, not vulnerable functions, more frequently than the minority class, i.e. , vulnerable functions [244].

There are a number of techniques to mitigate the class imbalance problem, most of which involve either *oversampling* or *undersampling* [244]–[246]. The underlying method in **oversampling** is often to replicate randomly selected samples from the minority class, whereas in **undersampling** samples are randomly removed from the majority class. Examples of oversampling techniques include simple random oversampling and SMOTE [247]; undersampling techniques include simple random undersampling, and undersampling using cluster centroids or Tomek link.

In this example application, we observe the impact to the learning algorithms when they are trained using the original unbalanced dataset verses balanced datasets. We create balanced datasets using simple random oversampling and random undersampling. In all cases we randomly split the data using 80% to train and 20% to test.

4.5.2 Experiment 1: Pure Synthetic Samples

In the first experiment, we used `BiSECT` to extract the $fuzzyInstruction$ for every function in an open source dataset of pure synthetic code samples containing known vulnerabilities, specifically, the Juliet C\C++ dataset. We then conducted a total of six tests using *fastText* and *Doc2Vec*. Three tests were conducted for each model, one for each of the following balanced or imbalanced version of the dataset:

1. Imbalanced— using original dataset
2. Balanced— using random undersampling
3. Balanced— using random oversampling

The overarching goal of this experiment was to assess whether *fastText* and/or *Doc2Vec* provide an effective method to identify potentially vulnerable functions in compiled binaries. We were also curious about the impact that class imbalance had on the classification models. We quantified the performance of each test using six evaluation metrics: accuracy, precision, recall, FPR, FNR, TPR, TNR, and F1-measure. The optimal overall performance was achieved using the *fastText* model in conjunction with the dataset balanced using random

oversampling.

Dataset

The Juliet C, C/C++ and Java test suites were developed by the National Security Agency (NSA) Center for Assured Software (CAS) to assess the capabilities and limitations of static analysis tools [142]. Each test case in the suite contains at least one known, synthetic, security flaw. The latest version of the Juliet test suites contain over 100K individual test cases, as outlined in Table 4.5.

Table 4.5. Juliet C#, C/C++ and Juliet Java descriptions

Dataset	Language	Total Programs	Label Granularity	Known Vulnerabilities
Juliet C#	C#	28,942	Function	28,942
Juliet C/C++	C/C++	64,099	Function	64,099
Juliet Java	Java	28,881	Function	28,881

The Juliet test suite was engineered to contain vulnerable functions; every intentional or known vulnerability in the corpus is labeled to specify the vulnerabilities associated CWE and whether that function is *good* or *bad*. Despite its limitations[19] the Juliet test suite contains a significant repository of labeled, known vulnerable (i.e., *bad*) and known benign (i.e., *good*) functions.

Fuzzy Instructions

Each test case in the Juliet C/C++ (v. 1.2) corpus was compiled using GCC (v. 9.3.0) on an Ubuntu 64-bit (v. 18.04.4) virtual machine. After compilation, we used `BiSECT`, which in turn leverages Ghidra, to extract a number of features from each compiled test case in the Juliet dataset. For this experiment we were particularly interested in the instruction set for each function, so that feature was isolated for further analysis.

To prepare the instruction sets to be compatible with the deep learning algorithms, *Doc2Vec* and *fastText* we performed a number of preprocessing steps using `BiSECT`. First, we tokenized the text into individual *words*, i.e., mnemonics and operands, and converted all characters to

[19] see Section 2.4.3

lowercase. We also removed all punctuation, and assembly size qualifiers such as $dword$ and $byte$. Finally, we performed a number of standardization's. For example, each memory addresses was converted to *addr*, each conditional and non conditional jump was converted to *jump*, and all remaining numeric constants were converted to *num*. The resulting instruction set loosely resembles x86, and thus we refer to the resulting feature as $fuzzyInstruction$ sequences. Figure 4.22 shows the format of the final feature matrix, presented as a Pandas dataframe.

	Sample Name	CWE Pillar	Fcn Name	CC	Vulnerable	Fuzzy Instructions
0	CWE121_Stack_Based_Buffer_Overflow__CWE129_con...	CWE_664	_CWE121_Stack_Based_Buffer_Overflow__CWE129_co...	10	True	push sbp mov rbp rsp sub num mov rax num mov r...
1	CWE121_Stack_Based_Buffer_Overflow__CWE129_con...	CWE_664	_CWE121_Stack_Based_Buffer_Overflow__CWE129_co...	1	False	push sbp mov rbp rsp call addr call addr pop s...
2	CWE121_Stack_Based_Buffer_Overflow__CWE129_con...	CWE_664	entry	1	False	push sbp mov rbp rsp sub num xor eax eax mov e...
3	CWE121_Stack_Based_Buffer_Overflow__CWE129_con...	CWE_664	_goodB2G	11	False	push sbp mov rbp rsp sub num mov rax num mov r...
4	CWE121_Stack_Based_Buffer_Overflow__CWE129_con...	CWE_664	_goodG2B	5	False	push sbp mov rbp rsp sub num mov rax num mov r...

Figure 4.22. Juliet features extracted using `BiSECT`

After preprocessing the data, we collected the $fuzzyInstruction$ sequences for roughly 4.8M functions. This included 4.3M that were not vulnerable (for this experiment we assumed that any function not explicitly labeled as *bad* was in fact *good*), and 451K that were vulnerable. The Juliet test suite contains a significant amount of duplicate code. We chose to remove all *identical functions*, that is, any function that had both a name and $fuzzyInstruction$ sequence that was identical to another function. After removing identical functions, 299K not vulnerable, and 61K vulnerable functions remained. The distribution before and after removing identical functions is described in Table 4.6.

Table 4.6. Function breakdown in Juliet C/C++ dataset

	Vulnerable	Not Vulnerable
Total Functions	451,982	4,344,057
Total Unique Functions	61,468	299,270

Class Imbalance

Even after removing duplicate functions, the ratio of vulnerable to not vulnerable functions was severely imbalanced, 83% not vulnerable to 17% vulnerable. We used the over and under sampling methods described in Section 4.5.1 to balance the dataset. Figure 4.23 depicts the distribution of vulnerable (True) to not vulnerable (False) functions in the original dataset, dataset using oversampling, and dataset using undersampling.

Evaluation Metrics

Table 4.7 describes the accuracy, precision, and recall metrics for each of the tests using *Doc2Vec* and *fastText* and the Juliet datasets. Using a balanced dataset provided an overall increase in accuracy, precision, and recall values. With both *fastText* and *Doc2Vec*, rebalancing the dataset using random oversampling provided the highest accuracy, precision, and recall values. The highest accuracy overall, 82%, was seen using *fastText* with the original unbalanced dataset.

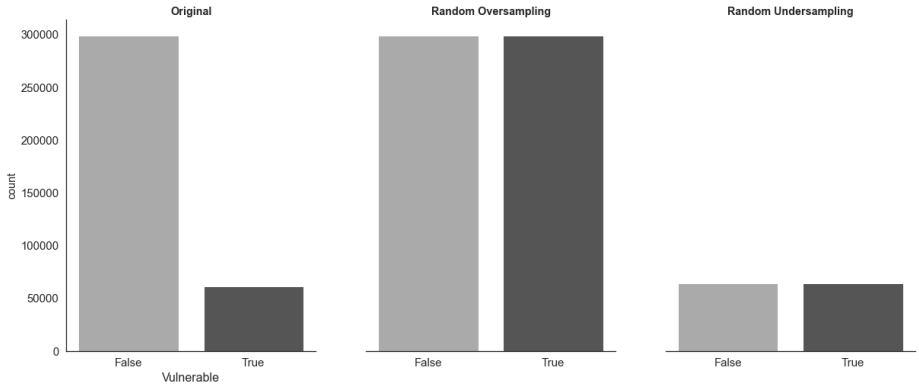


Figure 4.23. Distribution of samples in original and balanced Juliet C/C++ datasets

Table 4.7. Accuracy, precision, recall, false positive rate (FPR), false negative rate (FNR), and F1 metrics for the original and balanced Juliet datasets using *Doc2Vec* (deep learning) and *fastText* (linear classification)

Experiment	Accuracy	Precision	Recall	FPR	FNR	F1
Doc2Vec— Juliet						
Original (unbalanced)	0.6517	0.2417	0.4964	0.3167	0.5036	0.3251
Random Undersampling	0.6127	0.5924	0.7316	0.5070	0.2684	0.6547
Random Oversampling	0.7089	0.6838	0.7810	0.3637	0.2190	0.7292
fastText— Juliet						
Original (unbalanced)	0.8235	0.4667	0.3118	0.0725	0.6882	0.3739
Random Undersampling	0.7203	0.7463	0.6706	0.2296	0.3294	0.7064
Random Oversampling	0.7573	0.7844	0.7120	0.1970	0.2880	0.7465

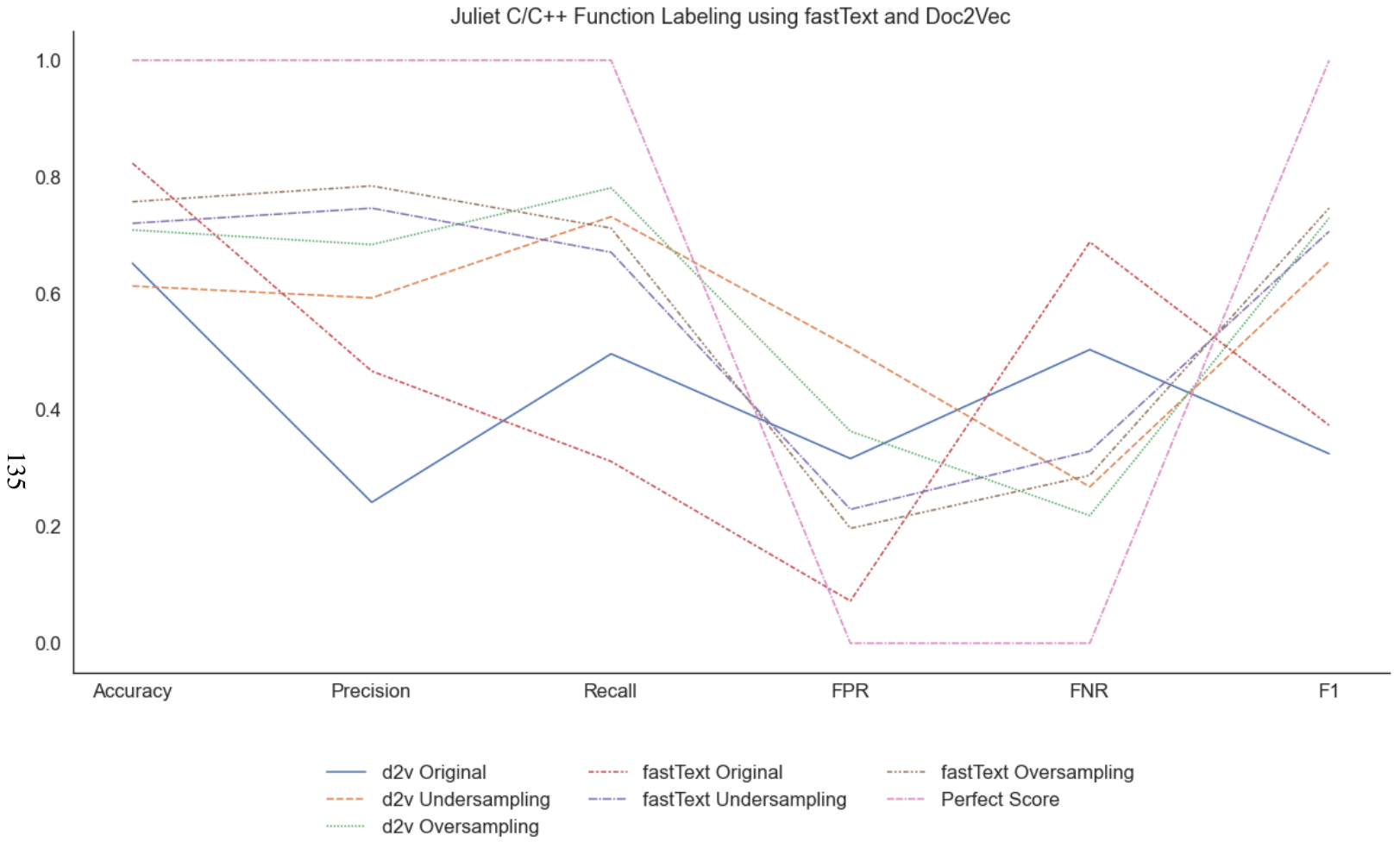


Figure 4.24. Evaluation metrics (Juliet with *Doc2Vec* and *fastText*). Metrics for each of the 6 experiments, by model (*Doc2Vec* or *fastText*), and dataset type (original unbalanced, and balanced with either random oversampling or undersampling)

The greatest performance overall was seen when using *fastText* or *Doc2Vec* with a dataset balanced using random oversampling. We were curious about whether there were observable differences in the performance metrics reported when examining specific types of vulnerabilities. To this end, we used *fastText* and*Doc2Vec* with a balanced (with random oversampling) dataset and assessed the performance metrics for test cases related to each of the 10 CWE Pillars[20]. Tables 4.9 and 4.8 provide the results from these tests. For reference, the description corresponding to each of the 10 CWE Pillars is provided in Table 4.10 [172].

