Selján Gábor

Machine Learning in Security Vulnerability Research

Doctoral School of Economics, Business and Informatics

Supervisor: Dr. Racskó Péter, Csc

© Selján Gábor

Corvinus University of Budapest
Doctoral School of Economics, Business and Informatics

Machine Learning in Security Vulnerability Research

Seed Selection with Binary Classification for Directed Fuzzing

Doctoral dissertation

Selján Gábor

Budapest, 2025

This dissertation is dedicated to my parents.

Abstract

AI has emerged as one of the most transformative technologies in cybersecurity. However, its capabilities are accessible to both defenders and adversaries, making it imperative to recognize its dual-use nature. The rapid evolution of cyberthreats in an increasingly digital world demands innovative approaches to vulnerability research. As cybercriminals develop more sophisticated techniques, traditional security measures often struggle to keep up with them. Although ML offers novel methodologies for improving cybersecurity, research in certain areas remains limited. This observation served as the main motivation for the present research project.

This dissertation explores the application of ML in cybersecurity, with a specific focus on offensive security. To examine existing work on the roles of AI and ML in vulnerability discovery, I conducted a targeted review of the literature, which indicated that fuzz testing, a method in which an automated tool evaluates software for security flaws using random input data, is a promising area for further exploration.

To improve the effectiveness of fuzzers, I propose an ML-based seed selection method built on the concept that known bugs provide valuable insights to guide fuzzers toward the security-sensitive parts of the target program. For a thorough examination of this issue, I constructed a NN for binary classification that uses knowledge derived from past fuzzing results to improve seed selection for subsequent test campaigns.

I conducted a case study to investigate the practical application of this approach in real-world scenarios. The Windows graphics component represents a valuable attack surface due to its core functionality being deeply integrated into the operating system, as any flaws in this component could have significant security implications. Metafiles serve as an ideal input format for fuzz testing due to their compact size, which enables efficient mutation and rapid execution of testcases. I evaluated the proposed approach by running experiments on a legacy version of the Windows operating system using various seed selection strategies. Comparative analysis indicated that seeds selected according to the model's predictions effectively directed the fuzzer toward code paths associated with known security issues.

In summary, this dissertation reaffirms the broad potential of applying ML in this context. The discovery of several zero-day vulnerabilities confirms the practical value of the proposed seed selection strategy. Moreover, the functional prototype of `SeedClass` and the accompanying dataset, together with the data processing scripts, are available on GitHub[1]. The dataset, entirely collected and curated by the author, includes samples for more than 50 unique bugs to support further analysis of vulnerabilities in the Windows graphics component and promote research on ML-assisted fuzz testing.

[1] The code repository of the `SeedClass` project can be found at https://github.com/sgabe/SeedClass.

Acknowledgements

My professional path in security vulnerability research began with the discovery of my first zero-day vulnerability in a media player. This early accomplishment introduced me to this challenging field and paved the way for my future career. This dissertation marks the culmination of a long and rewarding journey fueled by curiosity, perseverance, and the unwavering support of many.

I would like to express my gratitude to Dr. Erdődi László of Óbuda University, who supervised my undergraduate studies, for his mentorship and guidance in helping me take the first steps into the field of cybersecurity. Without his initial support and encouragement, I might have pursued a more conventional career path instead of discovering my true calling in research.

I am deeply grateful to Dr. Racskó Péter of Corvinus University of Budapest, who supervised my graduate and doctoral studies, for his openness to the topic and his steady support throughout my research. His foresight in integrating machine learning with vulnerability research provided a vital turning point in my work. His guidance elevated the quality of my research and broadened my perspective on the potential intersections of these dynamic fields.

I am also profoundly grateful to Dr. Kő Andrea of Corvinus University of Budapest, whose support was a significant pillar throughout my academic endeavors during my graduate and doctoral studies. Her encouragement and expertise shaped the direction of my research and contributed greatly to my professional growth.

I would also like to extend my thanks to my friends and colleagues who showed interest in my research and shared their feedback during our conversations, which played an important role in refining my ideas and broadening the scope of my work. Their willingness to challenge my assumptions and offer practical insights made this process even more impactful.

Finally, I extend my deepest gratitude to my family for their unwavering support, which has been instrumental throughout this endeavor. I owe an enduring debt of gratitude to my father, whose passion for informatics ignited my curiosity and set me on this path from an early age. I truly appreciate my mother's understanding of my passion, which has been a source of great comfort along the way. Special thanks go to Selján Péter, my twin brother, whose patience and guidance helped me navigate the inevitable difficulties of this journey.

Table of Contents

List of Figures iii

List of Tables v

List of Listings vii

List of Acronyms ix

1. Introduction 1
1.1 Overview and justification . . . 1
1.2 Key research questions . . . 6
1.3 Directions of the research . . . 8
1.4 Artifacts of the research . . . 13

2. Background 15
2.1 AI in cybersecurity . . . 15
2.2 Traditional fuzz testing . . . 26
2.3 Machine learning overview . . . 35

3. Theoretical concept 45
3.1 Related work . . . 45
3.2 Proposed framework . . . 53

4. Experimental implementation 61
4.1 Generating seed inputs . . . 61
4.2 Fuzzing the Windows graphics component . . . 65
4.3 Selecting seed inputs with classification . . . 72

5. Empirical assessment 87
5.1 Setup and execution . . . 87
5.2 Results and insights . . . 88
5.3 Vulnerabilities discovered . . . 91

6. Results and conclusion 103
6.1 Summary . . . 103
6.2 Outputs . . . 104
6.3 Limitations . . . 105
6.4 Conclusion . . . 107

Bibliography 109

Appendices 137

A. Assessment results 137

B. Unique crash samples 139

C. Stack backtraces 140

D. Fuzzing harness 142

E. Model prototype 144

F. Helper scripts 148

List of Figures

1 Types of artificial intelligence (author's own work) 3
2 Types of machine learning (author's own work) 4
3 Search and selection process overview (Molleri et al., 2016) 9
4 The design science framework (based on Wieringa, 2014) 10
5 Examples of research strategies (Johannesson and Perjons, 2021) 11
6 The engineering cycle as a problem-solving process (Wieringa, 2014) . . . 12
7 Phases of the cyber kill chain model (based on Hutchins et al., 2011) 16
8 The AI-based cyberthreat framework (based on Kaloudi and Li, 2020) . . . 17
9 General fuzzing process (Y. Wang et al., 2020) 22
10 Data preprocessing workflow (Y. Li et al., 2019) 24
11 Vulnerability-oriented fuzzing workflow (Y. Li et al., 2019) 25
12 The veto model used by Augmented-AFL (Rajpal et al., 2017) 25
13 Various types of machine learning (ML) techniques (Sarker, 2021) 35
14 Overall architecture of the concept (author's own work) 56
15 EMF metafile versions (Microsoft, 2022) 63
16 Overview of `FormatFuzzer` (Dutra et al., 2023) 64
17 Architecture of `DynamoRIO` (based on Bruening, 2004) 66
18 General workflow of `AFL` (Pang et al., 2021) 67
19 WinAFL Pet deployment schema (author's own work) 70
20 Two-dimensional PCA scatter plot of the data (author's own work) 78
21 t-SNE visualization of the similarity distribution in data (author's own work) 79
22 Learning progress over epochs (author's own work) 84
23 Alternative input encoding process (based on M. Wang and Robinson, 2023) 107

List of Tables

1 Related work summary comparison (based on Kaloudi and Li, 2020) 5
2 Common white-box, gray-box and black-box fuzzers (J. Li et al., 2018) . . . 29
3 Comparison of generation- and mutation-based fuzzers (J. Li et al., 2018) . . 33
4 Mutators used by `AFL` (Karamcheti et al., 2018) 33
5 Various types of ML techniques with example (Sarker, 2021) 36
6 Typical classification MLP architecture (Géron, 2022) 59
7 Summary of fuzzing output statistics 89
8 Distribution of unique bugs identified 90
9 Per fuzzer statistics for the *randomization* strategy 137
10 Per fuzzer statistics for the *minimization* strategy 137
11 Per fuzzer statistics for the *prediction* strategy 138
12 Per fuzzer statistics for the *combined* strategy 138
13 Summary of unique crash samples identified during the research 139

List of Listings

1 Storing duplicate crash samples separately . . . 69
2 Testing harness for fuzzing metafiles . . . 72
3 Verification script for *valid* samples . . . 74
4 Data loading and preprocessing . . . 75
5 Visualizing the data with a PCA scatter plot . . . 77
6 Visualizing the data with a t-SNE scatter plot . . . 78
7 Normalizing the training data with the Min-Max method . . . 79
8 Building the neural network model . . . 80
9 Training the neural network model . . . 81
10 Plotting the learning history of the model . . . 82
11 Making predictions with the trained model . . . 85
12 Decompiled source code of the `bCheckBitmapInfoHeader()` function . . . 92
13 Pseudocode of the `bMetaGetDIBInfo()` function . . . 93
14 Pseudocode of the `EmfEnumState::CreateModifiedDib` function . . . 95
15 Pseudocode of the `ValidateBitmapInfo()` function . . . 96
16 Pseudocode of the `MfEnumState::OutputDIB()` function . . . 97
17 Pseudocode of the `EmfEnumState::ModifyRecordColor` function . . . 98
18 Pseudocode of the `StartDocWImpl()` function . . . 98
19 Example `DOCINFO` structure . . . 99
20 Pseudocode of the `MfEnumState::OutputDIB()` function . . . 99
21 Pseudocode of the `GetDibNumPalEntries()` function . . . 100
22 Stack trace of `EMR_SETPIXELV` (CVE-2022-34728) . . . 140
23 Stack trace of `EMR_CREATEDIBPATTERNBRUSHPT` (CVE-2022-26934) . . . 140
24 Stack trace of `EMR_STRETCHDIBITS` (CVE-2022-21915) . . . 140
25 Stack trace of `EMR_BITBLT` (CVE-2022-29112) . . . 141
26 Stack trace of `EMR_STARTDOC` (CVE-2022-35837) . . . 141
27 Main functionality of the test harness used for fuzzing Windows GDI . . . 142
28 Main functionality of the `SeedClass` prototype . . . 144
29 `CrashCollect` - Gather unique crash samples from output directories . . . 148
30 `CrashSort` - Triage crash samples using BugID . . . 149
31 `CrashDistribute` - Optimize bug distribution for a balanced dataset . . . 150
32 `SeedFilter` - Optimize valid samples based on byte-level variance . . . 151
33 `SeedGroup` - Generate random, shuffled sample groups . . . 152
34 `SeedSplit` - Divide large samples into smaller pieces . . . 153
35 `SeedValidate` - Verify valid samples by opening them in Paint . . . 154

List of Acronyms

ACFG attributed control flow graph . . . 24
AGI artificial general intelligence . . . 3
AI artificial intelligence . . . 1
ANN artificial neural network . . . 37
API Application Programming Interface . . . 30
APT advanced persistent threat . . . 3
ASI artificial super intelligence . . . 3
ASLR address space layout randomization . . . 99
AST abstract syntax tree . . . 49
C2 command and control . . . 17
CFG control flow graph . . . 24
CGF coverage-guided graybox fuzzing . . . 52
CHI Calinski–Harabasz index . . . 40
CNN convolutional neural network . . . 36
CVE Common Vulnerabilities and Exposures . . . 92
DIB device-independent bitmap . . . 92
DL deep learning . . . 2
DM data mining . . . 3
DNN deep neural network . . . 23
DS design science . . . 9
DSR design science research . . . 9
EMF Enhanced Metafile . . . 7
EMF+ Enhanced Metafile Plus . . . 63
GAN generative adversarial network . . . 40
GDI Graphics Device Interface . . . 61
GDI+ Graphics Device Interface Plus . . . 61
IDS intrusion detection system . . . 19
IoT Internet of Things . . . 29
IPS intrusion prevention system . . . 19
IR intermediate representation . . . 30
LLM large language model . . . 16
LLVM Low Level Virtual Machine . . . 52
LSTM long short-term memory . . . 51
MAE mean absolute error . . . 39
MICE Metafile Image Code Execution . . . 62
ML machine learning . . . 1

MLP multi-layer perceptron . . . 58
MSE mean squared error . . . 39
MSRC Microsoft Security Response Center . . . 92
NAS network-attached storage . . . 70
NLP natural language processing . . . 35
NN neural network . . . 1
PCA principal component analysis . . . 39
PCSG probabilistic context-sensitive grammar . . . 49
PDF Portable Document Format . . . 49
PUT program under test . . . 21
RL reinforcement learning . . . 19
RNN recurrent neural network . . . 36
SP spear-phishing . . . 18
SVG Scalable Vector Graphics . . . 62
SVM support vector machine . . . 37
t-SNE t-distributed stochastic neighbor embedding . . . 43
TAR technical action research . . . 12
TTC time to crash . . . 89
TTP tactic, technique and procedure . . . 5
WMF Windows Metafile . . . 62

1. Introduction

1.1 Overview and justification

1.1.1 Foundations and rationale

The adoption of artificial intelligence (AI) has expanded considerably in recent years, fundamentally transforming traditional methodologies and introducing innovative solutions in various industries. This paradigm shift has also led to the increased deployment of AI-driven technologies by both cybersecurity defenders and adversaries. Although these advances improve the ability of defense professionals to detect and mitigate cyberattacks, they simultaneously equip sophisticated attackers with tools to execute increasingly complex and effective cyberattacks (Schneier, 2018).

This study is grounded in existing research on machine learning (ML) and fuzz testing, seeking to identify alternative methodologies and introduce innovative solutions to recurring challenges guided by the insights of previous studies. The following paragraphs summarize the four main logical units of the dissertation.

First, this investigation examines how malicious actors[2] leverage AI and evaluates its impact on cybersecurity. It highlights the increasing sophistication of harder-to-detect, AI-driven cyberattacks and provides an overview of existing research on attacker methodologies and their impact. Using AI to analyze large amounts of data, adversaries can identify software vulnerabilities and plan highly effective attacks. The increasing effectiveness of AI-driven threats highlights the necessity for defenders to implement AI-based solutions to counter them.

Second, the dissertation investigates research on the identification of zero-day[3] security vulnerabilities via fuzz testing, an approach in which an automated tool evaluates software for security flaws using random input data. Although this method has proven effective in discovering software bugs, it remains time-consuming and resource-intensive, particularly for large codebases (Evans et al., 2011). To address this difficulty, AI-assisted fuzzing presents a promising optimization strategy as leveraging ML algorithms may reduce the time and resources required to find bugs.

Third, the research outlines an ML-assisted seed selection strategy[4] that uses a neural network (NN) for a binary classification model to assess the similarity of the provided

[2] A malicious actor is an individual or a group that deliberately engages in harmful activities to exploit vulnerabilities, disrupt operations, or compromise security within an information system.

[3] Zero-day vulnerabilities are bugs for which no patch or fix has been publicly released (Libicki et al., 2015).

[4] A seed selection strategy in fuzzing refers to the method used to choose initial input samples that guide the fuzzing process.

input samples to known crash instances. By examining the characteristics of accurately labeled samples, the model predicts which inputs are more likely to reach states associated with known program crashes, thereby making the fuzz testing process more focused.

Fourth, the study employs a practical approach by evaluating the impact of the proposed ML-assisted seed selection method on directed fuzz testing. It presents a case study in which fuzzing the Windows graphics component using metafiles led to the discovery of several zero-day vulnerabilities. This study aims to demonstrate the practical applicability and real-world impact of the proposed method.

The aforementioned main logical units of the dissertation are organized into six main chapters that collectively establish and validate a comprehensive research framework. *Chapter 1* establishes the context by introducing key research questions and describing the methodology to conduct the literature review and develop multiple software artifacts to provide practical solutions to real-world problems. *Chapter 2* presents an overview of the various applications of AI in cybersecurity, traditional fuzz testing methods, and relevant ML concepts. *Chapters 3* and *Chapter 4* transition from theory to practice by exploring related work, followed by proposing a conceptual framework for an alternative seed selection method and detailing the implementation of an experimental prototype. Finally, *Chapters 5* and *Chapter 6* provide an empirical assessment of the experiments and conclude by summarizing the research findings, discussing the vulnerabilities discovered and offering a brief outlook on future research opportunities, while reflecting on the limitations and implications of the study.

1.1.2 Concepts, techniques and threats

According to Conn (2015), AI encompasses the science and engineering of developing ”smart” machines that imitate intelligent human behavior, appearing autonomous when performing tasks without pre-programming. Emmert-Streib et al. (2020) explains that deep learning (DL), a subset of ML, employs NNs modeled after the human brain to process large amounts of data. In essence, AI focuses on creating systems that replicate human intelligence, while ML involves designing algorithms that allow machines to learn from experience.

Despite the absence of a universally accepted definition, various interpretations have emerged, generating ongoing debate. Simmons and Chappell (1988) defines AI as ”*the behavior of a machine that, if replicated by a human, would be considered intelligent*”. AI is designed to improve the ability of computers and machines to perform tasks comparable to those performed by humans (Rebstock, 2025). This dissertation aligns with this perspective.

Kurzweil (2014) introduced the term ”*narrow AI*”, also known as ”*weak AI*”, to describe systems capable of performing specific ”*intelligent*” tasks, such as facial recognition, Internet searches or autonomous driving (Csáki, 2022). According to Goertzel

(2014), even minor variations in circumstances often require human intervention to maintain the level of intelligence of a system. Meanwhile, as Figure 1 shows, the industry continues to pursue the development of general AI, referred to as ”strong AI” or artificial super intelligence (ASI), which aims to surpass human intelligence in creativity, general knowledge and problem solving (Jajal, 2018). The prospect of AI exhibiting capabilities previously exclusive to highly skilled individuals has raised concerns about its potential social impact (Acemoglu & Restrepo, 2018; Smith & Shum, 2018; Yudkowsky, 2008). However, some studies question this possibility; for example, Świechowski (2022) outlines fundamental limitations that currently prevent deep learning NNs from achieving artificial general intelligence (AGI) (Dela Cruz & Rubi, 2022).

In recent years, discussions about AGI and ASI have noticeably changed from debating abstract theoretical definitions to focusing on measurable achievements demonstrated by standard tests and benchmarks. Researchers are now focusing on practical advances and immediate results in specific tasks that help to refine performance and address ethical and safety considerations (Cassidy et al., 2022). This transition highlights the trend of prioritizing progress over conceptual debates, as practical results provide clearer insight into the capabilities and future risks of AI. Morris et al. (2024) offers a framework for evaluating AGI that aligns with this change towards measurable performance.

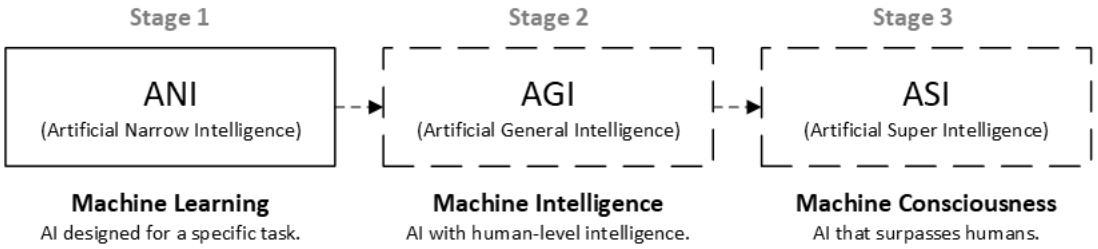


Figure 1: Types of artificial intelligence (author’s own work)

Fayyad et al. (1996) defines data mining (DM) as ” *the general method of discovering useful information from data*”. Sometimes, people confuse the distinction between DM and ML by using the terms interchangeably. However, Kubovic et al. (2018) notes that ML has significantly affected various fields, including cybersecurity, as it represents a step toward fully autonomous AI systems. Advancements in ML-driven security solutions have strengthened corporate defenses by improving scanning engines, accelerating detection times, and improving anomaly identification, particularly in relation to advanced persistent threats (APTs)[5] and emerging cyber risks.

AI has emerged as one of the most transformative technologies in cybersecurity. However, its capabilities are accessible to both defenders and adversaries, making it impera-

[5]A highly skilled and well-resourced adversary that persistently adapts to defensive measures over time to achieve its objectives, such as espionage, data exfiltration, or operational disruption (Dempsey et al., 2011).

tive to recognize its dual-use nature. Ensuring responsible and ethical deployment of AI requires awareness of its potential for both security enhancement and exploitation. Although computer technology surpasses human capabilities in terms of reach, speed, and scale, human expertise remains superior in reasoning and strategic thinking. Effectively countering AI-driven cyberattacks requires a thorough understanding of the tactics used by malicious actors.

Cybercriminals and other malicious actors recognize the advantages of AI and actively seek to integrate it into their operations, thereby increasing the complexity of detecting, monitoring and preventing attacks. ML algorithms play a crucial role in classifying and evaluating digital samples by analyzing patterns, assessing similarities, and assigning probability scores to categorize each item. Effective training of these models requires a substantial dataset of accurately labeled samples, ensuring optimal classification performance.

Given these challenges, the different ML approaches may contribute to cybersecurity in their own way (Kabanda, 2020). The following paragraphs outline the key differences between supervised, unsupervised, and semi-supervised learning while also hinting at how they could contribute to addressing cybersecurity challenges.

Supervised ML refers to the integration of human oversight, where the labeled data guide algorithmic training to produce more accurate and reliable results by optimizing the model parameters to minimize the discrepancy between the predicted and target outputs (Jo, 2023).

Unsupervised ML algorithms operate without fully labeled training data, instead model parameters are optimized to identify patterns and relationships within datasets that human analysts may overlook (Jo, 2023). Although these algorithms excel at detecting underlying structures and similarities, they do not inherently learn to differentiate between ”good” and ”bad” outcomes, as they lack predefined classification criteria.

Semi-supervised ML falls between supervised and unsupervised learning. As Kubovic et al. (2018) summarizes, it relies on a combination of labeled and unlabeled data to train the algorithm. Experts then refine the model output to achieve the desired level of accuracy (Camargo et al., 2020). Figure 2 below illustrates the various types of ML.

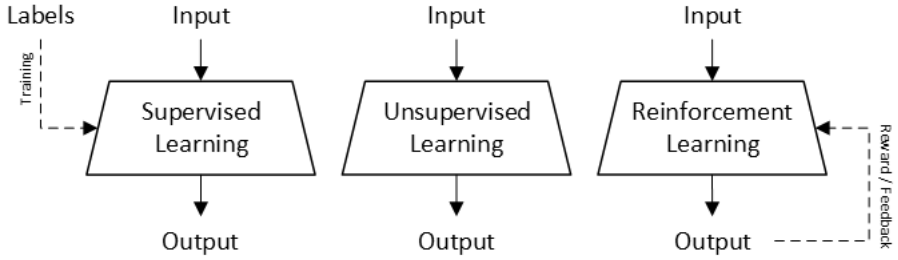


Figure 2: Types of machine learning (author’s own work)

Ethical hackers[6] play an important role in preventing cyberthreats by identifying vulnerabilities before malicious actors can exploit them. However, they have yet to fully embrace this responsibility in the context of AI technologies. Although many stakeholders focus on leveraging AI to enhance defense mechanisms and detect security breaches, it is urgent to increase efforts to proactively uncover new vulnerabilities to stay ahead of evolving attack strategies.

As Kaloudi and Li (2020) noted, existing literature reviews and surveys primarily examine the intentional or accidental development of harmful AI, focusing on unethical AI schemes rather than its use as an offensive cybersecurity tool, as summarized in Table 1. However, effectively countering emerging threats requires a proactive approach, one that emphasizes strategic anticipation rather than reactionary defense. Advancing research into adversarial tactic, technique and procedures (TTPs)[7] could equip the information security community with the knowledge needed to mitigate evolving cybersecurity threats and position defenders ahead in this ongoing race (Girhepuje et al., 2024; Valencia, 2024).

Article	Year	Contribution
Risk Sources and Risk Management Measures in Support of Standards for General-Purpose AI Systems	2024	Provides a catalog of GPAI risks and management measures to support AI governance.
Towards trustworthy Artificial Intelligence: Security risk assessment methodology for Artificial Intelligence systems	2023	Proposes an automated security risk assessment methodology and tool to enhance the robustness of AI-integrated digital systems against emerging cybersecurity threats.
Multilayer framework for Good Cybersecurity Practices for AI	2023	Proposes a 3-layered framework to provide a step-by-step approach for securing AI systems
The Emerging Threat of AI-driven Cyber Attacks: A Review	2022	Investigation of the emerging threat of AI-powered cyberattacks and provides insights into malicious use of AI in cyberattacks.
Towards Deep Learning: A Review on Adversarial Attacks	2021	Discusses the vulnerability of ML algorithms to adversarial attacks and suggests reviewing works and providing recommendations for future work.
The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation	2018	This is a summary of what the workshop and more studies have shown about how threats might change in three areas of security: physical, digital, and political.
Artificial Intelligence Crime: An Interdisciplinary Analysis of Foreseeable Threats and Solutions	2018	Finding possible artificial intelligence crime areas where AI can be used to help with possible criminal actions in each crime area.
Artificial Intelligence Safety and Cybersecurity: A Timeline of AI Failures	2016	Investigation of previous instances of AI blunders caused by the intelligence of AI systems, with the expectation that more will follow.
Unethical Research: How to Create a Malevolent Artificial Intelligence	2016	Guidelines for purposefully creating a hostile AI system to raise awareness since the intentionally evil design of intelligent robots could be even more damaging.
Responses to Catastrophic AGI Risk: A Survey	2015	Literature overview on AI risk, with a focus on the field of AI safety engineering.

Table 1: Related work summary comparison (based on Kaloudi and Li, 2020)

[6]Ethical hackers are professionals authorized by an organization to assess and strengthen their security defenses. They simulate adversarial attacks while strictly following legal and ethical boundaries to safeguard confidentiality and organizational assets (Jaquet-Chiffelle & Loi, 2020).

[7]TTPs define the structured behavioral patterns of threat actors, with tactics defining high-level objectives, techniques detailing methods used to achieve them and procedures outlining specific execution steps within a technique (Ross et al., 2022).

Identifying and addressing vulnerabilities in AI-powered systems is critical to preserving their security and integrity. Given that AI not only poses new risks (Turchin, 2015; Yudkowsky, 2008), but also creates a new attack surface, it is vital to implement proactive defense measures to address emerging threats (Marshall et al., 2022). Although there is currently no clear evidence linking recent cyberattacks to AI or ML technologies, the growing concern within the industry highlights the need to anticipate and mitigate potential risks.

The malicious use of AI accelerates attack speed and increases the success rates, significantly enhancing the capabilities of adversaries. This research aims to examine how AI technologies can support malicious activities and identify areas where their exploitation may have the most significant impact. There is an inherent overlap between attackers and defenders in this context. An effective approach to recognizing emerging AI-driven threats is to replicate adversarial behaviors and tactics. By simulating hypothetical attacks, defenders can gain valuable insights into the cybersecurity implications of AI and proactively mitigate evolving risks. Furthermore, this strategy raises awareness of potential AI-enabled threats, reinforcing defensive preparedness.

1.2 Key research questions

The increasing prevalence of cyberattacks points to the value of robust cybersecurity measures. According to Kim and Solomon (2021), cybersecurity involves protecting individuals, businesses, organizations, and governments in cyberspace, with all actors responsible for protecting their information and data. Hamadah and Aqel (2020) examined various AI-based cybersecurity models and concluded that AI and ML introduce new possibilities and provide more effective solutions to address security challenges.

1.2.1 Research areas and scope

This research evaluates the impact of AI on cybersecurity (Jimmy, 2021) by focusing on three main areas that address distinct aspects of the subject. At this stage, I introduce these broader research areas without revealing specific research questions. Later sections will present a more in-depth discussion, including the precise research questions guiding this study.

The first research area focuses on the application of *AI in cybersecurity*, with a particular emphasis on new capabilities such as advanced threat detection, real-time monitoring, and improved security measures that strengthen defenses against cyberthreats. The second research area focuses on the applications of AI in the exploitation phase of the cybersecurity kill chain. Specifically, it examines how *AI in security research* can help identify software vulnerabilities that real threat actors frequently exploit. The third research area

examines the role of *AI in fuzz testing*, a widely recognized software testing technique used by large companies to identify bugs and security vulnerabilities (Arya et al., 2019; Godefroid et al., 2012). By integrating AI-driven methodologies, fuzz testing can be optimized to improve efficiency and uncover deeper, previously undetected system weaknesses. Finally, the most important practical research area of this dissertation is the development and implementation of an initial working prototype of a tool that uses ML methods to optimize the seed selection process in fuzz testing.