Table 4.8. Performance metrics by CWE Pillar, *Doc2Vec* with random oversampling

CWE Pillar	Accuracy	Precision	Recall	FPR	FNR	F1
CWE 435	0.8855	0.0575	0.7667	0.1134	0.2333	0.1070
CWE 664	0.7030	0.6328	0.7721	0.3511	0.2279	0.6955
CWE 682	0.7167	0.5906	0.8052	0.3366	0.1948	0.6814
CWE 691	0.7336	0.5686	0.7834	0.2907	0.2166	0.6589
CWE 697	0.9252	0.0201	1.0000	0.0749	0.0000	0.0394
CWE 703	0.7835	0.6085	0.7983	0.2229	0.2017	0.6906
CWE 707	0.7112	0.5082	0.7852	0.3200	0.2148	0.6170
CWE 710	0.7001	0.5030	0.7727	0.3314	0.2273	0.6093

Table 4.9. Performance metrics by CWE Pillar, *fastText* with random oversampling

CWE Pillar	Accuracy	Precision	Recall	FPR	FNR	F1
CWE 435	0.9317	0.1158	1.0000	0.0689	0.0000	0.2076
CWE 664	0.7934	0.7705	0.7544	0.1760	0.2456	0.7623
CWE 682	0.8111	0.7302	0.7900	0.1761	0.2100	0.7589
CWE 691	0.8905	0.7899	0.9081	0.1181	0.0919	0.8449
CWE 697	0.9805	0.0729	1.0000	0.0195	0.0000	0.1358
CWE 703	0.9646	0.9712	0.9098	0.0117	0.0902	0.9395
CWE 707	0.8719	0.7279	0.9064	0.1427	0.0936	0.8074
CWE 710	0.8492	0.7031	0.8688	0.1593	0.1312	0.7772

[20]CWE Pillars were described in detail in Chapter 3

Table 4.10. CWE Pillar and descriptions (CWE Pillars 284 and 693 are not included in the Juliet C\C++ dataset, and thus, not included in this experiment)

CWE Pillar	Description
CWE 284	Improper Access Control
CWE 435	Improper Interaction Between Multiple Correctly-Behaving Entities
CWE 664	Improper Control of a Resource Through its Lifetime
CWE 682	Incorrect Calculation
CWE 691	Insufficient Control Flow Management
CWE 693	Protection Mechanism Failure
CWE 697	Incorrect Comparison
CWE 703	Improper Check or Handling of Exceptional Conditions
CWE 707	Improper Neutralization
CWE 710	Improper Adherence to Coding Standards

The greatest performance metrics overall were achieved when using *fastText* with samples related to CWE Pillar 703, *Improper Check or Handling of Exceptional Conditions*— 96.5% accuracy, 97.1% precision, and 91.0% recall, with a false positive rate of 1.1%, false negative rate of 9.0% and F1 of 94.0%. These results are encouraging and indicate that this may be an effective method to identify vulnerabilities related to the improper handling of exceptional conditions.

Process and Discussion

In this experiment, we used `BiSECT` to extract the $fuzzyInstruction$ sequence for every function in an open source repository of pure synthetic code samples containing known vulnerabilities, i.e., the Juliet C\C++ dataset. Then, we conducted six tests using two supervised learning models, *fastText* and *Doc2Vec* (three tests for each model):

1. Original unbalanced dataset
2. Dataset balanced using random undersampling
3. Dataset balanced using random oversampling

The overarching goal of this experiment was to assess whether *fastText* and/or *Doc2Vec* provided an effective method to identify potentially vulnerable functions in compiled binaries.

We quantified the performance of each test using six evaluation metrics: accuracy, precision, recall, FPR, FNR, TPR, TNR, and F1-measure. The optimal overall performance was achieved by using the *fastText* model in conjunction with the dataset balanced using random oversampling of the minority set.

These performance metrics seemed particularly promising for vulnerabilities related to CWE Pillar-703, *Improper Check or Handling of Exceptional Conditions*. In this case, functions in the dataset were labeled with 96.5% accuracy, 97.1% precision, and 91.0% recall, a false positive rate of 1.1%, false negative rate of 9.0%, and F1 of 94.0%.

We were encouraged to see how well this model performed using code outside of the Juliet suite. Unfortunately, when we attempted to use the same model to identify potentially vulnerable functions in another dataset, specifically, the CB-Multios dataset [141], the results were significantly less exceptional— on that dataset, functions were labeled with roughly 88% accuracy, but only 15% precision. We believe the disconnect in performance is directly correlated to the sophistication of the test cases in each repository. While every test case in both the Juliet and CB-Multios datasets was engineered to contain at least one known vulnerability, the CB-Multios test cases were designed to approximate real-world applications [140], while the Juliet cases were not [142]. The Juliet corpus has been criticized for being overly simplistic and redundant [30], [137] In this regard, we can think of the Juliet cases as *pure-synthetic* test cases, and the CB-Multios test cases as *hybrid-synthetic*. The result is that models trained using pure-synthetic, simplistic test cases were not able to precisely label functions from hybrid-synthetic or more complex test cases. Ultimately, these results led us to turn the experiment on its head and train new learning models using hybrid-synthetic test cases. The subsequent experiment, *Experiment 2*, is described next.

4.5.3 Experiment 2: Hybrid-synthetic

Experiment 2 was conducted following the same procedure specified in Experiment 1 with one primary difference, the CB-Multios corpus was used instead of the Juliet corpus. Unlike the Juliet corpus, each binary in the CB-Multios corpus was specifically designed to approximate real software, and includes at least one documented and exploitable vulnerability [140].

Dataset

At the conclusion of the CGC, DARPA released the entire corpus of challenge binaries to the public. Each binary was specifically designed to approximate real software, and includes at least one documented and exploitable vulnerability. The initial release of binaries was only compatible with the custom CGC DECREE operating system; researchers at TrailofBits modified and republished the binaries to be compatible with common operating environments including Windows, Macintosh, and Linux [141]. The TrailofBits repository of challenge binaries provides a relevant basis to demonstrate the data cleaning and transformation steps taken by BiSECT.

The author of each challenge binary provided a detailed description of the known vulnerabilities included in the sample, including an associated CWE classification. Using *gcc* on an Ubuntu virtual machine, we compiled 285 known vulnerable test cases or samples from the TrailofBits CB-Multios repository, and 285 patched (non-vulnerable) versions of the samples. We then used BiSECT to extract, clean, and transform the fuzzy instruction set for every function in all samples.

Unlike the Juliet corpus, which contains function level labels, known vulnerabilities in the CB-Multios corpus are labeled at the file level. However, users can manually query the source code for the presence of `ifdef PATCHED` which will reveal functions where known vulnerabilities reside. For example, the author of the challenge called, *BitBlaster*, indicates that the test case contains vulnerabilities related to CWE-284, *Access of Uninitialized Pointer* and CWE-476, *Null Pointer Dereference*. One method to label functions as vulnerable or not is to manually query the repository for the presence of `ifdef PATCHED`. We developed another, more automated method. This method was discussed extensively in Section 3.3.3. For the purpose of this experiment we label all 'unchanged' functions as 'not vulnerable'. Using the proposed method we identified and labeled 888 vulnerable functions, and 136,842 non-vulnerable functions in the CB-Multios corpus[21].

Class Imbalance

Like the Juliet dataset, the original CB-Multios dataset exhibits class imbalance, that is, an imbalance in the number of vulnerable and not vulnerable functions contained in the corpus.

[21] https://github.com/Kayla0x41/ghidraheadless$_b inexport$

The original dataset included 136,842 not vulnerable, and 888 vulnerable functions.

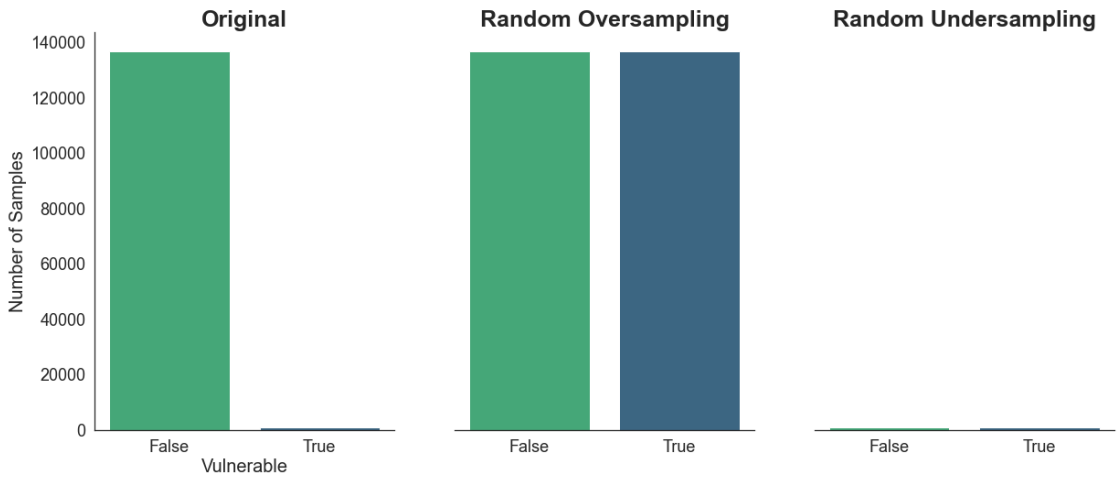


Figure 4.25. Distribution of samples in original and balanced CB-Multios datasets

As in Experiment 1, we used two rebalancing techniques to create a dataset with an equal number of vulnerable (True) and not vulnerable (*False*) functions— random oversampling of the minority class (i.e., vulnerable functions), and random undersampling of the majority class (i.e., not-vulnerable functions). Figure 4.25 depicts the distribution of function in the final dataset for each test run.

Evaluation Metrics

Each test was evaluated using the same six metrics that were used in Experiment 1- accuracy, precision, recall, FPR, FNR, TPR, TNR, and F1. Table 4.11 provides the evaluation metrics for each of the test runs.

The evaluation metrics for each test is also displayed graphically in Figure 4.26. In this graph, we have also included a "Perfect Score" for reference. If we follow the line for the "Perfect Score" in the graph, we can quickly observe that the greatest performance was consistently achieved via the *fastText* model with the dataset balanced using random oversampling. In this case, functions were identified with 96.4% accuracy, 97.8% precision, 94.8% recall, a FPR of 2.1%, FNR of 5.2% , and F1 of 96.3%. These results are consistent with the results

Table 4.11. Accuracy, precision, recall, false positive rate (FPR), false negative rate (FNR), and F1 metrics for the original and balanced CB-Multios datasets using *Doc2Vec* (deep learning) and *fastText* (linear classification)

Experiment	Accuracy	Precision	Recall	FPR	FNR	F1
Doc2Vec— CB-Multios						
Original (unbalanced)	0.9921	0.1077	0.0422	0.0021	0.9578	0.0606
Random Undersampling	0.8202	0.8647	0.7819	0.1369	0.2181	0.8212
Random Oversampling	0.7672	0.8345	0.6634	0.1301	0.3366	0.7392
fastText— CB-Multios						
Original (unbalanced)	0.9889	0.1911	0.2590	0.0066	0.7410	0.2199
Random Undersampling	0.4803	1.0000	0.0160	0.0000	0.9840	0.0314
Random Oversampling	0.9635	0.9778	0.9481	0.0213	0.0519	0.9627

achieved in Experiment 1, where the greatest performance was also achieved using *fastText* with random oversampling.

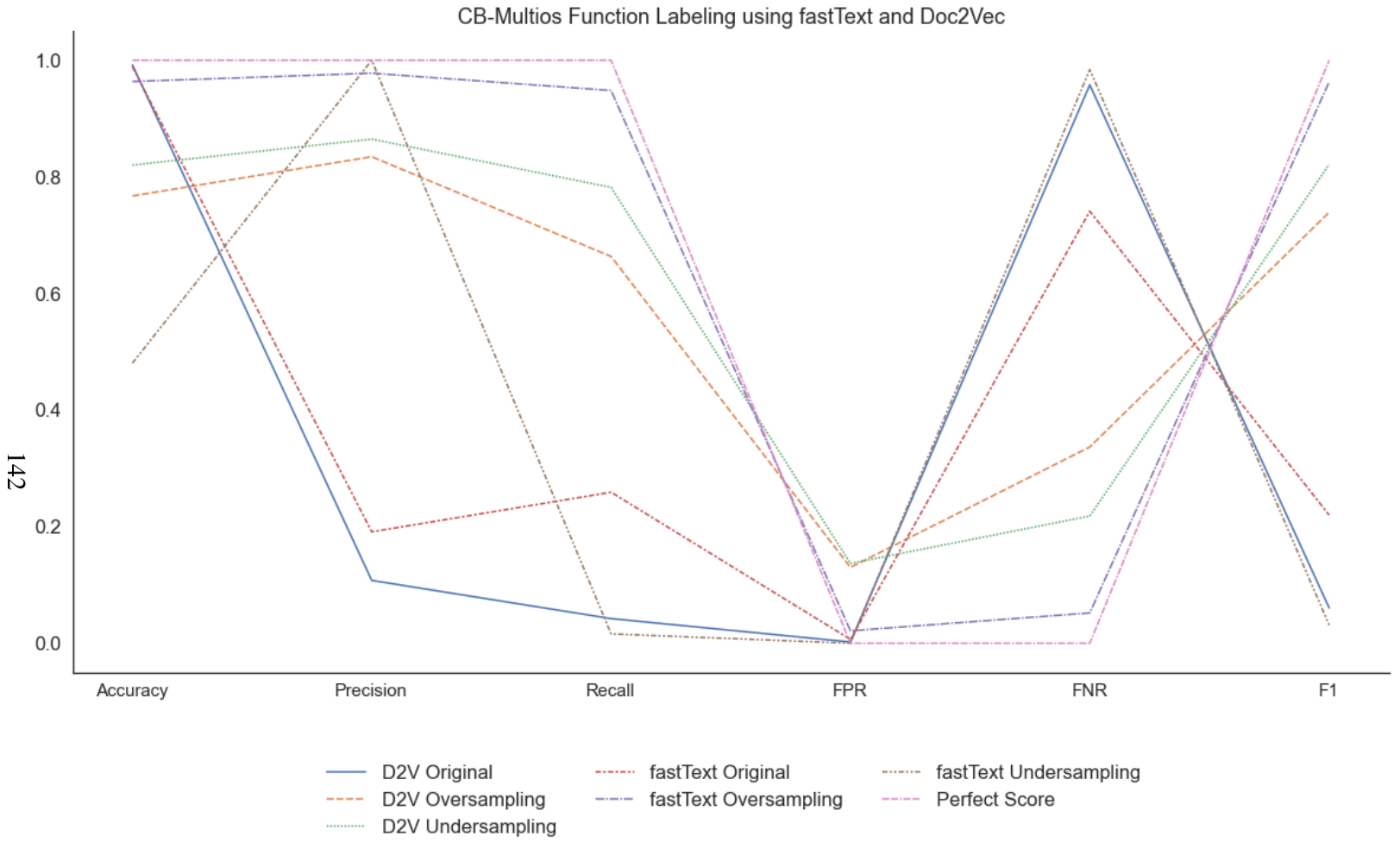


Figure 4.26. Evaluation metrics (CB-Multios with *Doc2Vec* and *fastText*)

Evaluation metrics for each of the 6 test runs, by model (*Doc2Vec* or *fastText*), and CB-Multios dataset type (original unbalanced, and balanced with either random oversampling or undersampling)

Process and Discussion

In this experiment, we first compiled then disassembled the CB-Multios test cases. Then, we used `BiSECT` to extract and curate the $fuzzyInstruction$ sequence for each function in all samples. We used the $fuzzyInstruction$ sequence to train the *Doc2Vec* and *fastText* learning models, then assessed the efficacy of each model using a test set and six evaluation metrics. As in Experiment 1 we conducted three tests, one for each balanced or imbalanced dataset. Ultimately the greatest performance was also achieved when using *fastText* and a dataset balanced with random oversampling. In this test, functions were identified with 96.4% accuracy, 97.8% precision, 94.8% recall, a FPR of 2.1%, FNR of 5.2% , and F1 of 96.3%.

The results from the Experiment 2, in which hybrid-synthetic code was used to train the classifier, are significantly more encouraging than the results from Experiment 1, which used pure-synthetic code. Experiment 2 demonstrates that *fastText* can be trained using the function-level $fuzzyInstruction$ set for disassembled x86 binaries, and that such a classifier provides an effective method to label functions as potentially vulnerable or likely not vulnerable. Furthermore, these experiments demonstrate the ease with which `BiSECT` can be used to extract and curate features to train various machine learning models.

4.6 `BiSECT` to Perform Test Suite Reduction

In Chapter 3 we reviewed a number of open source datasets that collectively contain over 120 thousand test cases. A stratified sample of vulnerability instances and weakness types as they have occurred in the wild throughout the past decade indicates that a total of 2,301 test cases are needed for BVATT. Table 4.12 shows the number of available and required test cases for each strata, i.e., CWE Pillar.

In the following section we demonstrate how `BiSECT` can be used to identify the least similar test cases from a test suite, and propose this method to identify a representative subset of test cases needed for the Benchmark [22][22]

[22]See Chapter 3 for a complete description of BVATT and additional context for this Section

Table 4.12. Available open-source test cases

Pillar	Required Test Cases	Available Test Cases
CWE-284	245	3,309
CWE-435	2	42
CWE-664	1,042	92,733
CWE-682	58	28,876
CWE-691	59	1,511
CWE-693	107	3,321
CWE-697	1	76
CWE-703	7	1,117
CWE-707	737	34,417
CWE-710	43	7,236

4.6.1 Measuring Similarity

Our objective is to reduce the number of test cases to the number required by the stratified sample for each CWE pillar. One approach to achieve this is to select the least *similar* test cases, so that BVATT contains the most diverse set of test cases. This begs the question, can we determine whether two test cases are similar?

Rice's Theorem states that all non-trivial properties of recursively enumerable languages are undecidable. Undecidable in this case means that there does not exist a Turing machine that can solve this problem in the general case. An analog of Rice's Theorem helps us discern whether we should even attempt to decide if two programs are semantically similar. Hopcroft states that [248],

> any nontrivial property that involves what the program does (rather than a lexical or syntactic property of the program itself) must be undecidable.

Ergo, determining whether two programs are semantically similar can be reduced to determining whether a program will halt, and thus is undecidable. All hope is not lost, however. By changing the scope our question we can pivot from an undecidable problem, to one that is certainly surmountable. Rather than examining the semantic properties of a

program, we will instead compare the similarities between the feature vectors for functions within each program.

This is precisely what many machine learning classifiers, e.g., k-nearest neighbors, attempt to achieve. By representing each sample as a feature vector, we can plot each vector as a point in an n-dimensional space, where n is equal to the number of distinct features in the vector. We can then calculate the geometric *distance* between any two points, or in our case, binary samples. The closer two points are the more similar the feature vectors are for those samples. Or, in the context of test suite reduction, the farther two points are from one another, the more dissimilar the feature vectors are for those samples.

There are several methods to calculate the distance between two points; in general, a distance metric must satisfy the following four properties [249]:

1. The distance must never be negative
2. The distance between two identical feature vectors, x and y, is zero
3. The distance from x to y is the same as the distance from y to x
4. The metric must satisfy the triangular inequality: $d(x, y) + d(y, z) >= d(x, z)$

4.6.2 Euclidean Distance

One method to determine the distance between points in an n-dimensional space is to calculate the Euclidean distance, or the hypotenuse between the points. In a two-dimensional space, the hypotenuse can be calculated using the Pythagorean theorem, and can be generalized for n-dimensions as follows,

$$d(\vec{x}, \vec{y}) = \|\vec{x} - \vec{y}\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 \ldots (x_n - y_n)^2} \tag{4.9}$$

More succinctly, the Euclidean distance between two samples, x and y, can be calculated as follows [249]:

$$d_E(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} (x_i - y_i)^2} \tag{4.10}$$

When a feature is comprised of scalar values, e.g., maximum complexity or number of functions, the function to calculate the distance between x_i and y_i merely involves the subtraction of scalar values. However, when a feature contains a set or list of values the the function to calculate the distance between x_i and y_i must include set operations. The **Jaccard distance** or Jaccard dissimilarity coefficient can be used to calculate the distance between features containing a set of values. The formula to calculate the Jaccard distance between two sets, A and B is as follows [37]:

$$d_J(A, B) = 1 - J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{4.11}$$

The Jaccard distance measures dissimilarity between two sets (opposed to the Jaccard Index which measures similarity), and is obtained by dividing the difference of the sizes of the set union and set intersection, by the size of the set union. Consider the feature, *Unique Strings*, which specifies a list of unique strings for every binary sample. To calculate the difference between each set of strings, we could apply the Jaccard distance formula. Figure 4.27 provides a short example that includes the unique strings feature for three binary samples. The *Jaccard Distance* column shows the Jaccard distance from each sample to every other sample in the dataset. The Jaccard distance from any sample to itself is zero. The Jaccard distance from $sample0$ to $sample1$ is 0.5, and the calculation is as follows:

$$d_J(sample0, sample1) = \frac{|sample0 \cup sample1| - |sample0 \cap sample1|}{|sample0 \cup sample1|} = 0.5 \tag{4.12}$$

	Binary Sample	Unique Strings	Jaccard Distance
0	sample_0	[hello, yes, password]	[0.0, 0.5, 0.8]
1	sample_1	[yes, password, no]	[0.5, 0.0, 0.5]
2	sample_2	[yes, no, welcome]	[0.8, 0.5, 0.0]

Figure 4.27. Jaccard Distance calculation for set of unique strings

In the following section we explore how `BiSECT` can be used to extract features from the binary samples, then apply the Euclidean, scalar, and Jaccard distance functions to select the

least similar test cases for BVATT.

4.6.3 Data Cleaning and Transformation

After compiling the samples, `BiSECT` was used to extract eight raw features from the compiled binaries including: strings, $fuzzyInstruction$s, N-grams, Shannon entropy, total functions, internal functions, external functions, and the McCabe cyclomatic complexity for each function in the sample. `BiSECT` was also used to construct additional features from the set of raw features, as described in Section 4.3. After extracting the features `BiSECT` was used to clean and transform all features into feature vectors, so that the dataset would be compatible with two popular visualization and clustering techniques. `BiSECT` outputs the extraction of each feature to a CSV file for easy processing. The `BiSECT` extraction resulted in 12 individual CSV files, which we then read as Pandas DataFrames [174].

After combining the features into a comprehensive feature vector for each sample, the next step is to compute the Euclidean distance between each sample, which in turn is used to create an *NxN similarity matrix* including all samples in the dataset. A similarity matrix is simply the inverse of the distance matrix— in a similarity matrix the diagonal is equal to 1, as a sample has maximum similarity with itself. The resulting similarity matrix will include the pairwise distance values, which are stored as a symmetric matrix. The distance between samples A and B is the same as the distance between samples B and A. Figure 4.28 shows the a subset of the resulting symmetric similarity matrix for 469 test cases associated with CWE 703. In this matrix the closer a value is to 1, the more similar the feature sets were for the respective test cases, and vice versa.

	0	1	2	3	4	5	6	7	8	9	...
0	1.00	0.93	0.93	0.93	0.76	0.93	0.76	0.75	0.76	0.76	...
1	0.93	1.00	0.95	0.95	0.76	0.95	0.76	0.76	0.76	0.76	...
2	0.93	0.95	1.00	0.95	0.76	0.95	0.76	0.76	0.76	0.76	...
3	0.94	0.95	0.95	1.00	0.76	0.95	0.76	0.76	0.76	0.76	...
4	0.78	0.78	0.78	0.78	1.00	0.78	0.95	0.93	0.95	0.95	...
...	...	...	...	...	...	...	...	...	...	...	...

Figure 4.28. Similarity matrix for 469 test cases associated with CWE 703

In the next section, we demonstrate how the similarity matrix can be used in conjunction with clustering algorithms and displayed visually to enable the identification of similar and dissimilar test cases.

4.6.4 Cluster Analysis and Visualization

To illustrate the similarity of each sample, we can display the distance matrix in a number of ways. For example, the matrix can be displayed as a heatmap, where darker tiles indicate less similarity (i.e., a greater Euclidean distance between the samples), while lighter tiles indicate greater overlap between feature vectors for sample pairs. Figure 4.29 shows an example heatmap of the similarity matrix for 469 test cases associated with CWE-703, *Improper Check or Handling of Exceptional Conditions*. The Figure also provides a zoomed view that clearly shows the diagonal values— in which samples are compared to themselves and have the greatest similarity.

The similarity matrix can also be used to identify clusters of similar test cases. During **cluster analysis** similar data objects are grouped into clusters based on similarity [250], [251]. One author states that [251],

> clustering is useful in several exploratory pattern-analysis, grouping, decision-making, and machine-learning situations, including data mining, document retrieval, image segmentation, and pattern classification.