1.2.2 Methodological framework and contextual focus

The dissertation explores cybersecurity, with a particular focus on the application of AI in offensive security. Hence, these research areas provide a foundation for the examination and define the scope of further exploration. The following sections outline the research directions and methodologies used throughout the research.

First, I conducted an initial literature review to assess the current state of AI applications in cybersecurity, with the aim of laying a solid foundation. The review employed the snowball methodology, which involved searching for specific terms on Google Scholar, selecting relevant papers, and analyzing their referenced work to expand the scope of the investigation. The reviewed literature highlights a research gap in the use of ML algorithms in offensive security. Further exploration narrows the research focus to improving the effectiveness of fuzzing by refining various steps in the methodology.

Second, this study focuses on the Windows operating system as the target platform, identifies the Windows graphics component as an especially interesting attack surface, and examines the processing of metafile image formats to accurately reflect real-world offensive security research (Jurczyk, 2016). Although the Enhanced Metafile (EMF) image format has declined in the past two decades in favor of raster-based representations, it remains a relevant attack vector and an area of interest for security researchers. Despite its official deprecation, Internet Explorer remains a built-in Windows component that continues to support the EMF format, which serves as the native image storage format in Office and plays a crucial role in print spooling.

Building on existing research and acknowledging the growing interest in this domain, this analysis considers research directions that align with the exploitation phase of AI-driven cyberthreats. To compromise a system, adversaries must either obtain a zero-day exploit or conduct independent vulnerability research (Drake & Coley, 2016; Huiras et al., 2022) to identify security flaws and develop the corresponding exploits. Once attackers establish a foothold within a system, remaining undetected for extended periods is crucial to achieve their objectives.

1.2.3 Research questions and goals

This research starts with a broad examination of AI applications in cybersecurity, before gradually refining its focus on ML in vulnerability research (Abraham & de Vel, 2017), fuzz testing (Saavedra et al., 2019), and seed selection (Cheng et al., 2019). To explore these challenges, three primary research inquiries are introduced that reflect this narrowing of scope. These questions aim to provide deeper insight into the role of AI in addressing cyberthreats.

- **RQ1**: The first research question opens up the realm of AI in cybersecurity, specifically focusing on its current application in the field:
  What are the current applications of AI and ML technologies in the field of cybersecurity?
- **RQ2**: The second research question focuses on AI-assisted vulnerability research:
  How can we utilize AI and ML technologies in offensive security, especially during the exploitation phase of the kill chain, to discover new zero-day vulnerabilities?
- **RQ3**: The third research question focuses on AI-assisted fuzz testing:
  How can AI and ML technologies improve fuzz testing?
- **RG1**: The final goal of the research is creating an ML-based fuzzing tool:
  How can we implement an AI-assisted program to support the initial seed selection phase of the fuzz testing process?

1.3 Directions of the research

1.3.1 AI-aided vulnerability research

For the literature review in this dissertation, I adopted a methodology initially employed by Molleri et al. (2016) and later refined by Kaloudi and Li (2020). This approach ensures both comprehensiveness and reliability, minimizing the risk of overlooking critical aspects. The process consists of identifying the research topic, formulating a structured plan to gather relevant documents, and reviewing the findings.

The first step in this research is to define clear objectives that establish the study's scope and provide a foundation for formulating research questions. The primary research inquiry that guided my investigation is the following: *How can the application of AI and ML technologies enhance cybersecurity?*

I conducted an iterative search and selection process to retrieve relevant literature in three stages, as illustrated in Figure 3. In the first stage, I focused on selecting articles that examine the application of AI and ML technologies in cybersecurity. This approach allowed me to compile a collection of relevant studies, providing the foundation for analyzing the subject matter.

I started the review by manually searching Google Scholar to compile an initial collection of relevant papers, ensuring that the selection process remained unbiased toward any specific publisher or institution. Due to the novelty of the field and the inherent challenges in identifying pertinent studies, a manual search was necessary to retrieve comprehensive and reliable sources.

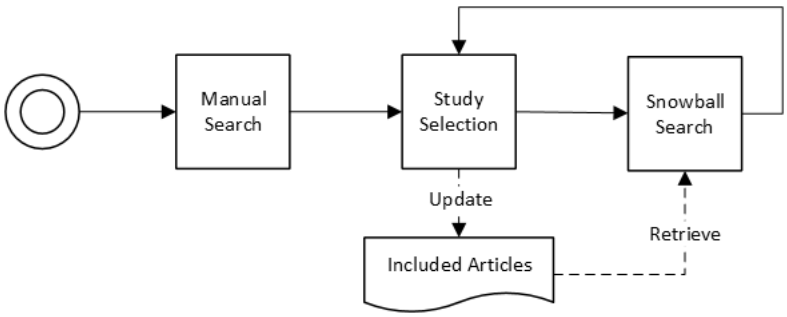


Figure 3: Search and selection process overview (Molleri et al., 2016)

To identify relevant articles, I carefully reviewed each search result to ensure accuracy and completeness. For each selected study, the title and abstract were examined to assess its relevance to the research questions. Following this initial screening, I conducted a detailed analysis to ensure that the articles chosen aligned with the objectives of my research.

Focusing on fuzz testing methodologies, I conducted an in-depth analysis of case studies on AI-assisted vulnerability research to identify various strategies employed by researchers. By reviewing papers on different fuzz testing approaches, I evaluated their objectives and the integration of AI and ML technologies to improve their effectiveness. Through a comprehensive examination of the collected data, key insights were identified while carefully considering assumptions, validity and credibility to ensure the robustness of the findings.

1.3.2 AI-powered fuzz testing

This work proposes an AI-assisted tool to identify and mitigate security vulnerabilities in software by optimizing the process of seed selection in the fuzz testing methodology. To achieve this goal, it is essential to both develop an artifact, such as a NN model for binary classification, and evaluate its ability to guide fuzzers toward parts of the program that are known to be security sensitive based on previously observed program crashes.

To structure the creation and assessment of this artifact, the design science (DS) paradigm provides a broad discipline for conceiving, constructing and utilizing artifacts to solve practical challenges, ranging from routine design to exploratory activities. Wieringa (2014) defines DS as ”*the design and investigation of artifacts in context*”. Within this discipline, design science research (DSR) emerges as a formal research paradigm characterized by a

structured and iterative process that produces both useful artifacts and knowledge that is generalizable.

DSR emphasizes systematic inquiry, explicit research guidelines, and rigorous evaluation. When combined with case study methods, it allows researchers to apply and test artifacts in real-world settings, thereby generating insights into their practical utility and limitations. According to Vaishnavi et al. (2004), DS provides the expertise to develop objects that satisfy specific functional requirements, whereas DSR is the iterative process of generating new knowledge through design, analysis, reflection and abstraction, by creating novel artifacts, DSR not only solves immediate problems, but also actively shapes our understanding of the domain.

A fundamental principle of a DS approach is problem-driven design: researchers first analyze a problem context and then develop a corresponding solution. Inquiry, therefore, alternates between building and evaluating artifacts, such as algorithms, human–computer interfaces, methodologies, or programming languages, while tackling intellectual risk due to the degree of unknowns embedded in the proposed solution and the knowledge gaps that distinguish research from routine development.

This study differentiates two types of research problems: Practical challenges that require solutions in real-world settings and knowledge inquiries that aim to understand existing phenomena without necessarily implementing changes. The specific technical issue addressed in this research is: *How can we develop an AI-assisted algorithmic artifact that satisfies the requirements of efficacy and performance so that security researchers can use it in fuzz testing?*

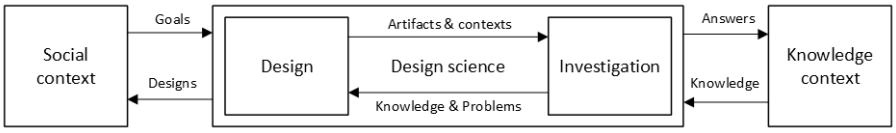


Figure 4: The design science framework (based on Wieringa, 2014)

The stakeholders (in this case, security researchers) define the problem context and contribute domain knowledge during artifact design. Figure 4 illustrates the comprehensive DS framework: the social context includes all stakeholders (e.g., blue and red teams), while the knowledge context includes current scientific and engineering theories, known system architectures, available tools, lessons from prior DS projects, and general reasoning patterns. DS projects leverage this collective expertise to create new artifacts or address knowledge-related questions, thus continuously expanding the scope of research both in practice and theory.

Johannesson and Perjons (2021) introduce a framework for DSR, detailing five key activities that range from problem investigation and requirement definition, through artifact design and development, and finally, to demonstration and evaluation, following a structured problem solving process, as shown in Figure 5. This process is inherently it-

erative, with insights from demonstration and evaluation continuously feeding back into earlier stages, such as problem explication, requirements definition, and artifact refinement, across the following key phases:

- **Explicate Problem** Analyze the practical problem and establish its significance on both local and global scales.
- **Define Requirements** Derive artifact requirements with respect to functionality, structure, and operating environment.
- **Design and Develop Artifact** Construct the artifact to address the problem and satisfy the requirements.
- **Demonstrate Artifact** Test the artifact in illustrative or real-world scenarios to verify feasibility.
- **Evaluate Artifact** Assess the effectiveness of the artifact in solving the initial problem.

The purpose of DSR, as outlined by Johannesson and Perjons (2021), is to create artifacts and to address questions about them and their environments. This type of research is flexible in methodology, allowing any research strategy or method to be employed based on the project's specific needs. Using a case study can provide a deeper understanding of a problem and its context, though it may limit the generalizability of the results.

Despite their widespread use, case studies in DSR lack a standardized methodological framework. Teegavarapu et al. (2008) examines the role of the case study method within DSR, providing a rationale for its suitability through a comparative analysis of other qualitative and quantitative research methodologies.

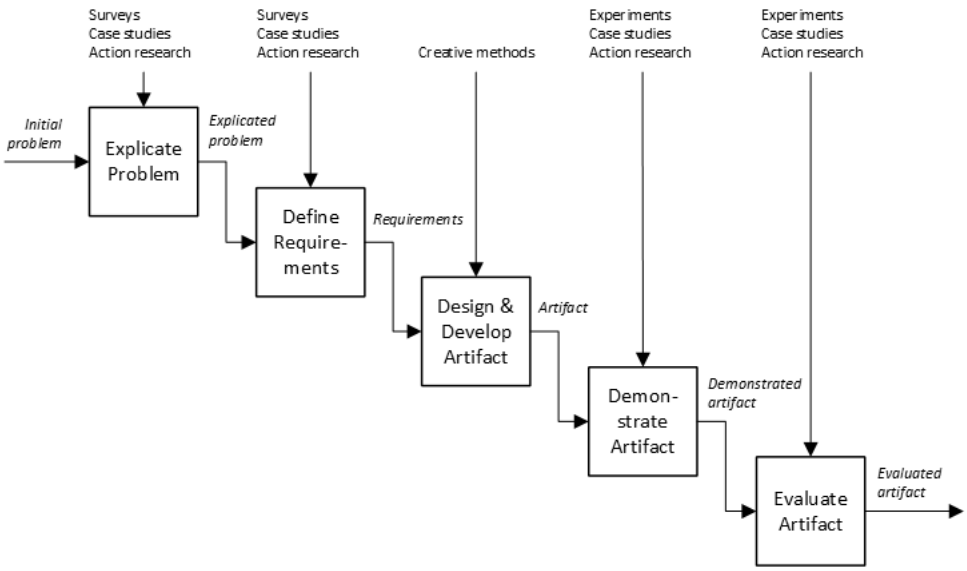


Figure 5: Examples of research strategies (Johannesson and Perjons, 2021)

Johannesson and Perjons (2021) also observe that many DSR projects prioritize only one or two of these research activities. A common pattern is to focus on development

and evaluation, starting with an existing requirements specification to prioritize designing and developing an artifact, then proceeding to demonstration and rigorous evaluation via experiments, case studies, or mixed methods. This dissertation adopts such a focus on development and evaluation: a NN model is designed and implemented, demonstrated in a case study on fuzzing the Windows graphics component with EMF metafiles, and then tested experimentally under controlled conditions.

The basis of this work is the broader design cycle of the engineering process, as illustrated by Figure 6, which involves the implementation and evaluation of a tested solution within its problem setting and iterating as needed. Although engineering and design cycles offer a structured framework for various tasks, they do not prescribe a specific sequence or dictate how the development process should unfold. Different execution methodologies, such as waterfall and agile approaches, provide distinct ways to structure engineering and design efforts.

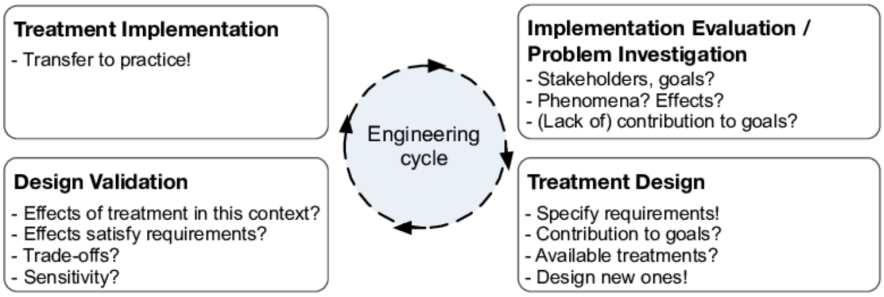


Figure 6: The engineering cycle as a problem-solving process (Wieringa, 2014)

In this research, the artifact is an AI-driven tool to improve the efficiency of fuzz testing by prioritizing the selection of seed inputs that are most likely to reveal new vulnerabilities. The complexity of EMF image file processing makes the Windows graphics component an ideal target for both case study demonstration and experimental evaluation.

This research demonstrates the feasibility of integrating ML into fuzzing, focusing on validating the concept of ML-assisted seed selection rather than comparing various ML algorithms exhaustively. Previous work in this area has typically evaluated ML-assisted tools against traditional fuzzing methods rather than multiple ML models in isolation. Given the ambitious scope of the research, this work compares a single, well-performing ML model with established traditional techniques, allowing for a focused validation of the overall concept without diverting attention to extensive model comparisons.