Intuitively, objects in different clusters are less similar then objects in the same cluster. Cluster analysis is often used in the context of unsupervised learning, where analysis is performed on unlabeled samples and used to organize the data into meaningful clusters based strictly on patterns in the data. There are numerous clustering algorithms that organize data into either hierarchies or partitions. Hierarchical clustering algorithms produce a dendrogram that represents the data as a nested series of partitions [251]. In 1963, Joe Ward described the basic procedure to create hierarchical clusters from an NxN matrix. In 1967, Stephen Johnson generalized Ward's procedure, and outlined the process to identify a hierarchical cluster given a similarity (or distance matrix). Given an NxN matrix Johnson's *hierarchical clustering scheme (HCS)* is as follows [252]:

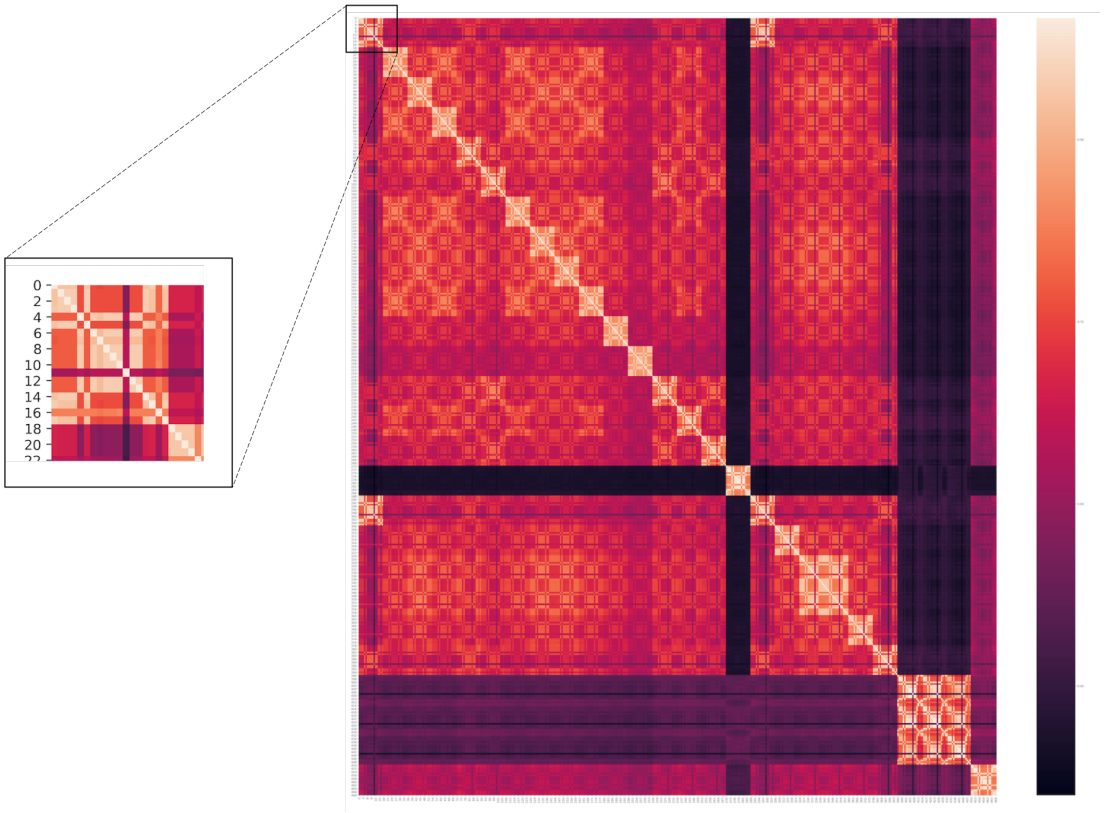


Figure 4.29. Heatmap depicting test case similarity based on extracted features and Euclidean distance. Darker tiles indicate a lower similarity association (i.e., a greater Euclidean distance between the samples); lighter tiles indicate stronger similarity between samples

1. Assign each data object to its own cluster, thus, given N objects, there will initially be N clusters
2. Identify the closest pair of clusters, and merge those into a single cluster, the total number of clusters is now $N - 1$
3. Recompute the distance between each cluster (including the newly merged cluster)
4. Repeat steps 2-3 until all N items have been clustered into a single cluster of size N

We followed Johnson's procedure to cluster the 469 sample test cases related to CWE-703. Figure 4.30 depicts a Seaborn hierarchical clustermap of test case similarity using Euclidean distances. This clustermap includes both a heatmap and hierarchical dendrogram [253]. The

dendrogram illustrates the clusters, and is shown on the upper X-axis. The numbers along the X and Y axis represent test case identifiers; the cases are reordered to support visualization of the clusters.

Cluster analysis can be used to down-select test cases for BVATT, by selecting the least similar test cases in an iterative fashion. In other words, the two test cases with the greatest distance between them will be selected, and so on, until the needed number of cases for each strata has been selected. This method ensures that BVATT will be comprised of test cases exhibiting the greatest differences in their feature sets, and thus, VATTs will be assessed on their performance against test cases with the greatest variance.

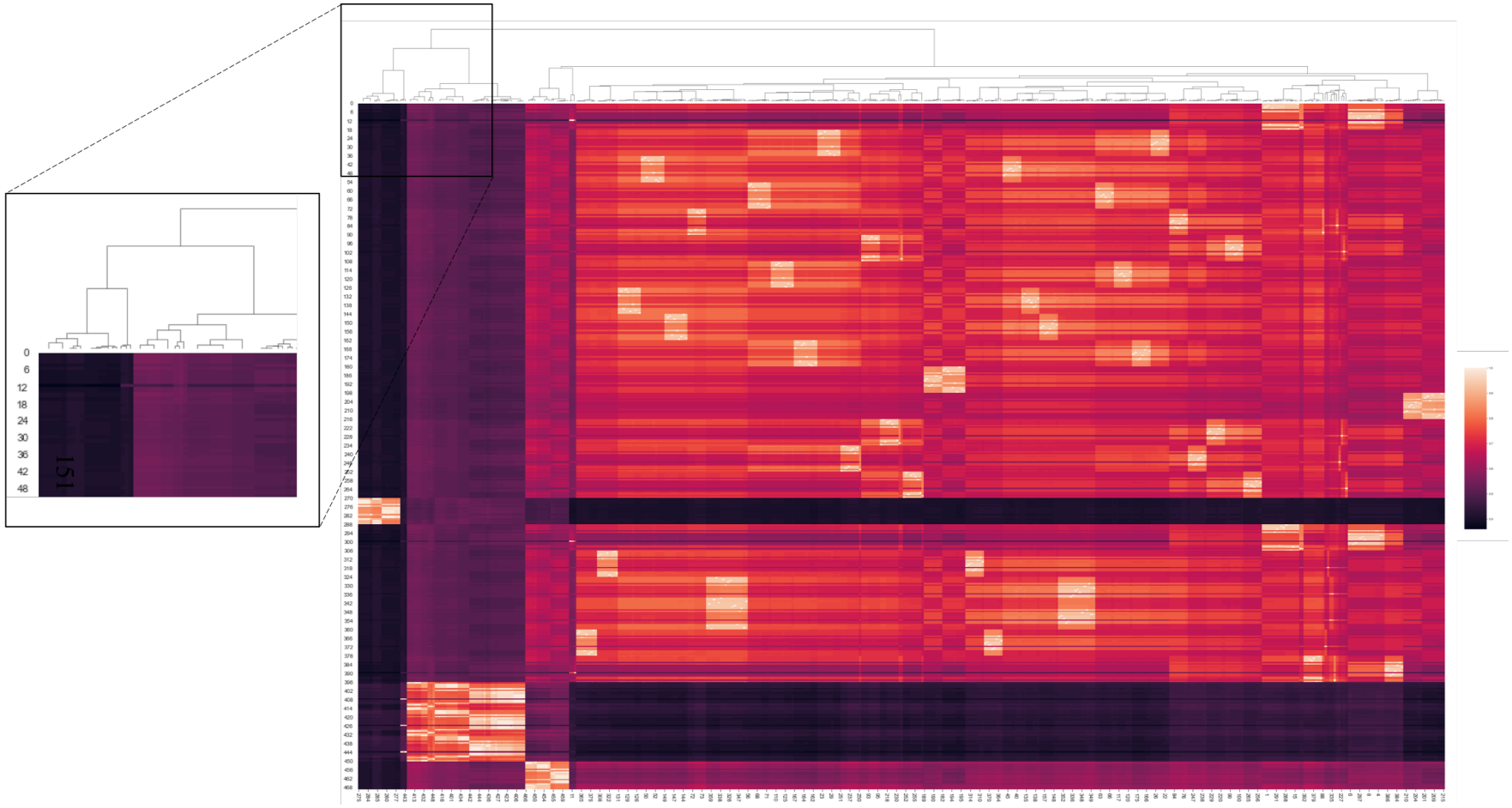


Figure 4.30. Clustermap depicting test case similarity using Euclidean distances

Clustermap depicting test case similarity using Euclidean distances; includes both a heatmap and hierarchical dendrogram. The dendrogram illustrates clusters, and is shown on the upper X-axis. Test case identifiers along the X and Y axis are reordered to support cluster visualization

Furthermore, to identify t test cases (in the case of BVATT $t = 2,301$) with the greatest distance between their feature vectors, and ultimately to select the most diverse test cases for BVATT, we can leverage the convenient matrix format generated by `BiSECT`. We simply convert the symmetric similarity matrix into a distance matrix, then convert it to a lower or upper triangle, and finally, apply a simple function to select t values. We provide an example of this technique in Listing 4.2:

4.6.5 Conclusion

The proposed method to perform test suite reduction for BVATT is conducted almost entirely using `BiSECT`. `BiSECT` extracts, cleans, and transforms the features, creates the feature vectors and corresponding matrices, and performs the distance calculations Test cases selected using this method will have the least similar feature vectors, and ensure that BVATT is comprised of the most diverse (from a feature perspective) set of test cases.

4.7 Summary

In this Chapter we introduced a new tool, `BiSECT`. `BiSECT` enables researchers to synthesize the extraction, cleaning, and transformation of a number of common features from compiled binaries. Ultimately, `BiSECT` reduces the domain expertise and manual effort required to complete the feature extraction, cleaning, and transformation steps in support of vulnerability analysis using machine learning and data mining techniques.

To demonstrate the utility of `BiSECT`, we provided two example applications in Sections 4.5 and 4.6. In Section 4.5, we use `BiSECT` to extract, clean, and transform roughly 5M disassembled x86 functions from 64K compiled C \ C++ programs. We then use the output of `BiSECT` to compare the efficacy of two representation models and corresponding classifiers, *fastText* and *Doc2Vec*, when they are given the task of labeling potentially vulnerable functions. In Section 4.6, we use `BiSECT` to identify and eliminate similar test cases in support of BVATT.

In the first example application we measured the impact that unbalanced and balanced training data (using random over or under sampling) can have on machine learning classifiers. We observed that classifiers trained using vulnerability datasets that were balanced using random oversampling consistently outperformed those trained using either unbalanced

```
def max_distance(tri_upper_no_diag):
    '''
    for every test case:
        get coordinates for the least similar test case to this one
        retrieve the actual value using the coordinates
        add the test case coordinates and actual value to dict.
        e.g., (0, 443): 0.6
    return the final dict of max values
    '''
    max_vals = {}
    for index, row in enumerate(tri_upper_no_diag):
        max_row_coordinates = list([index, (np.argpartition(row,
        ↪ -1)[-1:])[0]])
        max_row_val = (row[max_row_coordinates])[1]
        max_vals[tuple(max_row_coordinates)] = max_row_val
    return max_vals

def n_max(max_vals, t):
    # return the t largest values from the dict of max values
    t_largest = nlargest(t, max_vals.items(), key=lambda i: i[1])
    return t_largest

distance_matrix = 1-similarity_matrix # convert similarity to distance
↪ matrix
max_vals = max_distance(tri_upper_no_diag) # get a dict of max distances
t = 37 # specify how many test cases are needed
t_largest = n_max(max_vals, t) # get the t least similar test cases
```

Listing 4.2: Example implementation of selecting t least similar test cases

datasets or datasets balanced using random undersampling. We also demonstrated that classifiers trained using pure synthetic data (i.e., Juliet) do a poor job labeling function level code as likely vulnerable or likely not vulnerable in more realistic code (i.e., CB-Multios). Finally, the metrics achieved by *fastText* when trained using the CB-Multios dataset (balanced using random oversampling) were nearly perfect across the board. This success not only

demonstrates how easily features extracted by `BiSECT` can be used to train models such as these, but also that the models themselves provide an effective launching point for further vulnerability research.

Collectively, these example applications demonstrate the ease in which `BiSECT` can be used to synthesize the extraction, cleaning, and transformation of features from binary files in support of at-scale binary vulnerability analysis.

CHAPTER 5:
Conclusion

This Chapter presents a summary of our work. We begin with a review of the major challenges addressed throughout this research. This is followed by a discussion of our findings, and finally, we conclude with recommendations for future research.

5.1 Summary

5.1.1 Challenges and Solutions

The fundamental challenge and corresponding motivation for this work is simple, but mountainous:

Software pervades nearly every aspect of our lives today, and so do its vulnerabilities.

From this, a mountain range of challenges arises.

Hundreds of thousands of vulnerabilities have been reported to the CVE. In spite of billions of dollars allocated to cybersecurity, data exfiltration and ransomware attacks persist [14], [15]. The question remains:

How can we identify and reduce software vulnerabilities?

The number of specialists able to perform manual vulnerability testing is eclipsed by the amount and variety of software there is to secure. Static, dynamic, and hybrid testing can be used to identify vulnerabilities in software in a more automated fashion. Hundreds of vulnerability analysis tools and techniques have entered the market. Researchers and consumers rely on VATTs to amplify the vulnerability analysis process.

Throughout this work, two major challenges were addressed. We again summarize these challenges and our solutions as follows.

5.1.2 Challenge 1

There is no comprehensive method to assess the efficacy of VATTs. We have no comparative metrics of the types of vulnerabilities each tool can or cannot find, how quickly they can do so, or the number of false positives they report. So, when a consumer wants to use a vulnerability analysis tool, they have no method of comparing the alternatives to determine which is a good choice for a specific use case. When a developer wants to modify their tool or enter the market with a new one, no standard method is available to compare their VATT against existing ones. Consequently, consumers and developers compare tools in a disjointed fashion, which can result in misinformation, subjectivity, redundancy, and inconsistency.

Based on the limitations of existing datasets, and the absence of a community-accepted benchmark we determined that the community is in need of an overarching framework to guide the implementation of BVATTs. The framework for BVATT should include both the core characteristics of the benchmark, and metrics that can be used to quantitatively compare VATTs.

Solution 1

To address the first challenge, we provided a framework for BVATT in Chapter 3. Before creating the framework, we examined the limitations of existing datasets containing known vulnerabilities. We then explored lessons learned from other computing benchmarks. We used these lessons to identify key characteristics for BVATT. We determined that BVATT should be repeatable, reproducible, fair, and verifiable. After detailing each of these characteristics, we investigated what it would require for BVATT to be representative of reality. To this end, we determined that a stratified random sample of CVEs and corresponding CWEs would provide an excellent representation of known vulnerability instances and types that have occurred in the wild throughout the past decade. Finally, we explored methods to leverage test cases in existing vulnerability datasets for BVATT. In Section 3.3.6 we demonstrated that existing methods to identify and eliminate similar test cases may require significant manual effort. Thus, we sought a more automated solution. This research ultimately led us to create a new tool, `BiSECT`. On the journey to create `BiSECT` we were confronted with the second challenge addressed by this work.

5.1.3 Challenge 2

There is no tool to synthesize the at scale extraction, cleaning, and transformation of features commonly used in data mining and machine learning-based vulnerability analysis Thus, current research in this area is limited to a niche group of computer scientists with the requisite domain expertise in machine learning and binary vulnerability analysis. This disconnect acts as barrier to both security and machine learning researchers, and ultimately limits advancements in a promising area of vulnerability analysis.

Instead, we need to get to a place where a vulnerability analyst can succinctly extract common features from hundreds of binaries using a single tool. The tool should also clean and transform those features so that they're compatible with machine learning and data mining techniques.

Solution 2

To address the second challenge, we introduced a new tool, dubbed `BiSECT`, in Chapter 4. `BiSECT` enables researchers to synthesize the extraction, cleaning, and transformation of a number of common features from compiled binaries. Ultimately, `BiSECT` reduces the domain expertise required to complete the feature extraction, cleaning, and transformation steps in support of vulnerability analysis leveraging machine learning and data mining techniques.

To demonstrate the utility of `BiSECT`, we provided two example applications. In the first example, we used `BiSECT` to extract, clean, and transform roughly 5M disassembled x86 functions from 64K compiled C \ C++ programs. We then used the output of `BiSECT` to compare the efficacy of two representation models and corresponding classifiers, *fastText* and *Doc2Vec*, when they are given the task of labeling potentially vulnerable functions. Finally, we returned to the work we began in Chapter 3, and demonstrated how `BiSECT` can be used to identify and eliminate similar test cases in a test suite. This approach can be used to select the most diverse test cases for BVATT.

5.2 Ongoing and Future Work

In Chapter 3 we provided a framework for BVATT, but there remains much work to be done including the actual implementation and maintenance of a benchmark. One of the most significant challenges to this area is the identification of real-world benchmark problems.

We stand in solidarity with the numerous researchers who have stated that the vulnerability research community is in need of a standard dataset of real-world vulnerabilities [22], [25], [27], [28]. As a step towards this goal, we proposed a batch method to identify the differences (i.e., diff) in the original and patched versions of binary files using Ghidra and Python. During the diff process, the original function corresponding to any function modified during a patch is deemed vulnerable (the updated function is deemed not vulnerable). Using the CB-Multios corpus as an example, we demonstrated how our technique can be leveraged to identify and label vulnerabilities at a function level. In support of the development of BVATT and a standard dataset of known vulnerabilities, we have made our code publicly available via GitHub.

There remains a disparity between vulnerability researchers using source code versus compiled code. We discussed this imbalance extensively in Chapter 4. To summarize, of 58 publications reviewed in two major surveys [of machine learning and data mining based vulnerability analysis], a mere 3% used binary code to support their work [27], [28]. Binary vulnerability analysis using machine learning and data mining remains a niche area of research, yet there are situations when binary analysis is necessary, e.g., a user may need to validate that properties proven by analyzing a program's source code still hold after the program has been compiled, or they may not have access to a program's source code [21], [47], [49], [50]. We attribute this disparity to the inverse relationship between semantic information and level of difficulty. That is to say that as semantic information decreases, the level of difficulty of analyzing a sample for vulnerabilities increases.

To support and encourage research in binary code vulnerability analysis we provided the `BiSECT` tool. Our hope is that `BiSECT` will be used as both a research and educational aid. We see many opportunities for future work with `BiSECT`. For example, `BiSECT` could be leveraged to create open source courseware on vulnerability analysis using data mining and machine learning. Additionally, recent surveys indicate that graph-based methods to identify vulnerabilities in source and binary code show encouraging results [27], [28]. `BiSECT` does not currently extract any graphical program representations (e.g., AST, CPG, CFG, etc.), however, it could be extended to do so. Also, Experiment 1, presented in Chapter 4 could be extended to include the types of vulnerabilities correctly labeled in the hybrid-synthetic dataset. The experiments could also be redone using real-world code to train and test the models.

Additionally, researchers have attempted to develop new code representation models to recover the reuse of vulnerabilities in functions [146], [254]. We explored the efficacy of using features extracted by `BiSECT` directly with existing representation models to achieve comparable results. Like [146], we used the dataset of vulnerabilities provided by [254]. This dataset contains 108,474 functions (including stubs), associated with 8 vulnerabilities. Variants for each vulnerability are generated using different source code versions, and or, different versions of three popular compilers: CLANG, GCC, and ICC. The goal is to identify the variants of each vulnerability in the datasets. [146] demonstrated that their representation model, *Asm2Vec*, could be used to identify the variants for all vulnerabilities in the dataset with 100% recall, a FPR of 0, ROC and CROC of 1. We were curious if `BiSECT` could be used to identify function level vulnerability reuse. First, we used `BiSECT` to create the $fuzzyInstruction$ sequences and function level cyclomatic complexity features for all functions in the dataset. Then, we trained the *Doc2Vec* and *fastText* representation models using the extracted features. Using this workflow we correctly grouped all but one of the vulnerabilities in the dataset. We plan to continue to refine our technique and the features extracted by `BiSECT` to achieve 100% accuracy.

In addition to these opportunities for future work, we have continued to experiment with additional sentence embedded techniques to identify vulnerabilities in software. The experiments presented in Chapter 4 were conducted using Doc2Vec and fastText, however numerous other embedding techniques could be used. For example, our most recent experiment with Sentence-BERT (SBERT) [255], a state-of-the-art sentence embedding technique, indicates that it can be trained to achieve metrics comparable to those achieved by fastText in our best scenario.

5.2.1 Final Thoughts

In this work we did not set out to address the epidemic of software vulnerabilities as a whole. Rather we sought solutions that could chip away at major challenges stemming from the existence of software vulnerabilities.

Whether it's developing a dataset of real-world vulnerabilities in support of BVATT or creating new features for `BiSECT` there remains much work to be done.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] Apple, Inc., "App Store." Available: https://www.apple.com/app-store/

[2] AppBrain, "Number of Android applications on the Google Play store," July 2021. Available: https://www.appbrain.com/stats/number-of-android-apps

[3] N. Lee, "As the U.S. faces a flurry of ransomware attacks, experts warn the peak is likely still to come," June 2021. Available: https://www.cnbc.com/2021/06/10/heres-how-much-ransomware-attacks-are-costing-the-american-economy.html

[4] K. Olmstead and A. Smith, "Americans and Cybersecurity," *Pew Research Center*, vol. 26, pp. 311–327, Jan. 2017. Available: https://www.pewresearch.org/internet/2017/01/26/3-attitudes-about-cybersecurity-policy/

[5] M. Bada and J. R. Nurse, "The social and psychological impact of cyberattacks," in *Emerging Cyber Threats and Cognitive Vulnerabilities*. Elsevier, 2020, pp. 73–92. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780128162033000046

[6] MITRE, "CVE - Home," May 2020. Available: https://cve.mitre.org/about/index.html

[7] L. Allodi and F. Massacci, "Security Events and Vulnerability Data for Cybersecurity Risk Estimation: Cybersecurity Risk Estimation," *Risk Analysis*, vol. 37, no. 8, pp. 1606–1627, Aug. 2017. Available: http://doi.wiley.com/10.1111/risa.12864

[8] IBM Security, "Cost of a Data Breach Report 2020," Tech. Rep., 2020. Available: https://www.ibm.com/account/reg/us-en/signup?formid=urx-46542

[9] Nicolas Falliere, Liam O. Murchu, and Eric Chien, "W32.Stuxnet Dossier, Version 1.4," Symantec Corp., Tech. Rep., February 2011 [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[10] E. Chien, Liam O. Murchu, and N. Falliere, "W32.Duqu: The Precursor to the Next Stuxnet," Symantec Corp., Tech. Rep., November 23, 2011 [Online]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjukr65lqzyAhVlHTQIHZZcDhwQFnoECAUQAQ&url=https%3A%2F%2Fdocs.broadcom.com%2Fdoc%2Fw32-duqu-11-en&usg=AOvVaw3IwIHVcd_NMFWb7zILN506

[11] "Fiscal Years 2014-2018 Strategic Plan," U.S. Department of Homeland Security, Tech. Rep., May 2018. Available: https://www.dhs.gov/publication/dhs-cybersecurity-strategy

[12] Navy, "OPNAV 5239.4 (N2N6) Chief of Naval Operations Cybersecurity Safety Program," United States Navy, Tech. Rep., Sep. 2018. Available: https://standards.globalspec.com/std/13062304/OPNAV%205239.4

[13] FireEye, "Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor," Dec. 2020. Available: https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html

[14] B. Krebs, "At Least 30,000 U.S. Organizations Newly Hacked Via Holes in Microsoft's Email Software – Krebs on Security," Krebs on Security, Tech. Rep., Mar. 2021. Available: https://krebsonsecurity.com/2021/03/at-least-30000-u-s-organizations-newly-hacked-via-holes-in-microsofts-email-software/

[15] E. J. Macias, Amanda, "Colonial Pipeline paid $5 million ransom to hackers," May 2021. Available: https://www.cnbc.com/2021/05/13/colonial-pipeline-paid-ransom-to-hackers-source-says.html

[16] Department of Defense, "DOD Releases Fiscal Year 2021 Budget Proposal," Feb. 2020. Available: https://www.defense.gov/Newsroom/Releases/Release/Article/2079489/dod-releases-fiscal-year-2021-budget-proposal/

[17] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 2nd ed. (Business Data Processing: A Wiley Series). [Online]: John Wiley & Sons, Sep. 2011. Available: https://books.google.com/books?id=86rz6UExDEEC

[18] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy Magazine*, vol. 2, no. 6, pp. 76–79, Nov. 2004. Available: http://ieeexplore.ieee.org/document/1366126/

[19] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *2010 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, 2010, pp. 317–331. Available: http://ieeexplore.ieee.org/document/5504796/

[20] Y. Shoshitaishvili, "Building a Base for Cyber-Autonomy," Dissertation, University of California, Santa Barbara, CA, USA, 2017. Available: https://pqdtopen.proquest.com/doc/1968634917.html?FMT=ABS

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, and C. Kruegel, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[22] K. N. Afanador and C. E. Irvine, "Representativeness in the Benchmark for Vulnerability Analysis Tools (B-VAT)," in *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association, 2020, p. 5. Available: https://www.usenix.org/conference/cset20/presentation/afanador

[23] "Executive Order on Improving the Nation's Cybersecurity," May 2021. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[24] P. E. Black, "SARD: Thousands of Reference Programs for Software Assurance," *Journal of Cyber Security and Information Systems*, vol. 5, no. 3, p. 13, Oct. 2017. Available: https://www.nist.gov/publications/sard-thousands-reference-programs-software-assurance

[25] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA: IEEE, May 2016, pp. 110–121. Available: http://ieeexplore.ieee.org/document/7546498/

[26] B. Caswell, "Cyber Grand Challenge Corpus," Jan. 2017. Available: http://www.lungetech.com/cgc-corpus/

[27] S. M. Ghaffarian and H. R. Shahriari, "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey," *ACM Computing Surveys*, vol. 50, no. 4, pp. 1–36, Aug. 2017. Available: http://dl.acm.org/citation.cfm?doid=3135069.3092566

[28] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software Vulnerability Detection Using Deep Neural Networks: A Survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020. Available: https://ieeexplore.ieee.org/document/9108283/

[29] D. Wichers, "OWASP Benchmark," The OWASP Foundation, Oct. 2016. Available: https://owasp.org/www-project-benchmark/

[30] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN," *Applied Sciences*, vol. 9, no. 19, p. 4086, Sep. 2019. Available: https://www.mdpi.com/2076-3417/9/19/4086

[31] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. Available: http://arxiv.org/abs/1801.01681

[32] "STONESOUP Phase 3 Test and Evaluation Report," Office of the Director of National Intelligence, Tech. Rep., Dec. 2014. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjPh8q7mKzyAhUrGDQIHVBzC98QFnoECAIQAQ&url=https%3A%2F%2Fsamate.nist.gov%2FSARD%2Fresources%2FSTONESOUP_Test_and_Evaluation_Phase_3_Final_Report.pdf&usg=AOvVaw0STo3utK2kU1pLkGxLpeIa

[33] J. Walden, J. Stuckman, and R. Scandariato, "Predicting Vulnerable Components: Software Metrics vs Text Mining," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. Naples, Italy: IEEE, Nov. 2014, pp. 23–33. Available: http://ieeexplore.ieee.org/document/6982351/

[34] M. Siavvas, D. Kehagias, and D. Tzovaras, "A Preliminary Study on the Relationship Among Software Metrics and Specific Vulnerability Types," in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. Las Vegas, NV, USA: IEEE, Dec. 2017, pp. 916–921. Available: https://ieeexplore.ieee.org/document/8560918/

[35] K. Z. Sultana, "Towards a software vulnerability prediction model using traceable code patterns and software metrics," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL: IEEE, Oct. 2017, pp. 1022–1025. Available: http://ieeexplore.ieee.org/document/8115724/

[36] I. Kalouptsoglou, M. Siavvas, D. Tsoukalas, and D. Kehagias, "Cross-Project Vulnerability Prediction Based on Software Metrics and Deep Learning," in *Computational Science and Its Applications – ICCSA 2020*, vol. 12252, O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blečić, D. Taniar, B. O. Apduhan, A. M. A. C. Rocha, E. Tarantino, C. M. Torre, and Y. Karaca, Eds. Cham: Springer International Publishing, 2020, pp. 877–893. Available: http://link.springer.com/10.1007/978-3-030-58811-3_62

[37] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, 3rd ed. Burlington, MA: Elsevier, 2012.