Finally, although technical action research (TAR) provides formal mechanisms to assess adoption and practical usability, its application is deferred in this study. The development of the artifact and the testing of its technical properties through commonly used fuzzing techniques under controlled conditions are key aspects of this study. Future research should explore broader impacts, such as user adoption and real-world deployment, utilizing TAR or similar frameworks.

1.4 Artifacts of the research

This section provides an overview of the key results and software artifacts developed during the research conducted for this doctoral dissertation. These contributions enhance various aspects of fuzz testing, including data processing and bug detection. The advances made in this research collectively contribute to the progress of vulnerability research.

A1: Improved binary template for metafiles The first artifact is an improved binary template for the EMF image file format, designed for use with both the widely recognized `010 Editor` and the innovative `FormatFuzzer` testcase generator. This template fills a gap in existing tools by covering approximately 80% of available metafile records. I validated its accuracy using the `010 Editor` as a parser and `FormatFuzzer` as a generator to ensure its reliability for future fuzz testing applications.

A2: Fuzzing harness for the Windows graphics component The second artifact is a fuzzing harness, which plays a crucial role in the fuzzing process. The harness is responsible for the initialization of the testing environment and the integration of the target software component. It encapsulates the target program and prepares it to process EMF files, allowing the fuzzer to interact with the Windows graphics component. Compared to other publicly available test programs designed for the same purpose, this harness offers enhanced functionality. It supports the direct processing of sample files or the use of shared memory as a means of data transfer, enables testing of a broader range of graphics operations, such as creating thumbnails and converting metafiles, and includes some debugging features.

A3: Additional feature for crash deduplication The third artifact is an additional feature implemented in the `WinAFL` fuzzer for deduplicating crash samples. During testing, the fuzzer may detect multiple distinct crashes that, in reality, originate from the same underlying flaw. This feature enables the fuzzer to log the memory address of each program crash and store previously identified crashes separately in a `bug bucket`. This method simplifies the process of triaging new findings by keeping redundant crash samples stored separately.

A4: Fuzz manager for remote monitoring The fourth artifact is a fuzz manager known as `WinAFL Pet`, which I developed to serve as a web user interface for the remote management of `WinAFL` fuzzer instances. This tool facilitates the easy monitoring of fuzzing campaigns running on remote machines, utilizing a lightweight agent that operates as a system service. `WinAFL Pet` significantly enhances the usability of the `WinAFL` fuzzer by providing a user-friendly interface to manage and monitor fuzzing campaigns.

A5: ML-based prototype for seed file selection The fifth and main artifact, a working prototype of the `SeedClass` tool, enhances the seed selection process in fuzz testing. Coverage-guided mutational fuzzers produce thousands of mutated testcases that may uncover new code paths that lead to program crashes. However, fuzzing is a time- and resource-intensive testing method, making the identification of valuable seed inputs worthwhile to improve efficiency. `SeedClass` addresses this challenge by using a NN model to predict high-value seed files from a candidate dataset. This approach enables more informed decisions on which inputs are best suited for fuzzing. By accurately labeling the input, the tool can seamlessly adapt and integrate with various existing fuzzers.

2. Background

As the scope of the research narrows, I progressively delve deeper into the subject, moving from the general use of AI in cybersecurity to its specific use cases in fuzz testing. This chapter outlines that progression, followed by a presentation of the traditional fuzzing process, which establishes the theoretical foundation of the research. I also review relevant ML concepts, which are touched upon to varying degrees throughout the individual sections.

2.1 AI in cybersecurity

The increasing use of AI in cyberattacks has become a growing concern around the world, as AI-driven attack techniques can be highly destructive. While skilled professionals remain scarce and costly to recruit, the complexity and scale of these emerging threats are overwhelming (Nurse et al., 2021). Cyber threat actors leveraging AI-driven offensive techniques (Girhepuje et al., 2024) are capable of conducting targeted attacks while effectively evading traditional rule-based detection systems (Guembe et al., 2022).

As AI continues to learn and adapt, future cyberthreats may become even more personalized, scalable, and indistinguishable from human-generated attacks, rendering traditional security measures increasingly ineffective and necessitating the adoption of AI-driven approaches (Malatji & Tolah, 2025). Automating threat detection and response using AI and ML can significantly enhance cybersecurity by improving accuracy and reducing response times. One of the key advantages of AI and ML in cybersecurity is their ability to identify and mitigate threats in real time. AI can analyze large datasets to detect subtle patterns that indicate possible attacks. These insights enable security teams to respond quickly and minimize any damage.

AI has the potential to revolutionize cybersecurity by allowing organizations to anticipate, detect, and counteract threats more effectively. As cyberattacks become increasingly sophisticated, AI-driven security measures will be integral in protecting organizations from evolving threats. ML techniques can also improve the accuracy of threat detection by learning from previous incidents to identify emerging attack patterns. A proactive approach enables organizations to stay ahead of adversaries by preventing new threats and strengthening long-term cybersecurity resilience. Furthermore, AI-driven automation improves operational efficiency by streamlining threat response and incident management, allowing security teams to focus on more complex tasks (Group, 2023).

Babuta et al. (2020) highlight that AI-driven attacks frequently use sophisticated obfuscation methods to alter identifiable characteristics, making them difficult to detect by

traditional behavioral or signature-based protection tools. Consequently, existing security tools need to be improved with advanced security mechanisms to counter these evolving threats. Hamadah and Aqel (2020) reviewed different AI algorithms and ML models applied for cybersecurity, especially intrusion detection (Meshram, 2023), and concluded that while these technologies have shown that they can support security activities, the most effective methods are not yet available, and future work should focus on improving the performance of these models with further experimentation on larger or more diverse datasets.

2.1.1 AI-enabled cyberattacks

Chilton (2023) critically examined the cybersecurity implications of ChatGPT[8] and raised concerns that it creates significant opportunities for malicious actors to generate sophisticated phishing scams and potentially malicious code. Historically, many phishing attacks were identifiable due to poor grammar and spelling. However, the advanced capabilities of large language models (LLMs) make it much more difficult to distinguish legitimate messages from fraudulent ones. Although publicly available generative AI services are designed to prevent the creation of malicious code, they can still be manipulated into doing harmful actions[9].

In this context, *red teams* serve as a practical illustration of how ongoing professional training can strengthen cybersecurity measures. These teams consist of security specialists who adopt an adversarial perspective to evaluate defenses (Yamin et al., 2020). As Tubberville and Vest (2020) observes, red teams are employed in various domains to assess the effectiveness of personnel, processes, and technologies by simulating adversary tactics using tools and techniques similar to those applied in cyberattacks, often following a structured framework known as the Cyber Kill Chain, as CrowdStrike (2019) explains.

Figure 7: Phases of the cyber kill chain model (based on Hutchins et al., 2011)

Traditional network defenses focus on vulnerabilities and assume breaches have happened, but they fall short against APTs. Hutchins et al. (2011) describes an AI-driven approach to study intrusions from the perspective of adversaries and presents a new kill chain model specifically to describe the structure of an intrusion. The model outlines the phases of an attack as shown in Figure 7, starting with reconnaissance and usually con-

[8]ChatGPT is a conversational AI model developed by OpenAI that uses DL methods, refined through both supervised learning and human feedback, to produce coherent and contextually relevant text responses that closely mimic human conversation (Gregersen, 2025).

[9]In early 2023, multiple threat actors initiated discussions in underground forums about using ChatGPT for malicious activities (Kaspersky, 2024).

cluding with data exfiltration. It is important to note that the phrase *chain* implies that an adversary must successfully progress through each stage before achieving its desired objective, which means that defenders can gain a strategic advantage, as a single mitigation could disrupt the chain and block the attack.

Kaloudi and Li (2020) analyzed AI-driven cyberthreats and developed a structured framework that categorizes various malicious AI applications throughout the lifecycle of cyberattacks. This framework serves as a reference point for threat detection and future risk prediction. As depicted in Figure 8, AI enhances adversarial capabilities in all seven attack phases, from target identification to execution and maintaining the command and control (C2) infrastructure. In addition, AI facilitates the weaponization of software flaws, including the discovery of zero-day[10] vulnerabilities, which lack publicly available patches or workarounds and therefore pose a significant security risk.

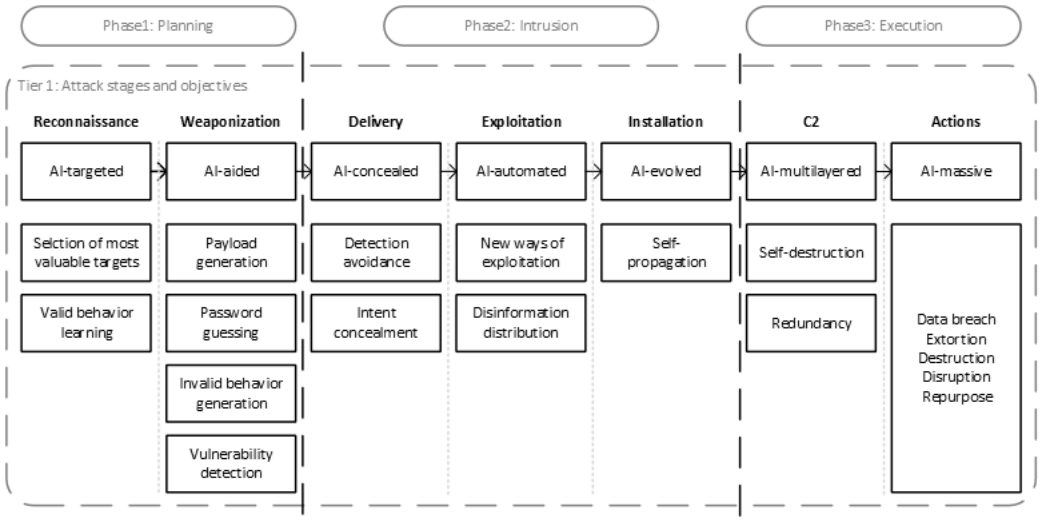


Figure 8: The AI-based cyberthreat framework (based on Kaloudi and Li, 2020)

Pratt (2023) highlights the increasing cybersecurity risks associated with the widespread adoption of AI. Cybercriminals are integrating AI into their operations at a faster rate than enterprise security teams, raising concerns about the increasing scale and velocity of attacks. AI enables cybercriminals to devise strategies that exceed human capabilities, enhance operational efficiency, and expand attack vectors. Threat actors leverage AI to optimize attack methodologies, generate sophisticated malware, orchestrate phishing and smishing[11] campaigns, and produce deepfakes that convincingly impersonate high-profile individuals.

As noted previously, the adoption of generative AI has significantly increased the sophistication of phishing attacks. Malicious actors can now accurately mimic the tone,

[10] The term *zero-day* refers to the period between the discovery of a vulnerability and its public disclosure.

[11] Smishing is a form of cyberattack that combines ”SMS” and ”phishing”, utilizing fraudulent text messages to deceive individuals into disclosing sensitive data, clicking harmful links, or downloading malicious software (Shoaib & Umar, 2023).

style, and format of genuine messages, making it more difficult to distinguish between malicious and valid content. AI-driven phishing campaigns can produce highly convincing messages in multiple languages, further complicating detection and expanding their reach.

Pratt (2023) concludes that AI-enabled cyberattacks pose a growing threat that organizations must address proactively. As AI technologies continue to evolve, cybercriminals are likely to refine their techniques, making threat detection and mitigation even more challenging. Organizations must continuously adapt cybersecurity strategies to stay ahead of evolving threats and protect sensitive information.

2.1.2 AI-concealed malware

Unlike software bugs [12] which cause unintended harm due to programming flaws, malware [13] actively seeks to compromise security by disrupting or damaging computer systems (Laka, 2022; Moir, 2009). Attackers often distribute malware through spear-phishing (SP)[14] emails or embed them into widely known, seemingly legitimate and harmless programs that can be freely downloaded from the Internet. To accomplish its objective, malware must evade detection by widely used security mechanisms such as sandboxes[15] and antivirus software. Through the implementation of advanced evasion techniques, malware can conceal its activities, impede forensic analysis, and thwart efforts to attribute its origin.

As described by Mugeshwaran (2023), sandbox environments function as isolated and controlled systems that allow the analysis of potentially harmful software and serve as a first line of defense. However, malicious software authors continually refine their tactics to prevent malware from executing within a sandbox and running only in genuine environments, thus avoiding detection and analysis (Roccia, 2019). AI and ML are progressively integrated into detection systems to continuously evaluate extensive datasets and identify emerging threat patterns. Despite their potential, these methods require extensive training data and themselves may be susceptible to adversarial attacks (Barriga & Yoo, 2017).

Research specifically addressing the use of AI by malware itself is scarce. Still, a survey paper presented by Pan and Fung (2008) reviewed the ongoing AI-driven malware arms race, highlighting how both malware and anti-malware systems are progressively adopting more sophisticated techniques. Among early examples, the `Zellome` virus reportedly utilized genetic algorithms to achieve its polymorphic behavior. More recent

[12] A software bug is an unintended flaw in the design or code of a computer program that causes the software to produce incorrect results, behave in unexpected ways, or even crash (Tan et al., 2014; WelTec, n.d.).

[13] Malware is software designed to purposely damage other software that is normally safe (Kramer & Bradfield, 2010).

[14] Spear-phishing is a highly targeted form of phishing attack where cybercriminals tailor their deceptive messages to a specific individual or organization (Wassermann et al., 2023).

[15] A sandbox is a secure, isolated environment used for safely running and analyzing potentially harmful software, preventing any damage to the real system (Alrawi et al., 2024).

academic discourse recognizes the potential for future AI-powered malware (Cohen et al., 2025), but empirical evidence and detailed analysis remain limited. As a recent example, Padovan (2024) explored the use of reinforcement learning (RL) for evasion, introducing a novel approach that uses system calls for device behavioral fingerprinting and anomaly detection.

Recognizing the need for automated malware analysis due to the increasing volume of malware samples, Yokoyama et al. (2016) highlighted the potential risks where attackers might exploit these systems to learn the characteristics of the sandbox. To address this concern, they introduced `SandPrint`, a tool designed to measure and extract features specific to Windows-targeted sandboxes. By analyzing data originating from multiple malware services, they assessed whether adversaries could discern characteristics common to all sandboxes. Using supervised learning techniques, they demonstrated that attackers could develop a classifier capable of distinguishing sandboxes from real target systems. Furthermore, their methods allowed them to identify commercial malware security appliances from three major vendors, which underscores the challenges that sandbox operators face in concealing key features of their system.