[38] A. K. Agrawal, J. Gans, and A. Goldfarb, "The Economics of Artificial Intelligence: An Agenda," *University of Chicago Press*, p. 42, 2019.

[39] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in *2010 Third*

International Conference on Software Testing, Verification and Validation. Paris, France: IEEE, 2010, pp. 421–428. Available: http://ieeexplore.ieee.org/document/5477059/

[40] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward Large-Scale Vulnerability Discovery using Machine Learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. New Orleans Louisiana USA: ACM, Mar. 2016, pp. 85–96. Available: https://dl.acm.org/doi/10.1145/2857705.2857720

[41] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. Urbana Illinois: ACM, Apr. 2015, pp. 1–9. Available: https://dl.acm.org/doi/10.1145/2746194.2746198

[42] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. Chengdu: IEEE, Dec. 2017, pp. 1298–1302. Available: http://ieeexplore.ieee.org/document/8322752/

[43] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning Binary Code with Deep Learning to Detect Software Weakness," in *KSII the 9th International Conference on Internet (ICONI) 2017 Symposium.*, 2017, p. 5.

[44] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, and L. Qu, "Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection," in *International Conference on Learning Representations*, 2018, p. 15. Available: https://openreview.net/pdf?id=ByloIiCqYQ

[45] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Orlando, FL: IEEE, Dec. 2018, pp. 757–762. Available: https://ieeexplore.ieee.org/document/8614145/

[46] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *arXiv:1803.04497 [cs, stat]*, Aug. 2018. Available: http://arxiv.org/abs/1803.04497

[47] K. Redmond, L. Luo, and Q. Zeng, "A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis," *arXiv:1812.09652 [cs]*, Dec. 2018. Available: http://arxiv.org/abs/1812.09652

[48] X. Li, Q. Yu, and H. Yin, "PalmTree: Learning an Assembly Language Model for Instruction Embedding," *arXiv:2103.03809 [cs]*, May 2021. Available: http://arxiv.org/abs/2103.03809

[49] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, "The Mayhem Cyber Reasoning System," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 52–60, Mar. 2018. Available: http://ieeexplore.ieee.org/document/8328972/

[50] Kenneth Thompson, "Reflections on Trusting Trust," *Communications of the A.C.M.*, vol. 27, no. 8, pp. 761–763, 1984. Available: https://dl.acm.org/doi/pdf/10.1145/1283920.1283940

[51] Hopper, Grace, "Log Book With Computer Bug," 1947. Available: https://americanhistory.si.edu/collections/search/object/nmah_334663

[52] T. P. Hughes, *Networks of Power: Electrification in Western Society, 1880-1930*. JHU Press, Mar. 1993.

[53] F. R. Shapiro, "Etymology of the Computer Bug: History and Folklore," *American Speech*, vol. 62, no. 4, p. 376, 1987. Available: https://www.jstor.org/stable/455415?origin=crossref

[54] P. M. Melliar-Smith and B. Randell, "Software reliability: The role of programmed exception handling." *ACM conference on Language design for reliable software*, pp. 95–100, 1976.

[55] P. J. Denning, "Fault Tolerant Operating Systems," *ACM Computing Surveys*, vol. 8, no. 4, pp. 359–389, Dec. 1976. Available: https://dl.acm.org/doi/10.1145/356678.356680

[56] "CVE - Terminology." Available: https://cve.mitre.org/about/terminology.html#vulnerability

[57] National Institute of Standards and Technology, "Standards for security categorization of federal information and information systems," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST FIPS 199, Feb. 2004. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.199.pdf

[58] R. D. Houston and G. Harmon, "Vannevar Bush and memex," *Annual Review of Information Science and Technology*, vol. 41, no. 1, pp. 55–92, 2007. Available: https://onlinelibrary.wiley.com/doi/10.1002/aris.2007.1440410109

[59] Turing, Alan M, "Lecture to the London Mathematical Society on 20 February 1947," *MD Computing*, vol. 12, pp. 390–390, 1995.

[60] P. Denning and T. Lewis, "Intelligence May Not Be Computable," *American Scientist*, vol. 107, no. 6, p. 346, 2019. Available: https://www.americanscientist.org/article/intelligence-may-not-be-computable

[61] W. S. Mcculloch and W. Pitts, "A Logical Calculus of the Ideas Immanent In Nervous Activity," *Bulletin of Mathematical Biophysics 5*, pp. 115–133, 1943.

[62] C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2018. Available: http://link.springer.com/10.1007/978-3-319-94463-0

[63] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. Available: http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519

[64] R. E. Neapolitan, *Artificial Intelligence: With an Introduction to Machine Learning*, 2nd ed. Chapman and Hall/CRC, Mar. 2018. Available: https://www.taylorfrancis.com/books/9781315144863

[65] X. Wang, Y. Zhao, and F. Pourpanah, "Recent advances in deep learning," *International Journal of Machine Learning and Cybernetics*, vol. 11, no. 4, pp. 747–750, Apr. 2020. Available: http://link.springer.com/10.1007/s13042-020-01096-5

[66] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. Available: http://doi.wiley.com/10.1112/plms/s2-42.1.230

[67] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, June 2012.

[68] P. Jalote, *An Integrated Approach to Software Engineering*. Springer Science & Business Media, Oct. 2005.

[69] S. C. Johnson, *Lint, a C Program Checker*. Murray Hill: Bell Telephone Laboratories, 1977. Available: http://squoze.net/UNIX/v7/files/doc/15_lint.pdf

[70] T. Koenig, "Gets(3) - Linux manual page," 2021. Available: https://man7.org/linux/man-pages/man3/gets.3.html

[71] DeKok, Alan, "PScan: A limited problem scanner for C source files," 2000. Available: http://deployingradius.com/pscan/

[72] Secure Software Inc., "RoughAuditing Tool for Security (RATS)," Dec. 2013. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[73] J. Viega, J. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. New Orleans, LA, USA: IEEE Comput. Soc, 2000, pp. 257–267. Available: http://ieeexplore.ieee.org/document/898880/

[74] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *10th USENIX Security Symposium (USENIX Security 01)*. Washington, D.C.: USENIX Association, Aug. 2001. Available: https://www.usenix.org/conference/10th-usenix-security-symposium/statically-detecting-likely-buffer-overflow

[75] Wheeler, David, "Flawfinder home page," 2006. Available: http://www.dwheeler.com/flawfinder

[76] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. Available: http://portal.acm.org/citation.cfm?doid=96267.96279

[77] Felderer, Michael, *Testing Code Security*. Auerbach Publications, June 2007. Available: http://www.taylorfrancis.com/https://www-taylorfrancis-com.libproxy.nps.edu/books/mono/10.1201/9781420013795/testing-code-security-maura-van-der-linden

[78] A. Takanen, J. DeMott, C. Miller, and A. Kettunen, Eds., *Fuzzing for Software Security Testing and Quality Assurance*, second edition ed. (Artech House Information Security and Privacy Series). Norwood, MA: Artech House, 2018.

[79] Sutton, Michael, Greene, Adam, and Amini, Pedram, *Fuzzing Brute Force Vulnerability Discovery*, 1st ed. New Jersey: Addison-Wesley, 2007.

[80] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. Available: https://www.ndss-symposium.org/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[81] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding Software Vulnerabilities by Smart Fuzzing," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. Berlin, Germany: IEEE, Mar. 2011, pp. 427–430. Available: http://ieeexplore.ieee.org/document/5770635/

[82] Zalewski, Michal, "American Fuzzy Lop," Nov. 2013. Available: https://lcamtuf.coredump.cx/afl/

[83] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, p. 6, Dec. 2018. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y

[84] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976. Available: https://dl.acm.org/doi/10.1145/360051.360056

[85] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, July 1977. Available: https://dl.acm.org/doi/10.1145/359636.359712

[86] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli, "Detecting Control Flow in Smarphones: Combining Static and Dynamic Analyses," in *Cyberspace Safety and Security*, vol. 7672, Y. Xiang, J. Lopez, C.-C. J. Kuo, and W. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 33–47. Available: http://link.springer.com/10.1007/978-3-642-35362-8_4

[87] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, Oct. 2009. Available: http://link.springer.com/10.1007/s10009-009-0118-1

[88] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[89] R. Sethi, *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1996.

[90] A. Meneely and L. Williams, "Strengthening the empirical analysis of the relationship between Linus' Law and software security," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*. Bolzano-Bozen, Italy: ACM Press, 2010, p. 1. Available: http://portal.acm.org/citation.cfm?doid=1852786.1852798

[91] Y. Shin and L. Williams, "An initial study on the use of execution complexity metrics as indicators of software vulnerabilities," in *Proceeding of the 7th International Workshop on Software Engineering for Secure Systems - SESS '11*. Waikiki, Honolulu, HI, USA: ACM Press, 2011, p. 1. Available: http://portal.acm.org/citation.cfm?doid=1988630.1988632

[92] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, Nov. 2011. Available: http://ieeexplore.ieee.org/document/5560680/

[93] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, vol. 2013, no. 5, pp. 8–17, May 2013. Available: https://linkinghub.elsevier.com/retrieve/pii/S1361372313700459

[94] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. Baltimore, Maryland: IEEE, Oct. 2013, pp. 65–74. Available: http://ieeexplore.ieee.org/document/6681339/

[95] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. Hong Kong, China: ACM Press, 2014, pp. 257–268. Available: http://dl.acm.org/citation.cfm?doid=2635868.2635880

[96] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver Colorado USA: ACM, Oct. 2015, pp. 426–437. Available: https://dl.acm.org/doi/10.1145/2810103.2813604

[97] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Functionwith an Existing Exploit," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. New Orleans Louisiana USA: ACM, Mar. 2016, pp. 97–104. Available: https://dl.acm.org/doi/10.1145/2857705.2857750

[98] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, p. 16, 2001. Available: https://dl.acm.org/doi/pdf/10.1145/502059.502041

[99] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 10, 2005. Available: https://dl.acm.org/doi/pdf/10.1145/1095430.1081754

[100] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 10, 2005. Available: https://dl.acm.org/doi/pdf/10.1145/1095430.1081755

[101] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering - ESEC-FSE '07*. Dubrovnik, Croatia: ACM Press, 2007, p. 35. Available: http://portal.acm.org/citation.cfm?doid=1287624.1287632

[102] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.*, 2007, p. 10. Available: https://dl.acm.org/doi/pdf/10.1145/1287624.1287630

[103] C. Ray-Yaung, A. Podgurski, and J. Yang, "Discovering Neglected Conditions in Software by Mining Dependence Graphs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 579–596, Sep. 2008. Available: http://ieeexplore.ieee.org/document/4492791/

[104] S. Thummalapenta and T. Xie, "Alattin: Mining Alternative Patterns for Detecting Neglected Conditions," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. Auckland, New Zealand: IEEE, Nov. 2009, pp. 283–294. Available: http://ieeexplore.ieee.org/document/5431765/

[105] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: Lightweight cross-project anomaly detection," in *Proceedings of the 19th International Symposium on Software Testing and Analysis - ISSTA '10*. Trento, Italy: ACM Press, 2010, p. 119. Available: http://portal.acm.org/citation.cfm?doid=1831708.1831723

[106] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 499–510. Available: http://dl.acm.org/citation.cfm?doid=2508859.2516665

[107] F. Yamaguchi, "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, 2011, p. 10. Available: https://dl.acm.org/doi/abs/10.5555/2028052.2028065

[108] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*. Orlando, Florida: ACM Press, 2012, p. 359. Available: http://dl.acm.org/citation.cfm?doid=2420950.2421003

[109] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. Essen, Germany: ACM Press, 2012, p. 310. Available: http://dl.acm.org/citation.cfm?doid=2351676.2351733

[110] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, Oct. 2014. Available: http://ieeexplore.ieee.org/document/6860243/

[111] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2014, pp. 590–604. Available: http://ieeexplore.ieee.org/document/6956589/

[112] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic Inference of Search Patterns for Taint-Style Vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 797–812. Available: https://ieeexplore.ieee.org/document/7163061/

[113] Y. Pang, X. Xue, and A. S. Namin, "Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. Miami, FL, USA: IEEE, Dec. 2015, pp. 543–548. Available: http://ieeexplore.ieee.org/document/7424372/

[114] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. Miami Beach, FL, USA: IEEE, Dec. 2007, pp. 477–486. Available: http://ieeexplore.ieee.org/document/4413013/

[115] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining Bug Databases for Unidentified Software Vulnerabilities," in *2012 5th International Conference on Human System Interactions*. Perth, Australia: IEEE, June 2012, pp. 89–96. Available: http://ieeexplore.ieee.org/document/6473768/

[116] M. Alvares, T. Marwala, and F. B. de Lima Neto, "Applications of computational intelligence for static software checking against memory corruption vulnerabilities," in *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*. Singapore, Singapore: IEEE, Apr. 2013, pp. 59–66. Available: http://ieeexplore.ieee.org/document/6597207/

[117] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the 23rd International Conference on World Wide Web - WWW '14*. Seoul, Korea: ACM Press, 2014, pp. 63–74. Available: http://dl.acm.org/citation.cfm?doid=2566486.2568024

[118] A. Sadeghi, N. Esfahani, and S. Malek, "Mining the categorized software repositories to improve the analysis of security vulnerabilities," in *Fundamental Approaches to Software Engineering*, S. Gnesi and A. Rensink, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 155–169.