As ML methods have advanced, attackers have developed adversarial techniques to bypass ML-based malware detectors. Adversarial attacks have become so effective that major corporations have hosted public challenges aimed at encouraging researchers to circumvent their security systems, thereby refining their detection mechanisms. Ceschin et al. (2020) participated in this challenge and successfully bypassed all tests using an automatic exploitation method that not only bypassed the ML detectors, but also had a lower detection rate from real antivirus engines. Another similar competition revealed biases in the ML models and raised concerns about their practical deployment, as some system files were misclassified as malware (Jarek, 2020; Rehberger, 2020). As adversarial techniques evolve, researchers and security professionals must continuously refine ML-based malware detectors to mitigate evasion risks.

Given that adversarial malware can bypass even advanced detection systems, how does this challenge reshape our understanding of what it means to identify and respond to unauthorized activity? Intrusion detection includes a set of methodologies and practices designed to identify activities that compromise the confidentiality, integrity, and availability of computer and network components. Any unauthorized attempt to gain access, manipulate, alter, or destroy data is classified as an intrusion (Bruneau, 2001). Malware attacks that employ analogous tactics fall similarly under this definition.

An intrusion detection system (IDS)[16] is employed to continuously monitor information systems through analysis of network traffic and device usage, thus facilitating the interpretation of various system events and subsequent alerting of security analysts. De-

[16]It is important to emphasize that the primary role of an IDS is to detect and report potential security issues rather than to prevent them automatically. For immediate and proactive responses, organizations typically deploy an intrusion prevention system (IPS).

spite their overall utility, signature-based IDS solutions exhibit significant limitations, particularly in detecting zero-day attacks that exploit previously unknown vulnerabilities. Although some incidents may be indirectly identified through analogous patterns, detecting such attacks remains challenging because standard software security updates cannot preemptively address unknown exploits.

However, heuristic-based and anomaly-based detection strategies can improve identification rates by recognizing suspicious deviations from expected behavior. Holm (2014) evaluated the performance of Snort (Roesch et al., 1999) using the Metasploit Framework (Moore, 2004) and found that it detected only 17% of zero-day attacks. This outcome emphasizes the need to develop more advanced detection methods. IDS methodologies have evolved from manual supervision by experts and rule-based detections to approaches that leverage advances in ML, with modern solutions increasingly relying on data science to efficiently process large-scale event logs and network traffic.

According to Grakovski et al. (2022), intrusion detection solutions can generally be classified into two categories: anomaly-based and signature-based approaches. The anomaly-based approach establishes a model of expected behavior and subsequently flags deviations that exceed predefined thresholds, while the signature-based approach focuses on analyzing data for patterns corresponding to known attacks, including malware signatures. Brunner (2020) further investigated solutions that employ various ML models capable of signature-based and hybrid detection. Although ML methods have demonstrated considerable efficacy in detecting a wide range of intrusions, sophisticated attacks that exploit zero-day vulnerabilities may continue to evade detection. Furthermore, the potential misuse of AI in the context of intrusion detection raises further concerns, thus underscoring the need for responsible AI development.

2.1.3 AI-assisted fuzz testing

In cybersecurity, a vulnerability refers to a flaw in the design, implementation, or configuration of a software system that can be exploited by attackers to compromise security, allowing unauthorized actions such as data theft or system manipulation. Given that corporations often operate under strict resource constraints, effective vulnerability management[17] is crucial (Williams & Nicolett, 2005). Foreman (2019) argues that allocating resources to security measures such as intrusion detection or security event management before implementing vulnerability management is unjustifiable, as addressing critical vulnerabilities significantly improves an organization's overall risk profile. I fully support this rationale, as proactively identifying and addressing both known and unknown vulnerabilities, before they can be exploited, offers the most immediate and strategic impact on organizational security.

[17] Vulnerability management is the iterative process of identifying, classifying, remediating, and mitigating vulnerabilities (Foreman, 2019).

Aligned with this perspective, vulnerability research, which is the main focus of this dissertation, aims to identify previously undetected security flaws (Iannone et al., 2023). A fundamental concept in this field is that the cost of remediating a vulnerability rises the later it is found, making early detection far more cost-effective. Ideally, vulnerabilities should be detected as early as possible in the development cycle. However, the sheer scale of code bases and the limited expertise among developers present considerable challenges (Austin & Williams, 2011). To address these difficulties, various techniques have been introduced to assist developers in detecting vulnerabilities, although no single approach is capable of identifying all types of vulnerabilities. Over time, vulnerability research has evolved from manual methods to increasingly sophisticated and automated approaches, reflecting ongoing advances in the field (B. Liu et al., 2012).

The integration of ML in vulnerability research is a particularly promising field. In their overview of common vulnerability research methods that focus on key technologies that enable intelligent vulnerability discovery, Zhu and Liang (2019) highlights the combination of ML with program analysis techniques to improve vulnerability detection. While static and dynamic analysis remain the central methods, both have inherent limitations. For example, integrating ML methods into a dynamic analysis method, such as fuzzing, can improve the quality of test samples, thereby increasing their effectiveness in triggering unknown vulnerabilities. Fuzzing involves the iterative execution of a program using altered inputs that may contain syntactic or semantic errors. While adversaries often employ fuzzing not only to identify vulnerabilities, but also to facilitate the development of reliable exploits, defenders increasingly use it to proactively identify security weaknesses before attackers do.

First introduced by B. Miller (1988), fuzz testing is a widely used automated software testing method that has proven to be highly effective in detecting security flaws in complex programs. It systematically applies invalid or unexpected input data to reveal potential vulnerabilities. Recognizing its significance, Manès et al. (2021) and his team proposed a comprehensive fuzzing model, classifying existing research, and providing a structured perspective on fuzz testing methodologies. They define fuzzing as the process of executing the program under test (PUT) using inputs that exceed its typical or expected operational parameters.

The core principle of fuzzing is to continuously apply malformed inputs to a target program to expose unexpected behaviors, such as crashes and exceptions. Due to its simplicity and low performance overhead, fuzzing remains an effective method to identify security flaws in various real-world systems. As a key component of vulnerability research, fuzzing contributes to the discovery of previously unknown security flaws, complementing traditional vulnerability discovery techniques and vulnerability management strategies that aim to address known weaknesses.

Figure 9 outlines the fuzz testing procedure as described by Y. Wang et al. (2020). The

process consists of four primary stages: testcase generation, program execution, runtime status monitoring, and crash investigation. The generation of testcases involves selecting a seed file, applying the mutation, creating test samples, and filtering.

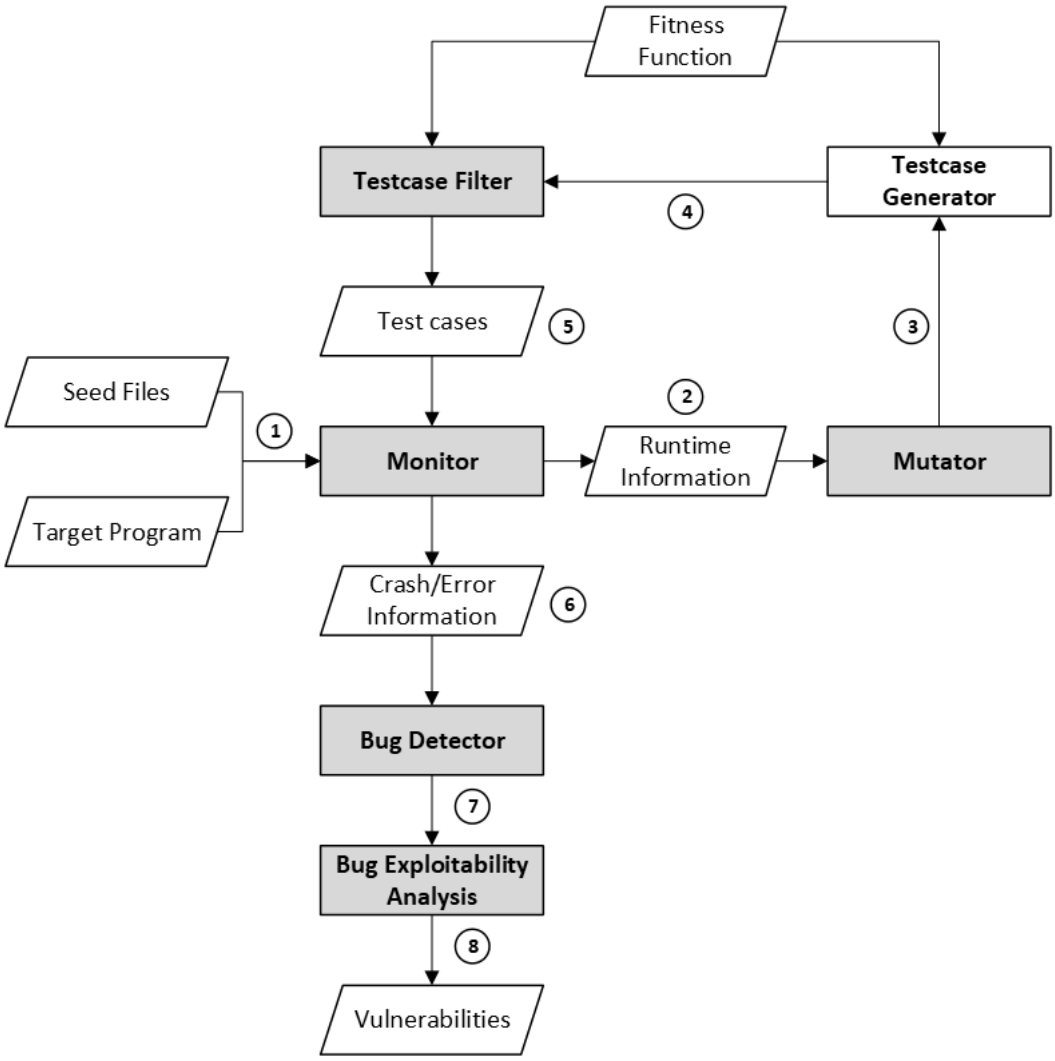


Figure 9: General fuzzing process (Y. Wang et al., 2020)

During execution, the testcases are loaded into the PUT, which is then run under controlled conditions. The monitoring stage tracks the program's behavior at runtime, directing test sample generation by feeding information back to the testcase generator. When crashes occur, they are analyzed to determine whether they are the result of software bugs and whether they constitute security vulnerabilities.

Despite its effectiveness, traditional fuzzing techniques face limitations, particularly in handling large programs where fuzzers frequently generate redundant inputs without uncovering deeply hidden vulnerabilities (She et al., 2019). Researchers have explored enhancements, including static and dynamic analysis, to improve the efficiency of fuzzing. However, challenges remain, including optimizing input mutation, increasing code coverage, and bypassing verification routines (Binosi et al., 2023; Y. Li et al., 2019).

To address these issues, fuzz testing has increasingly integrated ML. Y. Wang et al. (2020) examined recent advances in ML-enhanced fuzzing, categorizing fuzzing steps by the problems that ML can solve: seed file generation, testcase generation, filtering, mutation operator selection, fitness function optimization, and exploitability analysis. Y. Wang et al. (2019) note that the generation of testcases is the most commonly studied ML-driven fuzzing step, probably due to its role in collecting large volumes of labeled data for model training. The subsequent sections will explore key stages in the fuzz testing process that incorporate AI technologies.

Seed input generation

Coverage-guided fuzzing is widely recognized as one of the most effective fuzzing methods for identifying vulnerabilities due to its speed and scalability (Y. Wang et al., 2019). However, traditional approaches prioritize code coverage over susceptibility, focusing on as many execution paths as possible rather than on the most vulnerable ones. Many fuzzers treat all seed inputs equally, disregarding the fact that different inputs traverse paths with varying degrees of vulnerability. This inefficiency may result in wasted computational resources, as time is spent testing less relevant paths instead of those more likely to expose security flaws.

To address this limitation, Lyu et al. (2018) introduced `SmartSeed`, a ML-driven approach to seed generation. Traditional methods struggle to accommodate increasingly complex software systems, which makes targeted seed selection essential for effective fuzzing. `SmartSeed` analyzes the input format of the target program and leverages learned data to produce candidate seeds with a higher probability of triggering program crashes. These candidates are then evaluated through a coverage-guided fuzzer to assess their effectiveness based on code coverage and bug-finding capability.

The empirical evaluation of `SmartSeed` on real-world software applications, including image processing libraries and media players, demonstrated its performance compared to conventional seed generation techniques. The results indicated that `SmartSeed` achieved significantly higher code coverage and identified more bugs with fewer fuzzing iterations.

Similarly, Y. Wang et al. (2019) introduced `NeuFuzz`, an intelligent seed selection approach that uses a deep neural network (DNN). `NeuFuzz` trains a predictive model to assess whether an execution path is vulnerable by learning from a large dataset of *clean* and *vulnerable* code paths. The fuzzer then prioritizes seed inputs that are likely to expose security weaknesses. Experimental results suggest that `NeuFuzz` identifies vulnerabilities more efficiently than traditional fuzzers, reducing the time required for bug detection. These advancements highlight the versatility of ML-driven seed selection in fuzzing, enhancing both efficiency and effectiveness in vulnerability detection.

Testcase generation

The core function of a fuzzer, as described in Rawat et al. (2017), is its ability to generate input that triggers software bugs. Most state-of-the-art fuzzers prioritize code coverage to uncover vulnerabilities; however, not all code that is covered is necessarily prone to security flaws. Although greater coverage increases the probability of finding bugs (Yi et al., 2021), vulnerable code typically represents only a small fraction of a program. Achieving maximum code coverage for real-world applications is computationally demanding and requires significant resources.

To improve efficiency, Y. Li et al. (2019) proposed `V-Fuzz`, a vulnerability-oriented fuzzing framework designed to rapidly identify vulnerabilities in binary programs. `V-Fuzz` integrates a NN-based vulnerability prediction model with a novel fuzzer that focuses on high-risk program components. The prediction model assesses which sections of a binary are likely to contain security flaws, while the fuzzer generates targeted inputs to reach these components. Experimental results demonstrated that `V-Fuzz` detected more vulnerabilities in less time than conventional fuzzers, including ten known security flaws, three of which were zero-days.

To represent binary functions, Y. Li et al. (2019) introduced attributed control flow graph (ACFG), an extension of traditional control flow graph (CFG). While CFG is commonly used for bug detection, it is not inherently compatible with DL models, as it does not exist in numerical vector form. ACFG addresses this limitation by encoding basic-block level attributes as numerical vectors, allowing binary functions to be represented as structured data suitable for ML-based analysis.

Figure 10 illustrates the data preprocessing workflow, while Figure 11 describes the vulnerability-oriented fuzzing workflow, where `V-Fuzz` utilizes their proposed algorithm to generate inputs that target potentially weak components of the program.

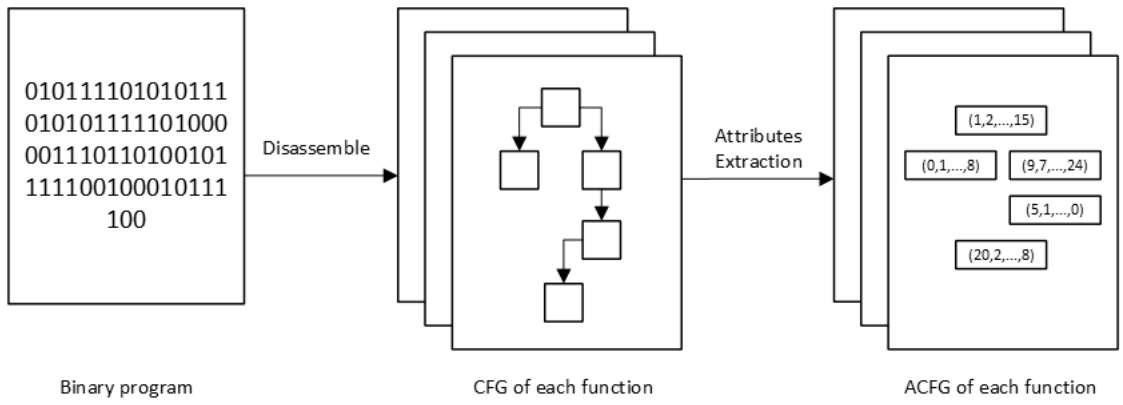


Figure 10: Data preprocessing workflow (Y. Li et al., 2019)

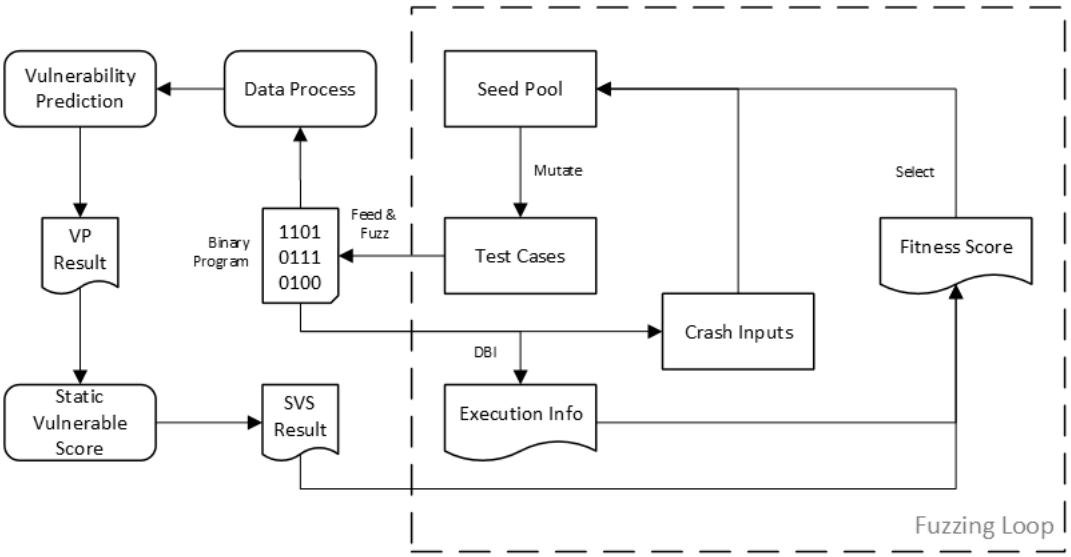


Figure 11: Vulnerability-oriented fuzzing workflow (Y. Li et al., 2019)

Testcase filtering

Recent advances in ML have led to numerous cybersecurity studies exploring its applications in vulnerability discovery and fuzz testing. While standard fuzzing approaches rely on uniform random mutations, Rajpal et al. (2017) introduces an ML-driven technique that utilizes NNs to analyze patterns in previously executed input files. This approach enhances the efficiency of gray-box fuzzers by prioritizing mutation locations likely to guide the fuzzer toward unexplored code paths.

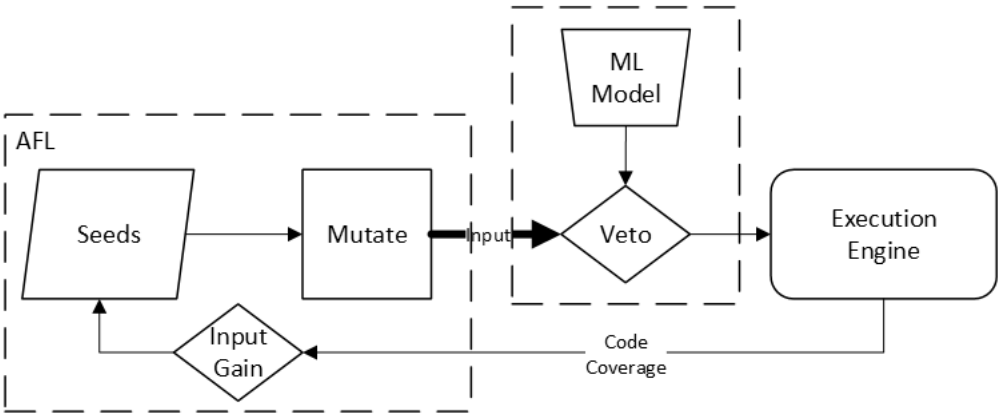


Figure 12: The veto model used by Augmented-AFL (Rajpal et al., 2017)

The proposed method augments the `AFL` fuzzer by integrating NNs trained on past mutation data and the corresponding code coverage metrics, enabling the model to predict optimal mutation locations within input files. This enhancement aims to reduce computational overhead by minimizing the time spent generating ineffective inputs while improving overall code coverage and crash discovery rates.

Before fuzzing begins, `Augmented-AFL` queries the model with each seed file, classifying input regions as useful or ineffective. Mutations targeting non-useful regions are vetoed, streamlining execution efficiency. Figure 12 illustrates this approach, demonstrating how ML-driven filtering optimizes mutation targeting. Evaluations on file formats such as `ELF`, `PNG`, `PDF`, and `XML` indicate substantial improvements in code coverage and unique execution paths for `ELF` and `PNG`. However, the approach exhibits performance limitations when applied to `PDF` files due to the computational overhead of handling large inputs.

Rajpal et al. (2017) highlights the potential of ML to refine fuzz testing and suggests several promising directions for further exploration, including online adaptation based on RL and generative models for input mutation. The findings suggest that integrating NNs into fuzzing workflows can advance vulnerability discovery, although optimization challenges persist. Rajpal et al. (2017) envisions future fuzzers that leverage ML within an iterative feedback loop, continuously refining mutation strategies based on prior execution data.

2.2 Traditional fuzz testing

Fuzzing is an automated technique to discover software vulnerabilities by providing manipulated, random, or anomalous inputs (M. Sutton et al., 2007). The word *fuzz* is the result of a class project led by B. Miller (1988) at the University of Wisconsin (Fell, 2017). In that project, the students performed *fuzz testing*, a method to evaluate the reliability of command-line programs (B. P. Miller et al., 1990). The approach involved automatically generating random input and command-line parameters for the utilities tested. Teams repeatedly ran test programs using various random input values until the programs crashed.

Students successfully triggered crashes in approximately one-third of the utilities tested. Following each crash, they debugged the code to determine its root cause. The class made the source code of their tools and raw result data publicly available, enabling other researchers to conduct similar experiments on different software. This early fuzzing approach is now referred to as black-box fuzzing, also known as ”dumb” fuzzing.

Fuzzing is a computationally intensive and time-consuming software testing methodology that supplies random or semi-random input to a system to uncover unknown defects. From a security perspective, fuzzing seeks to identify vulnerabilities that threat actors could exploit to compromise the system, steal sensitive data, or trigger a denial-of-service condition. This technique is widely applied to testing network protocols, web applications, file formats, and operating systems, since it generally does not require access to the internal structure of the system.

Most randomly generated inputs are malformed or invalid. Consequently, the processing program often rejects them early, only accepting inputs that meet specific syntactic and structural criteria. Any input that does not meet these requirements is typically flagged as

invalid and discarded. However, certain techniques enable the generation of syntactically valid inputs that have a higher likelihood of being accepted by the processing program (Zeller et al., 2024).

There are several approaches to fuzzing (Pramanik & Tayade, 2017), including mutation-based, generation-based, and hybrid (C. Chen et al., 2018). *Mutation-based fuzzing* introduces small modifications to existing inputs (such as flipping bits, altering values, or deleting data) to maintain validity while inducing new program behavior. *Generation-based fuzzing* relies on basic knowledge of the data, creating new inputs based on a predefined grammar or template. C. Miller and Peterson (2007) demonstrated that generation-based file format fuzzing outperforms mutation-based fuzzing due to the limited diversity of sample files available for testing. *Hybrid fuzzing* integrates mutation- and generation-based techniques to improve coverage and efficiency.

Fuzzing can be performed manually or automatically. *Manual fuzzing* requires the manual creation of testcases and their individual execution, a process that is slow, tedious, and prone to human error. In contrast, *automatic fuzzing* leverages simple tools or complex frameworks to systematically generate and execute testcases. This automated approach is significantly faster and more efficient.

Fuzzing can be also classified as deterministic or non-deterministic. *Deterministic fuzzing* systematically tests inputs by applying a predefined set of mutations to the system sequentially. This method is effective in detecting simple flaws, but may overlook more complex vulnerabilities that require a broader range of inputs. In contrast, *non-deterministic fuzzing* dynamically produces input based on the observed behavior of the system, incorporating random techniques to further mutate the data. Although this approach is more effective in uncovering complex issues, it typically requires more computational resources and also more time.

Fuzzers have inherent limitations as they can only identify bugs triggered by the executed testcases, potentially overlooking vulnerabilities that require specific conditions or inputs. In addition, they may produce false results, which can lead to wasted time and effort. Therefore, the recommended approach is to integrate complementary testing techniques to achieve a more comprehensive evaluation of software security. Fuzzers can be classified into multiple, potentially overlapping categories, depending on the perspective or criteria used. The following list provides a concise summary of various fuzzing techniques, with subsequent sections offering a more detailed examination of each type (Fell, 2017).

- *Knowledge of the input data*:
  - **Dumb** fuzzers generate random input without understanding the behavior of the target program.
  - **Smart** fuzzers apply data analysis techniques to generate input that is more likely to trigger bugs.

- *Knowledge of the target program*:
  - **Black-box** fuzzers operate without knowledge of the program under test and generate random input.
  - **Gray-box** fuzzers possess partial knowledge of the application and generate input that can reach specific execution paths.
  - **White-box** fuzzers have significant knowledge about the program and generate input that can examine all possible code paths.
- *Program exploration strategy*:
  - **Coverage-guided** fuzzers use feedback from the application under test to refine input generation for maximizing code coverage.
  - **Directed** fuzzers generate input that targets specific areas of the program to systematically test critical components.
- *Input data creation methods*:
  - **Mutation-based** fuzzers start with existing input and modify it to create new testcases.
  - **Generation-based** fuzzers create new input using predefined rules or models.

2.2.1 Knowledge of the input data

Dumb fuzzing

Dumb fuzzing is a straightforward fuzz testing technique that generates random input data for the PUT to observe its response. Although easy to implement, it has both advantages and limitations.

One of its benefits is the ability to quickly identify input validation errors, such as buffer and integer overflows, that traditional testing methods might overlook. Moreover, dumb fuzzing is easy to configure, supports automation, and enables continuous testing throughout the development process.

However, because it relies on random inputs, dumb fuzzing may fail to uncover more complex vulnerabilities requiring specific conditions or execution paths. It can also generate invalid or non-sensical inputs that do not reflect real-world usage.

Although dumb fuzzing is a useful testing tool, it should not be used exclusively. Combining it with techniques such as smart fuzzing (C. Miller, 2007) or manual testing increases coverage and improves the accuracy of vulnerability detection.

Smart fuzzing

Smart fuzzers use advanced algorithms to optimize the fuzzing process, thereby improving efficiency and effectiveness in bug detection. They analyze the input data to identify patterns and structures that help generate more impactful testcases. Furthermore, some

smart fuzzers utilize ML to adapt strategies based on previous tests, making them more effective than traditional fuzzers that rely solely on random data generation.

One example is `AFL`, which utilizes a genetic algorithm to produce testcases with a higher probability of triggering bugs. Another example is `Honggfuzz`, which employs a feedback-driven approach to prioritize testcases based on their impact on the program.

Smart fuzzers offer several advantages. They improve bug detection efficiency, saving time and computational resources. They also excel at uncovering complex vulnerabilities that may be difficult to identify using other testing methods. Furthermore, they can be applied in various software domains, including web applications, mobile apps, and Internet of Things (IoT) devices.

However, smart fuzzers can be computationally demanding and may overlook certain vulnerabilities. Some may also generate false positives, requiring additional time and effort to validate the findings. Despite these challenges, their ability to refine and enhance the fuzzing process makes them a valuable tool in security testing.

2.2.2 Knowledge of the target program

Fuzzers are typically classified into three categories, white-box, gray-box, and black-box, based on their reliance on knowledge of the program's internal structure and the depth of analysis they perform. Each approach differs in its ability to explore execution paths, detect vulnerabilities, and optimize code coverage. Table 2 provides an overview of commonly used fuzzers within these categories, further distinguishing them based on their input generation strategy: generation-based or mutation-based.