[119] D. Wijayasekara, M. Manic, and M. McQueen, "Vulnerability identification and classification via text mining bug databases," in *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*. Dallas, TX, USA: IEEE, Oct. 2014, pp. 3612–3618. Available: http://ieeexplore.ieee.org/document/7049035/

[120] N. Fenton and M. Nell, "Software Metrics: Roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, p. 14. Available: https://dl.acm.org/doi/pdf/10.1145/336512.336588

[121] IEEE Standards Coordinating Committee and others, "IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Los Alamitos," vol. 269, p. 132, 1990. Available: http://ieeexplore.ieee.org/document/159342/

[122] M. Jimenez, "Evaluating Vulnerability Prediction Models," Ph.D. dissertation, University of Luxembourg, Luxembourg, 2018. Available: https://orbilu.uni.lu/bitstream/10993/36869/1/thesis.pdf

[123] P. K. Shamal, K. Rahamathulla, and A. Akbar, "A study on software vulnerability prediction model," in *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. Chennai: IEEE, Mar. 2017, pp. 703–706. Available: http://ieeexplore.ieee.org/document/8299852/

[124] J. Stuckman, J. Walden, and R. Scandariato, "The Effect of Dimensionality Reduction on Software Vulnerability Prediction Models," *IEEE Transactions on Reliability*, vol. 66, no. 1, pp. 17–37, Mar. 2017. Available: http://ieeexplore.ieee.org/document/7779151/

[125] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic Feature Learning for Predicting Vulnerable Software Components," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 67–85, Jan. 2021. Available: https://ieeexplore.ieee.org/document/8540022/

[126] D. E. Denning, "An Intrusion-Detection Model," in *1986 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, Apr. 1986, pp. 118–118. Available: http://ieeexplore.ieee.org/document/6234848/

[127] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, p. e00938, Nov. 2018. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405844018332067

[128] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 297–308. Available: https://dl.acm.org/doi/10.1145/2884781.2884804

[129] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, Oct. 2017, pp. 2539–2541. Available: https://dl.acm.org/doi/10.1145/3133956.3138840

[130] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-Project Transfer Representation Learning for Vulnerable Function Discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, July 2018. Available: https://ieeexplore.ieee.org/document/8329207/

[131] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021. Available: http://arxiv.org/abs/1807.06756

[132] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019. Available: http://arxiv.org/abs/2001.02334

[133] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building Program Vector Representations for Deep Learning," in *Knowledge Science, Engineering and Management*, vol. 9403, S. Zhang, M. Wirsing, and Z. Zhang, Eds. Cham: Springer

International Publishing, 2015, pp. 547–553. Available: http://link.springer.com/10.1007/978-3-319-25159-2_49

[134] M.-j. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. Melbourne, Australia: International Joint Conferences on Artificial Intelligence Organization, Aug. 2017, pp. 1546–1553. Available: https://www.ijcai.org/proceedings/2017/214

[135] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with AI," *arXiv:1808.09897 [cs, stat]*, Sep. 2018. Available: http://arxiv.org/abs/1808.09897

[136] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang, "Defect Prediction in Android Binary Executables Using Deep Neural Network," *Wireless Personal Communications*, vol. 102, no. 3, pp. 2261–2285, Oct. 2018. Available: https://doi.org/10.1007/s11277-017-5069-3

[137] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep Learning-Based Vulnerable Function Detection: A Benchmark," in *Information and Communications Security*, vol. 11999, J. Zhou, X. Luo, Q. Shen, and Z. Xu, Eds. Cham: Springer International Publishing, 2020, pp. 219–232. Available: http://link.springer.com/10.1007/978-3-030-41579-2_13

[138] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 2012, pp. 380–394. Available: http://ieeexplore.ieee.org/document/6234425/

[139] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A Taint Based Approach for Smart Fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Montreal, QC, Canada: IEEE, Apr. 2012, pp. 818–825. Available: http://ieeexplore.ieee.org/document/6200194/

[140] F. Dustin, "Cyber Grand Challenge (CGC)," 2016. Available: https://www.darpa.mil/program/cyber-grand-challenge

[141] TrailofBits, "DARPA Challenge Binaries on Linux, OS X, and Windows," Trail of Bits, Feb. 2021. Available: https://github.com/trailofbits/cb-multios

[142] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, vol. 45, no. 10, pp. 88–90, Oct. 2012. Available: http://ieeexplore.ieee.org/document/6329885/

[143] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the Reproducibility of Crowd-reported Security Vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 919–935. Available: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-mu.pdf

[144] R. Ferenc, P. Hegedus, P. Gyimesi, G. Antal, D. Ban, and T. Gyimothy, "Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions," in *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 8–14. Available: https://ieeexplore.ieee.org/document/8823747/

[145] J. H. Lau and T. Baldwin, "An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation," in *Proceedings of the 1st Workshop on Representation Learning for NLP*. Berlin, Germany: Association for Computational Linguistics, 2016, pp. 78–86. Available: http://aclweb.org/anthology/W16-1609

[146] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 472–489. Available: https://ieeexplore.ieee.org/document/8835340/

[147] W. Kelvin and Baron, Thomson, *Popular Lectures and Addresses* (Nature Series). Macmillan & Company, 1894, vol. 1. Available: https://books.google.com/books?id=HQ7kkjk1ptoC

[148] G. Smith, "Newton's philosophiae naturalis principia mathematica," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., winter 2008 ed. Metaphysics Research Lab, Stanford University, 2008. Available: https://plato.stanford.edu/archives/win2008/entries/newton-principia/

[149] B. C. Lewis and A. E. Crews, "The Evolution of Benchmarking as a Computer Performance Evaluation Technique," *MIS Quarterly*, vol. 9, no. 1, p. 7, Mar. 1985. Available: https://www.jstor.org/stable/249270?origin=crossref

[150] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Miami, FL: IEEE, June 2009, pp. 248–255. Available: https://ieeexplore.ieee.org/document/5206848/

[151] Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. Pande, "MoleculeNet: A benchmark for molecular machine learning," *Chemical Science*, vol. 9, no. 2, pp. 513–530, 2018. Available: http://xlink.rsc.org/?DOI=C7SC02664A

[152] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to Build a Benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering - ICPE '15*. Austin, Texas, USA: ACM Press, 2015, pp. 333–336. Available: http://dl.acm.org/citation.cfm?doid=2668930.2688819

[153] J. Henning, "SPEC CPU2000: Measuring CPU performance in the New Millennium," *Computer*, vol. 33, no. 7, pp. 28–35, July 2000. Available: http://ieeexplore.ieee.org/document/869367/

[154] K. Huppler, "The Art of Building a Good Benchmark," in *Performance Evaluation and Benchmarking*, vol. 5895, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Nambiar, and M. Poess, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–30. Available: http://link.springer.com/10.1007/978-3-642-10424-4_3

[155] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 344–384, Nov. 1996. Available: http://dl.acm.org/doi/10.1145/235543.235545

[156] N. Antunes and M. Vieira, "On the Metrics for Benchmarking Vulnerability Detection Tools," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Rio de Janeiro, Brazil: IEEE, June 2015, pp. 505–516. Available: https://ieeexplore.ieee.org/document/7266877

[157] R. Kohavi and F. Provost, "Glossary of terms," *Journal of Machine Learning*, vol. 30, pp. 271–274, 1998.

[158] J. Vitek and T. Kalibera, "Repeatability, reproducibility, and rigor in systems research," in *Proceedings of the Ninth ACM International Conference on Embedded Software - EMSOFT '11*. Taipei, Taiwan: ACM Press, 2011, p. 33. Available: http://dl.acm.org/citation.cfm?doid=2038642.2038650

[159] G. J. Matheson, "We need to talk about reliability: Making better use of test-retest studies for study design and interpretation," *PeerJ*, vol. 7, p. e6918, May 2019. Available: https://peerj.com/articles/6918

[160] J. P. Weir, "Quantifying Test-Retest Reliability Using the Intraclass Correlation Coefficient and the SEM," *The Journal of Strength & Conditioning Research*, vol. 19, no. 1, pp. 231–240, 2005.

[161] K. Mather, P. A. P. Moran, and C. A. B. Smith, "Commentary on R. A. Fisher's paper on The Correlation Between Relatives on the Supposition of Mendelian Inheritance." *Population Studies*, vol. 20, no. 3, p. 372, Mar. 1967. Available: https://www.jstor.org/stable/2172683?origin=crossref

[162] R. A. Fisher, "Statistical Methods for Research Workers," in *Breakthroughs in Statistics*, S. Kotz and N. L. Johnson, Eds. New York, NY: Springer New York, 1992, pp. 66–70. Available: http://link.springer.com/10.1007/978-1-4612-4380-9_6

[163] T. E. Oliphant, "Python for Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007. Available: http://ieeexplore.ieee.org/document/4160250/

[164] C. Collberg, T. Proebsting, and A. M. Warren, "Repeatability and Benefaction in Computer Systems Research — A Study and a Modest Proposal," *University of Arizona TR*, vol. 14, no. 4, p. 68, 2015. Available: http://reproducibility.cs.arizona.edu/v2/RepeatabilityTR.pdf

[165] F. Chirigati, M. Troyer, D. Shasha, and J. Freire, "A Computational Reproducibility Benchmark," *IEEE Computer Society Technical Committee on Data Engineering*, p. 6, 2013.

[166] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux journal*, vol. 239, p. 2, 2014.

[167] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning based Vulnerability Detection: Are We There Yet," *IEEE transactions on software engineering*, pp. 1–1, 2021.

[168] N. R. Directorate, "Ghidra," NSA, Online, Apr. 2019. Available: https://ghidra-sre.org/

[169] "BinDiff," Zynamics, June 2021. Available: https://www.zynamics.com/bindiff

[170] Afanador, Kayla, "Ghidraheadless_binexport," 2021. Available: https://github.com/Kayla0x41/ghidraheadless_binexport

[171] Joslin, E. O., "Application Benchmarks: The Key to Meaningful Computer Evaluations," in *ACM 20th National Conference*, EDP Equipment Office, USAF, 1965, pp. 27–37.

[172] MITRE, “About CWE,” Mar. 2021. Available: https://cwe.mitre.org/about/index.html

[173] Richardson, Leonard, “Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation.” Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/#id19

[174] W. McKinney and P.D. Team, “Pandas: Powerful Python data analysis toolkit,” *Pandas—Powerful Python Data Anal Toolkit*, vol. 1625, p. 3411, 2015.

[175] M. Bostock, V. Ogievetsky, and J. Heer, “D^3 Data-Driven Documents,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011. Available: http://ieeexplore.ieee.org/document/6064996/

[176] M. Allen, *The SAGE Encyclopedia of Communication Research Methods*. 2455 Teller Road, Thousand Oaks California 91320: SAGE Publications, Inc, 2017. Available: https://sk.sagepub.com/reference/the-sage-encyclopedia-of-communication-research-methods/

[177] J. P. Leighton, *Encyclopedia of Research Design*. Thousand Oaks: SAGE Publications, Inc., May 2020. Available: http://sk.sagepub.com/reference/researchdesign

[178] K. Dumicic, “Representative samples,” in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1222–1224. Available: https://doi.org/10.1007/978-3-642-04898-2\protect$\relax_5$8

[179] W. Kruskal and F. Mosteller, “Representative Sampling, IV: The History of the Concept in Statistics, 1895-1939,” *International Statistical Review / Revue Internationale de Statistique*, vol. 48, no. 2, p. 169, Aug. 1980. Available: https://www.jstor.org/stable/1403151?origin=crossref

[180] Levy, Paul S. and Lemeshow, Stanley, *Sampling of Populations: Methods and Applications*. John Wiley & Sons., 2013.