Fuzzers	White-box	Gray-box	Black-box

Generation-based	`SPIKE`
	`Sulley`
	`Peach`		

Mutation-based	`Miller`	`AFL`
		`Driller`
		`Vuzzer`
		`TaintScope`
		`Mayhem`
			`SAGE`
			`LibFuzzer`

Table 2: Common white-box, gray-box and black-box fuzzers (J. Li et al., 2018)

Black-box fuzzing

Conducting black-box fuzzing is like searching for a needle in a haystack, as it is entirely based on random input data without knowing the internal structure of the target program or the source code (Takanen et al., 2008). Instead, testers act as external observers who interact solely with the input and output interfaces (Woo et al., 2013) and submit various inputs to analyze the responses of the PUT, making this approach particularly suitable for closed-source, proprietary software where the source code remains inaccessible.

Widely employed for testing file formats, Application Programming Interface (API) interactions, and network protocols (Fan & Chang, 2018; Gascon et al., 2015), black-box fuzzing reveals unexpected behaviors in input handling without requiring source code access. Its broad applicability allows use across diverse software, including closed-source applications. However, its effectiveness is inherently limited, as the detection of complex or subtle vulnerabilities remains challenging compared to more informed fuzzing techniques.

White-box fuzzing

White-box fuzzing leverages the internal structure of the target program by analyzing its source code or intermediate representation (IR)[18] to systematically explore execution paths (Ganesh et al., 2009; Godefroid et al., 2012). Unlike black-box fuzzing, which mutates inputs without insight into the underlying code, white-box fuzzing employs techniques such as symbolic execution and constraint solving to generate testcases that uncover deeper execution paths (Xie et al., 2020).

This approach begins by executing the program with an initial input while collecting encountered branch constraints. These constraints are systematically resolved to produce new inputs that steer execution toward unexplored paths, thereby enhancing code coverage (Sparks et al., 2007). By utilizing structural insights, white-box fuzzers efficiently reach complex program states that random mutation-based fuzzing may not be able to access (Godefroid, Levin, & Molnar, 2008).

White-box fuzzing is particularly effective for open-source software, where direct access to the source code enables the generation of higher-quality testcases. In closed-source environments, it remains viable through reverse engineering techniques or symbolic execution on assembly-level representations, although with significantly greater difficulty. Advanced tools, such as `Angr`, help adapt white-box fuzzing methods for binary analysis (Stephens et al., 2016).

Despite its advantages, such as its ability to uncover intricate security vulnerabilities and logic errors, white-box fuzzing faces notable challenges. Computational overhead is a significant concern, as constraint solving and path exploration scale exponentially in large codebases. Furthermore, successful implementation requires expertise in program analysis and is often constrained by the availability of source code.

Gray-box fuzzing

Gray-box fuzzing is a hybrid approach that integrates elements of black-box and white-box fuzzing. It involves testing a target program with partial knowledge of its internal

[18] An intermediate representation is a machine- and language-agnostic abstraction that converts low-level code, such as binary or assembly, into a standardized high-level format that retains the original structure and behavior (Koschke et al., 1998).

structure, allowing the tester to access certain portions of the source code while remaining unaware of the system as a whole (Nascimento, 2024). This method employs input mutations and lightweight coverage feedback to systematically explore the input space within a constrained timeframe (Vikram et al., 2023). Given the complexity of the process, fuzzers utilize instrumentation feedback to identify promising inputs[19] and generate additional testcases.

This method is particularly useful for testing partially open-source or closed-source software. Limited insight into the program's internal structure enables testers to design more targeted testcases, although full code modification remains impossible. The advantage of gray-box fuzzing is its ability to balance the strengths of black-box and white-box approaches (Liang et al., 2021). It can effectively uncover complex bugs while remaining applicable to closed-source applications. However, its effectiveness may not match that of white-box fuzzing in detecting certain vulnerabilities that require a deeper level of code analysis.

2.2.3 Exploration strategy

Directed fuzzing

Directed fuzzing builds on traditional fuzz testing by taking a more targeted approach. Tools such as *AFLGo* (Böhme et al., 2017), *Hawkeye* (H. Chen et al., 2018), and *SemFuzz* (You et al., 2017) generate testcases designed to explore specific areas of the target program. These testcases are developed using a combination of automated and manual techniques to evaluate the program's behavior under various conditions. By focusing on specific program areas, it can expose flaws that remain undetected during standard execution when using conventional testing methods.

However, creating effective testcases requires a careful selection of input data and target areas of the program. Additionally, directed fuzzing can only identify vulnerabilities within reach of the predefined testcases. If an issue falls outside the tested scenarios, it may go undetected. For a more comprehensive approach, directed fuzzing should be complemented with other testing methods to ensure robust and thorough security analysis.

Coverage-guided fuzzing

Coverage-guided fuzzing leverages program feedback to refine the fuzzing process, primarily aiming to improve code coverage. The fuzzer slightly mutates seed inputs, discarding those that do not explore new execution paths (Böhme et al., 2019). Although many coverage-based fuzzers treat all code equally, real-world programs contain sections that require more attention due to their complexity.

[19] Interesting inputs are those that trigger new, unexplored parts of the program, without being blocked by early input format checks (van Rooij et al., 2021).

The core principle of coverage-guided fuzzing is to monitor the execution of a program and use the insights gained to guide the generation of testcases. For example, when a program crashes, the fuzzer examines the crash report to create new inputs that are likely to trigger the same bug again. In a similar vein, if a particular code path is executed, the fuzzer produces inputs specifically designed to explore that path in greater depth. Several tools support coverage-guided fuzzing, including `AFL`, `libFuzzer` and `Honggfuzz` (Swiecki & Gröbert, 2016). These tools instrument the application under test, monitor its execution, and utilize the gathered data to improve the fuzzing process.

Coverage-guided fuzzing has several advantages, including its ability to dynamically adapt test inputs, which makes it highly effective in identifying hard-to-detect bugs. It is also highly automated, capable of operating on large-scale codebases with minimal human intervention, and can be applied in both black-box and white-box testing scenarios.

A significant challenge with this approach is managing the large volume of data generated, especially when testing complex or extensive systems. To address this issue, many fuzzing tools include input minimization features. For instance, `AFL` offers a *minimization* function that automatically reduces the input data to the smallest size necessary to trigger the same bug. Beyond reducing data redundancy, such minimization techniques also serve as effective seed selection strategies for subsequent fuzzing campaigns, which is the central focus of this dissertation.

2.2.4 Input generation method

Fuzzers produce new testcases using either generation-based or mutation-based approaches. Generation-based fuzzers require prior knowledge of the program's input format (Shi et al., 2023). In file format fuzzing, they rely on a formal specification detailing encoding methods, data types, and file structure. Although some file formats have publicly available specifications, others remain proprietary and are not disclosed. Testcases are generated based on this knowledge, which makes them more likely to pass validation and reach deeper areas of the target program. However, effective analysis depends on access to comprehensive documentation.

Due to their flexibility and ease of use, mutation-based fuzzers are more commonly used in modern fuzzing techniques. These fuzzers do not require predefined specifications; instead, they modify valid initial inputs to create new testcases. They continuously mutate previously used input during the fuzzing process, making them more widely applicable across different software systems. Table 3 provides a comparison between generation-based and mutation-based fuzzers, as summarized by J. Li et al. (2018).

Fuzzers	Ease of start	Prior knowledge	Coverage	Ability to pass validation
Generation-based	hard	needed, hard to acquire	high	strong
Mutation-based	easy	not needed	low, affected by initial inputs	weak

Table 3: Comparison of generation- and mutation-based fuzzers (J. Li et al., 2018)

Mutation-based fuzzing

Mutation-based fuzzing focuses on modifying valid input data to create testcases (Hansen & Jensen, 2023), referred to as *mutants*. These modifications, known as *mutations*, involve altering bytes, flipping bits, or adding and removing characters, and can be applied randomly or according to predefined rules and patterns (Nunez et al., 2024).

This method effectively creates input data that remains valid for processing, but may lead to unexpected behavior. After the target program processes the mutated input, the fuzzer analyzes the execution results to identify errors or anomalies. Key advantages of mutation-based fuzzing include its independence from prior knowledge of the target system, its applicability to various data formats, including binary code, plain text, and structured data, as well as the ability to generate numerous testcases quickly.

Mutation engines produce new inputs by applying mutators and eliminating redundant mutants (Qian et al., 2022). Most fuzzing tools provide built-in mutators, some of which are dynamically selected during runtime (Lyu et al., 2019). Mutators define where and how input data should be modified, varying across fuzzing tools, which may also support custom mutators. Table 4 summarizes the mutators used by `AFL`.

Mutator	Granularity	Notes
Bitflips	bit	Flip single bit
Interesting values	byte, word, dword	NULL, -1, 0, etc.
Addition	byte, word, dword	Add random value
Subtraction	byte, word, dword	Subtract random value
Random value	byte(s)	Insert random value
Deletion	byte(s)	Delete from parent
Cloning	byte (unbound)	Clone/add from parent
Overwrite	byte (unbound)	Replace with random
Extra Overwrite	byte (unbound)	Extras: strings scraped from binary
Extra Insertion	byte (unbound)	Extras: strings scraped from binary

Table 4: Mutators used by `AFL` (Karamcheti et al., 2018)

A common mutator, bit-flipping, randomly alters bits in an input file to diversify testcases. Each bit has a probability of being flipped, controlled by a `mutation rate` that determines the degree of change per iteration. Higher mutation rates result in more substantial modifications. Although bit-flipping is straightforward and effective, it may not produce sufficiently distinct inputs to uncover new vulnerabilities. Arithmetic mutations

address this limitation by manipulating input data through operations such as addition or subtraction.

Mutation engines enable fuzzing tools to test diverse inputs, allowing for comprehensive program analysis. However, generation-based fuzzing, discussed in the next section, has been shown to execute significantly more code than mutation-based fuzzing (C. Miller & Peterson, 2007).

Generation-based fuzzing

Generation-based fuzzers employ probabilistic algorithms to produce syntactically valid input data according to predefined format rules. Unlike mutation-based fuzzers, which modify existing valid data, generation-based fuzzing does not use reference input; instead, it requires the manual study of relevant documentation to create testcases. Röning et al. (2002) applied this approach in their `PROTOS` framework using a simplified specification to describe protocol interactions.

As noted by Padhye et al. (2019), programs that handle complex structured data typically convert inputs into appropriate data structures through syntax parsing and semantic analysis before executing their core functionality. A generator must understand the format of acceptable input, such as ensuring that a string follows a valid address format. With that knowledge, testcases are generated to cover as many execution paths in the target program as possible. In addition, generators can apply common programming error patterns to create inputs that are more likely to trigger unexpected behavior.

Most generation-based fuzzers can produce inputs that pass integrity and grammatical checks (You et al., 2019), facilitating efficient progress beyond the initial analysis of the sample. However, J. Wang et al. (2017) observed that generic algorithms often fail to generate inputs that also pass semantic validation, thereby limiting the proportion of testcases that reach the target function within the program.

Despite the advantages of structured input generation, creating valid testcases for complex data structures or protocols remains a challenge, and this approach does not scale well for large software projects. A promising strategy involves combining mutation- and generation-based fuzzing techniques. Generation-based fuzzers create structured testcases tailored to specific program components, while mutation engines rapidly generate variations of these cases. This hybrid approach improves vulnerability detection. For example, `FormatFuzzer` features an integrated parser and generator designed to produce input that adheres to specific format specifications while also mutating them in a way that maintains their validity (Dutra et al., 2023).

2.3 Machine learning overview

ML is a subfield of AI that employs algorithms and statistical models to improve classification, prediction, and other problem-solving tasks using historical and real-time data (or feedback) (Dzeroski et al., 1970). Essentially, ML focuses on detecting patterns in data to inform decision-making (Mitchell, 1997). The underlying concept is that reality can be modeled by initially unknown mathematical functions that ML algorithms can uncover through the analysis of empirical data and leverage for various tasks (J. Chen et al., 2025). This actual learning procedure involves the model observing numerous known examples and their features, enabling it to provide a solution when encountering unknown data.

As illustrated in Figure 13, ML algorithms are generally classified into four categories: supervised, unsupervised, semi-supervised and reinforcement learning. Although some literature categorizes semi-supervised learning as a subset of supervised learning, in this work it is regarded as a distinct category, and this dissertation focuses on supervised and unsupervised learning methods, while briefly mentioning reinforcement learning. Supervised learning trains models on labeled data, whereas unsupervised learning enables algorithms to detect patterns in unlabeled data autonomously. Reinforcement learning trains models to make decisions based on feedback in the form of rewards or penalties (Han et al., 2012).

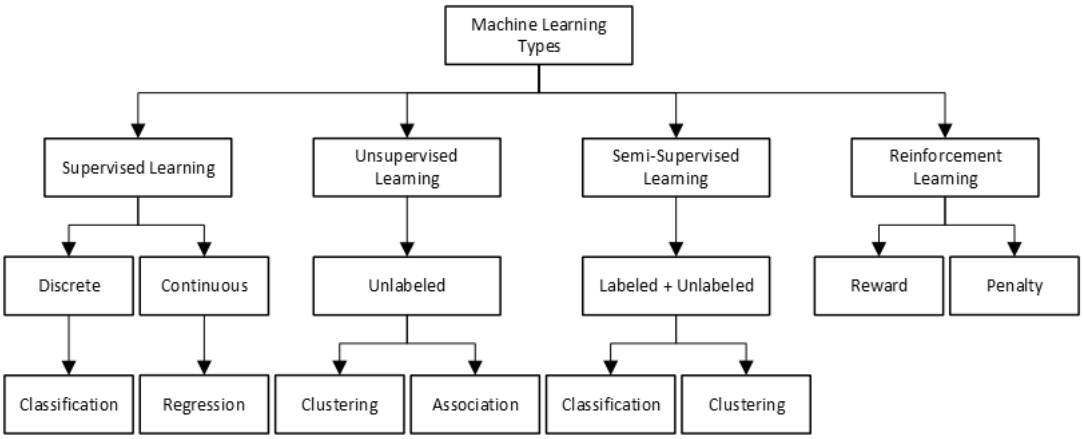


Figure 13: Various types of ML techniques (Sarker, 2021)

The performance of these algorithms depends heavily on both the quality and quantity of the training data. Their applications span across diverse domains, including image and speech recognition, natural language processing (NLP), fraud detection and recommendation systems (Yilmaz, 2023). Inspired by the structure of the human brain, NNs are one of the most prominent approaches to ML. As Abada et al. (2022) notes, each interconnected node functions as a mathematical operator that transforms its input into an output, which subsequently serves as an input for the following layers.