[181] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, “LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 60–71. Available: https://ieeexplore.ieee.org/document/8812029/

[182] K. M. Dixit, “Overview of the SPEC Benchmarks,” *IBM Corporation*, p. 33, 1993. Available: http://people.cs.uchicago.edu/~chliu/doc/benchmark/chapter9.pdf

[183] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empirical Software Engineering*, vol. 11, no. 4, pp. 583–611, Nov. 2006. Available: http://link.springer.com/10.1007/s10664-006-9024-2

[184] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, Dec. 2002. Available: http://doi.wiley.com/10.1002/stvr.256

[185] A. Gladston, H. K. Nehemiah, P. Narayanasamy, and A. Kannan, "Test Case Selection Using Feature Extraction and Clustering:," *International Journal of Knowledge-Based Organizations*, vol. 8, no. 2, pp. 18–31, Apr. 2018. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJKBO.2018040102

[186] E. N. Narciso, M. E. Delamaro, and F. D. L. D. S. Nunes, "Test Case Selection: A Systematic Literature Review," *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 04, pp. 653–676, May 2014. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218194014500259

[187] M. Harrold, R. Gupta, and M. Soffa, "A methodology for controlling the size of a test suite," in *Proceedings. Conference on Software Maintenance 1990*. San Diego, CA, USA: IEEE Comput. Soc. Press, 1990, pp. 302–310. Available: http://ieeexplore.ieee.org/document/131378/

[188] G. Kumar and P. K. Bhatia, "Software testing optimization through test suite reduction using fuzzy clustering," *CSI Transactions on ICT*, vol. 1, no. 3, pp. 253–260, Sep. 2013. Available: http://link.springer.com/10.1007/s40012-013-0023-3

[189] A. M. Smith and G. M. Kapfhammer, "An empirical study of incorporating cost into test suite reduction and prioritization," in *Proceedings of the 2009 ACM Symposium on Applied Computing - SAC '09*. Honolulu, Hawaii: ACM Press, 2009, p. 461. Available: http://portal.acm.org/citation.cfm?doid=1529282.1529382

[190] A. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test suite reduction and prioritization with call trees," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering - ASE '07*. Atlanta, Georgia, USA: ACM Press, 2007, p. 539. Available: http://portal.acm.org/citation.cfm?doid=1321631.1321733

[191] M. Bishop, "Vulnerabilities Analysis," in *Proceedings of the Recent Advances in Intrusion Detection*, 1999, pp. 125–136. Available: ftp://ftp.zedz.net/pub/security/development/secure-programming/bishop-1999-vulnerabilities-analysis.pdf

[192] S. Engle, S. Whalen, D. Howard, A. Carlson, E. Proebstel, and M. Bishop, "A Practical Formalism for Vulnerability Comparison," p. 19.

[193] S. Engle, S. Whalen, D. Howard, and M. Bishop, "Tree Approach to Vulnerability Classification," *Department of Computer Science, University of California, Davis, Tech. Rep. CSE-2006-10*, p. 10, 2005. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.6848&rep=rep1&type=pdf

[194] M. Bishop, S. Engle, D. Howard, and S. Whalen, "A Taxonomy of Buffer Overflow Characteristics," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 3, pp. 305–317, May 2012. Available: http://ieeexplore.ieee.org/document/6133295/

[195] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter Notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87–90.

[196] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, pp. 123–147, Mar. 2019. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167404818303808

[197] V. Kouliaridis and G. Kambourakis, "A Comprehensive Survey on Machine Learning Techniques for Android Malware Detection," *Information*, vol. 12, no. 5, p. 185, Apr. 2021. Available: https://www.mdpi.com/2078-2489/12/5/185

[198] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," in *Security and Privacy in Communication Networks*, vol. 127, T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds. Cham: Springer International Publishing, 2013, pp. 86–103. Available: http://link.springer.com/10.1007/978-3-319-04283-1_6

[199] J. Kinder, "Static Analysis of x86 Executables," Ph.D. dissertation, Technische Universität, Darmstadt, Nov. 2010. Available: https://tuprints.ulb.tu-darmstadt.de/2338/

[200] E. Raff, W. Fleming, R. Zak, H. Anderson, B. Finlayson, C. Nicholas, and M. McLean, "KiloGrams: Very Large N-Grams for Malware Classification," *arXiv:1908.00200 [cs, stat]*, July 2019. Available: http://arxiv.org/abs/1908.00200

[201] I. Kwiatkowski, "Manalyze," Aug. 2021. Available: https://github.com/JusticeRage/Manalyze

[202] D. Cavalca and E. Goldoni, "HIVE: An Open Infrastructure for Malware Collection and Analysis," *In proceedings of the 1st workshop on open source software for computer and network forensics*, p. 12, 2008.

[203] A. Zheng and A. Casari, *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, 2018.

[204] H. Liu, *Feature Engineering for Machine Learning and Data Analytics*, 1st ed., G. Dong, Ed. CRC Press, Mar. 2018. Available: https://www.taylorfrancis.com/books/9781351721271

[205] F. Nargesian, H. Samulowitz, U. Khurana, E. B. Khalil, and D. Turaga, "Learning Feature Engineering for Classification," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. Melbourne, Australia: International Joint Conferences on Artificial Intelligence Organization, Aug. 2017, pp. 2529–2535. Available: https://www.ijcai.org/proceedings/2017/352

[206] B. Chernis and R. Verma, "Machine Learning Methods for Software Vulnerability Detection," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*. Tempe AZ USA: ACM, Mar. 2018, pp. 31–39. Available: https://dl.acm.org/doi/10.1145/3180445.3180453

[207] C. Kruegel, R. Lippmann, and A. Clark, *Recent Advances in Intrusion Detection: 10th International Symposium, RAID 2007, Gold Coast, Australia, September 5-7, 2007, Proceedings*. Springer, Aug. 2007.

[208] P. Laud, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, Eds., *Information Security Technology for Applications: 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers* (Lecture Notes in Computer Science). Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7161. Available: http://link.springer.com/10.1007/978-3-642-29615-4

[209] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012. Available: https://security-informatics.springeropen.com/articles/10.1186/2190-8532-1-1

[210] E. Bodden, M. Payer, and E. Athanasopoulos, *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*. Springer, June 2017.

[211] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, p. 156, 2007. Available: http://www.inderscience.com/link.php?id=16865

[212] Z. Yu, H. Hu, C. Bai, K.-Y. Cai, and W. E. Wong, "GUI Software Fault Localization Using N-gram Analysis," in *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*. Boca Raton, FL, USA: IEEE, Nov. 2011, pp. 325–332. Available: http://ieeexplore.ieee.org/document/6113915/

[213] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security and Privacy Magazine*, vol. 5, no. 2, pp. 40–45, Mar. 2007. Available: http://ieeexplore.ieee.org/document/4140989/

[214] K. S. Han, J. H. Lim, B. Kang, and E. G. Im, "Malware analysis using visualized images and entropy graphs," *International Journal of Information Security*, vol. 14, no. 1, pp. 1–14, Feb. 2015. Available: http://link.springer.com/10.1007/s10207-014-0242-0

[215] M. Bat-Erdene, H. Park, H. Li, H. Lee, and M.-S. Choi, "Entropy analysis to classify unknown packing algorithms for malware detection," *International Journal of Information Security*, vol. 16, no. 3, pp. 227–248, June 2017. Available: http://link.springer.com/10.1007/s10207-016-0330-4

[216] P. Bereziński, B. Jasiul, and M. Szpyrka, "An Entropy-Based Network Anomaly Detection Method," *Entropy*, vol. 17, no. 4, pp. 2367–2408, Apr. 2015. Available: http://www.mdpi.com/1099-4300/17/4/2367

[217] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security - CCS '07*. Alexandria, Virginia, USA: ACM Press, 2007, p. 529. Available: http://portal.acm.org/citation.cfm?doid=1315245.1315311

[218] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '08*. Kaiserslautern, Germany: ACM Press, 2008, p. 315. Available: http://portal.acm.org/citation.cfm?doid=1414004.1414065

[219] Y. Shin and L. Williams, "Is Complexity Really the Enemy of Software Security?" in *Proceedings of the 4th ACM Workshop on Quality of Protection*, 2008, pp. 47–50. Available: https://dl.acm.org/doi/pdf/10.1145/1456362.1456372

[220] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*,

vol. 57, no. 3, pp. 294–313, Mar. 2011. Available: https://linkinghub.elsevier.com/retrieve/pii/S1383762110000615

[221] M. O. Shudrak and V. V. Zolotarev, "Improving Fuzzing Using Software Complexity Metrics," in *ICISC 2015*. Springer, 2015, pp. 246–261. Available: https://arxiv.org/pdf/1807.01838

[222] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976. Available: http://ieeexplore.ieee.org/document/1702388/

[223] S. Banerjee and T. Pedersen, "The design, implementation, and use of the ngram statistics package," in *Computational Linguistics and Intelligent Text Processing*, A. Gelbukh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 370–381.

[224] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Systems Technical Journal*, vol. 27, no. 4, pp. 379–443, 1948. Available: https://dl.acm.org/doi/pdf/10.1145/584091.584093

[225] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Security Informatics*, vol. 1, no. 1, p. 1, Dec. 2012. Available: https://security-informatics.springeropen.com/articles/10.1186/2190-8532-1-1

[226] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. Oakland, CA, USA: IEEE Comput. Soc, 2001, pp. 38–49. Available: http://ieeexplore.ieee.org/document/924286/

[227] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* Hong Kong: IEEE, 2004, pp. 41–42 vol.2. Available: https://ieeexplore.ieee.org/document/1342667/

[228] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '04*. Seattle, WA, USA: ACM Press, 2004, p. 470. Available: http://portal.acm.org/citation.cfm?doid=1014052.1014105

[229] G. Canfora, A. D. Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Effectiveness of Opcode ngrams for Detection of Multi Family Android Malware," in *2015 10th International Conference on Availability, Reliability and Security*, 2015, p. 8. Available: https://arts.units.it/bitstream/11368/2864919/1/2015_IWSMA_OpcodeNGramsForAndroidMalwareDetection.pdf

[230] M. Siddiqui, "Data Mining Methods for Malware Detection," Ph.D. dissertation, University of Central Florida, USA, 2008. Available: https://stars.library.ucf.edu/cgi/viewcontent.cgi?article=4709&context=etd

[231] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. No Starch Press, Sep. 2020.

[232] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, p. 10. Available: https://ieeexplore.ieee.org/iel5/4412959/4412960/04413008.pdf

[233] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003, p. 10. Available: https://dl.acm.org/doi/pdf/10.1145/948109.948149

[234] M. Russinovich, "Strings - Windows Sysinternals," June 2021. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/strings

[235] M. Stevanovic, "Linux Toolbox," in *Advanced C and C++ Compiling*, M. Stevanovic, Ed. Berkeley, CA: Apress, 2014, pp. 243–276. Available: https://doi.org/10.1007/978-1-4302-6668-6_12

[236] Y. Goldberg and O. Levy, "Word2vec Explained: Deriving Mikolov et al.'s negative-sampling word-embedding method," *arXiv:1402.3722 [cs, stat]*, Feb. 2014. Available: http://arxiv.org/abs/1402.3722

[237] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. Available: https://www.aclweb.org/anthology/N19-1423

[238] E. Kocagüneli, A. Tosun, A. Bener, B. Turhan, and B. Çağlayan, "Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool," in *SEKE*, 2009, p. 6.

[239] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv:1301.3781 [cs]*, Sep. 2013. Available: http://arxiv.org/abs/1301.3781

[240] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," in *31st International Conference on Machine Learning*. Beijing, China: JMLR.org, 2014, pp. II–1188–II–1196.

[241] M. Mimura and Y. Suga, "Filtering Malicious JavaScript Code with Doc2Vec on an Imbalanced Dataset," in *2019 14th Asia Joint Conference on Information Security (AsiaJCIS)*. Kobe, Japan: IEEE, Aug. 2019, pp. 24–31. Available: https://ieeexplore.ieee.org/document/8827012/

[242] O. Mendsaikhan, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Identification of Cybersecurity Specific Content Using the Doc2Vec Language Model," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Milwaukee, WI, USA: IEEE, July 2019, pp. 396–401. Available: https://ieeexplore.ieee.org/document/8753992/

[243] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of Tricks for Efficient Text Classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Valencia, Spain: Association for Computational Linguistics, 2017, pp. 427–431. Available: http://aclweb.org/anthology/E17-2068

[244] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, "Handling imbalanced datasets: A review," *GESTS International Transactions on Computer Science and Engineering*, vol. 30, no. 1, pp. 25–36, 2006.

[245] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, pp. 249–259, Oct. 2018. Available: http://arxiv.org/abs/1710.05381

[246] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection," *IEEE Transactions on Fuzzy Systems*, pp. 1–1, 2019. Available: https://ieeexplore.ieee.org/document/8930093/

[247] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, June 2002. Available: https://www.jair.org/index.php/jair/article/view/10302

[248] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

[249] M. Kubat, *An Introduction to Machine Learning*. Cham: Springer International Publishing, 2017. Available: http://link.springer.com/10.1007/978-3-319-63913-0

[250] E. Alba, G. Luque, F. Chicano, C. Cotta, D. Camacho, M. Ojeda-Aciego, S. Montes, A. Troncoso, J. C. R. Santos, and R. Gil-Merino, Eds., *Advances in Artificial Intelligence: 19th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2020/2021, Málaga, Spain, September 22–24, 2021, Proceedings* (Lecture Notes in Artificial Intelligence). Springer International Publishing, 2021. Available: https://www.springer.com/gp/book/9783030857127

[251] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, Sep. 1999. Available: https://dl.acm.org/doi/10.1145/331499.331504

[252] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, Sep. 1967. Available: http://link.springer.com/10.1007/BF02289588

[253] M. Waskom, M. Gelbart, O. Botvinnik, J. Ostblom, P. Hobson, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. Warmenhoven, J. B. Cole, J. D. Ruiter, J. Vanderplas, S. Hoyer, C. Pye, A. Miles, C. Swain, K. Meyer, M. Martin, P. Bachant, E. Quintero, G. Kunter, S. Villalba, Brian, C. Fitzgerald, C. Evans, M. L. Williams, D. O'Kane, T. Yarkoni, and T. Brunner, "Mwaskom/seaborn: V0.11.1 (December 2020)," Zenodo, Dec. 2020. Available: https://zenodo.org/record/592845

[254] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Santa Barbara CA USA: ACM, June 2016, pp. 266–280. Available: https://dl.acm.org/doi/10.1145/2908080.2908126

[255] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," *arXiv:1908.10084 [cs]*, Aug. 2019. Available: http://arxiv.org/abs/1908.10084

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California