Training a NN involves optimizing its weights and biases to minimize the discrepancies between the expected and the actual output (Chapman, 2023). Optimization algorithms, such as gradient descent, iteratively adjust these parameters to converge on optimal values. Although effective, this training process is resource-intensive and requires a substantial amount of data. To address these challenges, researchers have developed specialized models, such as convolutional neural networks (CNNs) for image processing, recurrent neural networks (RNNs) for sequential data, and transfer learning (Zhuang et al., 2020) approaches that leverage pre-trained models[20]. Sarker (2021) provide a detailed summary of these techniques in Table 5.

Learning type	Model building	Examples
Supervised	Algorithms or models learn from labeled data (task-driven)	Classification, regression
Unsupervised	Algorithms or models learn from unlabeled data (data-driven)	Clustering, associations, dimensionality reduction
Semi-supervised	Models are built using combined data (labeled + unlabeled)	Classification, clustering
Reinforcement	Models are based on reward or penalty (environment-driven)	Classification, control

Table 5: Various types of ML techniques with example (Sarker, 2021)

Despite its strengths, ML faces limitations, including inherent biases in data and algorithms that can lead to discriminatory results in domains such as hiring or lending. Moreover, the opaque and complex nature of models like NNs hinders the interpretability of their decisions, a critical concern in areas such as finance and healthcare.

2.3.1 Supervised learning

Supervised learning, as described by Osisanwo et al. (2017), is a branch of ML in which a model is trained on labeled data to predict or classify previously unseen instances. In this approach, the input data is paired with its corresponding target value, and the model parameters are continuously adjusted to minimize the discrepancy between the predicted and actual outcome, until the model learns a function that maps inputs to outputs based on the known examples of established associations. The input data is usually represented as a set of features, while the output value can be either a class label (e.g., in classification) or a numerical value (e.g., in regression).

Models trained on labeled data can later produce accurate predictions on new instances, making them widely used in fields such as image and speech recognition, NLP, and predictive analytics, especially due to their ability to identify patterns and relationships between input features and output values in complex, high-dimensional datasets. There are a wide range of algorithms available, such as decision trees, NNs, support vec-

[20]Transfer learning aims to improve target models by leveraging knowledge acquired from models trained on related tasks or domains (Feuz & Cook, 2015).

tor machines (SVMs), and ensemble methods, that offer distinct advantages and trade-offs, hence selecting the appropriate technique depends on the specific context of the problem being addressed.

One of the key limitations of supervised learning is its reliance on labeled data (Y. Liu et al., 2022), which are often difficult to obtain in domains involving sensitive or confidential information, such as research on security vulnerabilities. Moreover, the quality of the training data significantly influences the performance of the model, biases or gaps in the dataset can undermine the reliability of the outcomes.

However, a more critical issue is overfitting, a fundamental challenge in supervised learning often caused by noisy and limited datasets, which hinders a model's ability to generalize to both known patterns in the training data and previously unseen instances in the test data (Ying, 2019). Various strategies exist to mitigate the negative effects of overfitting, including cross-validation (Webb, 2010), halting the training process before performance plateaus (Prechelt, 2012) or noise reduction in the training set (Srivastava et al., 2014). These approaches are also employed in the ML-based seed selection tool introduced later in this dissertation.

Furthermore, many supervised ML models render the prediction process opaque and difficult to interpret (Ables et al., 2024). This lack of transparency poses significant challenges in critical fields such as medical diagnosis and legal decision-making, where understanding the underlying rationale is essential. However, for the proposed ML-based seed selection strategy, this opacity is not a concern, as the primary focus is on the predictive performance of the model, meaning that the tool can effectively and reliably identify the most relevant seeds from the candidate set. To further illustrate supervised learning approaches, the following sections explore its two primary paradigms, which address discrete and continuous predictive tasks, respectively.

Classification

The most well-known type of supervised learning is classification, which assigns a class label to unknown input data based on the observable properties of the input using a training set of examples with known class membership. Predictive models like logistic regression and artificial neural networks (ANNs) analyze empirical data to identify trends and calculate binary outcomes (usually 0 or 1) for inputs. Because there is often no established formula that directly relates the observed features to the proper category, probabilities are used to represent this relationship, implying that the optimal prediction is to select the category with the highest likelihood (Dreiseitl & Ohno-Machado, 2002). Classification is widely applied in numerous real-world scenarios, such as object identification in image recognition, speech recognition in language interpretation, and text classification to categorize documents (Alpaydin, 2020).

Dreiseitl and Ohno-Machado (2002) demonstrates that ANNs are a generalized extension of logistic regression. Despite their distinct mathematical formulations, a NN using the `sigmoid` activation function without any hidden layers is equivalent to a logistic regression model. NNs can be regarded as collections of perceptrons and can achieve greater flexibility through the inclusion of nonlinear hidden neurons that allow the modeling of complex, nonlinear decision boundaries. Reflecting this theoretical progression, the initial prototype of the proposed ML-based seed selection strategy, as outlined in my thesis proposal, began as a simple logistic regression model, which later evolved into an ANN through the introduction of hidden layers.

In the context of classification, features refer to the measurable attributes or characteristics of the input data, while labels represent the corresponding categories assigned to each instance (Kotsiantis et al., 2007). Models are constructed using labeled training data that capture the underlying associations between these features and labels (Kumari & Srivastava, 2017), then their predictive performance is evaluated using unlabeled test data, where the model's ability to accurately classify new instances is assessed.

Various classification techniques offer several advantages, including efficiency, high accuracy, interpretability, and versatility. When combined with suitable feature selection and parameter tuning, these techniques can manage large datasets effectively while achieving robust predictive performance (Sofia, 2024). Moreover, some models, for example, decision trees, are inherently interpretable, which broadens their applicability across various domains.

Given the wide range of classification techniques available, the scope of this research did not extend to the evaluation of multiple models. Instead, implementing a single method and assessing its viability in a real-world case study was sufficient to meet the research objectives. As Kotsiantis et al. (2007) demonstrates, the selection of a classification method depends on understanding the specific conditions under which a particular approach outperforms others in a given application. With a clear view of the strengths and weaknesses of each technique, future research may benefit from exploring the integration of multiple algorithms to better address the present research problem.

Regression

Moving from classification to regression reveals the shared foundations and distinct objectives of these supervised learning techniques. Both paradigms rely on identifying and modeling the relationships between features and outcomes using training data; however, they diverge in their evaluation criteria and underlying assumptions. Regression and classification are distinguished primarily by the nature of their output. Although classification involves predicting discrete class labels or categories, in regression, the goal is to predict continuous numerical values based on input features (Fahrmeir et al., 2022).

Regression models are often designed to capture trends and estimate values along a continuum, whereas classification models, including logistic regression or ANNs, emphasize the delineation of boundaries between distinct categories. Recent research also highlights that these paradigms are not entirely isolated. In some cases, regression problems can be reformulated as classification tasks, an approach that may take advantage of differences in learning bias and the representation of the underlying features (Kotsiantis et al., 2007; Stewart et al., 2023).

As also summarized by James et al. (2013), problems with numerical outcomes are typically referred to as regression problems, while those with qualitative responses, such as class or category labels, fall under classification problems. However, since logistic regression estimates class probabilities, it is predominantly used for binary classification tasks. Given these distinctions in problem formulation, the evaluation of these approaches also differs substantially. For regression tasks, performance is typically assessed using error metrics such as mean squared error (MSE) or mean absolute error (MAE), along with an analysis of residuals and uncertainty measures like confidence intervals. In contrast, classification models are evaluated based on metrics such as accuracy, precision, and recall, often using confusion matrices to examine the distribution of errors (James et al., 2013; Kotsiantis et al., 2007).

2.3.2 Unsupervised learning

According to Hastie et al. (2009), unsupervised learning is ”*learning without a teacher*”. It involves analyzing a dataset composed solely of input observations, without the corresponding output labels, to infer the underlying structure of the data. Unlike supervised learning, where a ”*teacher*” provides explicit error signals that guide prediction accuracy, unsupervised methods must rely on heuristic and often indirect measures of performance, since no ground-truth responses are available. Consequently, this learning paradigm focuses on identifying salient features such as low-dimensional manifolds, clusters, or high-density regions within high-dimensional data, employing techniques like principal component analysis (PCA), multidimensional scaling, and mixture modeling to reveal latent structures.

In the following sections, I briefly summarize common unsupervised learning problems, such as clustering, association, and generative modeling. Clustering techniques focus on partitioning a dataset into distinct groups by identifying convex regions in the input space where data points are densely concentrated. This approach suggests that the overall data distribution can be approximated as a mixture of simpler distributions, each corresponding to a particular type or class of observation. In contrast, learning of association rules is tailored for high-dimensional binary data. It seeks to form simple conjunctive rules that capture specific regions of high data density, providing clear and interpretable descriptions of patterns within the dataset. Generative models like LLMs or generative adversarial

networks (GANs) are designed to synthesize data that closely mimic real-world examples. Feature selection techniques bridge the gap between raw high-dimensional data and these methods by transforming complex feature spaces into simpler, lower-dimensional representations, thereby enhancing the effectiveness and interpretability of unsupervised learning (Pena et al., 2001).

Clustering

Clustering, as described by Aggarwal and Reddy (2018) and also thoroughly explored by Dinh et al. (2025), is an unsupervised learning technique that categorizes unlabeled samples into groups based on their inherent similarity. The primary objective is to maximize similarity within clusters while minimizing similarity between them (Pitafi et al., 2023).

This method is particularly valuable in feature selection, where the relevance of individual features may be ambiguous in the absence of labels. Identifying the appropriate features is crucial to mitigating challenges associated with high-dimensional spaces, which can otherwise result in computational inefficiencies and diminished learning performance. As data continues to grow exponentially, manual labeling has become impractical, underscoring the importance of automated clustering. This technique serves a dual purpose: it facilitates the discovery of hidden patterns and functions as a preprocessing step for subsequent DM and ML tasks.

In general, clustering methods can be categorized into three main approaches: partitioning, hierarchical, and density-based approaches. Partitioning methods, such as k-means and k-medoids, employ distance-based metrics to form non-overlapping clusters. Hierarchical techniques, in contrast, construct multi-level groupings that enhance visualization. Meanwhile, density-based methods, exemplified by DBSCAN and OPTICS, detect clusters of arbitrary shapes by identifying densely packed data points and distinguishing them from low-density regions.

Evaluating the quality of the clustering is essential to ensure the reliability of the generated clusters. Internal validation measures, including the silhouette coefficient, Dunn index, and Calinski–Harabasz index (CHI), assess cluster cohesion, while external indices, such as the adjusted Rand index and Normalized Mutual Information, gauge stability and optimal cluster determination. Vysala and Gomes (2020) propose a rigorous validation framework, emphasizing the importance of these methods in producing meaningful and interpretable clustering results.

Association

While clustering is about grouping similar observations into clusters, association is about identifying frequently co-occurring items or attributes to form interpretable rules of association. Association analysis is a DM technique designed to reveal latent relationships

among variables or items in large data sets, thereby uncovering patterns that enhance predictive capabilities and inform decision-making. At its core, the method employs association rules, typically formulated as "*if-then*" statements, to articulate probabilistic links between items. For example, a rule might state that "*if a customer buys milk and bread, then they are likely to buy eggs*". The strength of these rules is evaluated using two fundamental measures: support, which quantifies the frequency with which an itemset appears in the dataset, and confidence, which assesses the likelihood that the presence of a given itemset implies the concurrent occurrence of another.

The discovery of frequent itemsets, groups of items that co-occur above a specified frequency threshold, is critical to this analytical framework. This process, known as itemset mining, is robustly implemented by algorithms such as `Apriori`, which iteratively generates candidate itemsets and systematically prunes those that do not satisfy the minimum support criterion (Agrawal et al., 1996).

Based on the identification of frequent itemsets, association rule mining extracts statistically significant rules that elucidate the interdependencies between items (Agrawal et al., 1993). In practice, frequent itemsets serve as the basis for generating candidate rules, which are then refined by applying a confidence threshold to ensure that only robust associations are retained. This methodology is widely applied in various fields, including market basket analysis, customer behavior modeling, and web log mining, where understanding item co-occurrences can yield actionable insights.

Notwithstanding its efficacy, association analysis is challenged by the combinatorial explosion inherent in large datasets and issues related to data sparsity. To mitigate these challenges, researchers have proposed alternative strategies, such as pattern-growth algorithms, which bypass exhaustive candidate generation by directly identifying frequent patterns, and parallel processing techniques that significantly reduce computational overhead. Together, these advances improve both the precision and the scalability of association analysis in increasingly complex data environments.

Large language models

Clustering and association analysis employ statistical methods to reveal underlying data patterns, whereas LLMs leverage self-supervised DL techniques to capture complex language structures and generate coherent text. LLMs have emerged as a groundbreaking class of AI systems that excel in processing and generating human-like text. These models leverage self-supervised learning[21] on vast and diverse text corpora, ranging from books and scholarly articles to web pages and code repositories, to learn intricate linguistic patterns without relying on manually labeled data (Brown et al., 2020). The underpinning of

[21] Self-supervised models are fundamentally unsupervised; however, they can learn useful representations from unlabeled data by generating pseudo-labels or pretext tasks from the data itself, thus they incorporate elements of supervision (Jing & Tian, 2021).

these capabilities is the transformer architecture, which enables LLMs to capture long-range dependencies and manage complex contextual relationships within text, separating them from earlier DL approaches.

The typical training paradigm for LLMs involves an unsupervised pretraining phase followed by a fine-tuning phase for task-specific applications. During pretraining, the model builds a robust understanding of syntax and semantics from the raw data, and fine-tuning subsequently refines this knowledge for specialized tasks such as language translation, summarization, or even security vulnerability analysis. This two-stage process has paved the way for few-shot learning capabilities, as exemplified by models such as GPT-3, which have exhibited impressive performance even when provided with minimal task-specific examples. Furthermore, studies on model scaling have shown that increasing the number of parameters and training data can significantly enhance performance, although such improvements often come with increased computational demands.

Despite their transformative impact on NLP, LLMs introduce several challenges that the research community is actively addressing. Concerns such as inherent biases in training data, hallucinations of inaccurate information, and extensive resource requirements for training and inference have prompted ongoing investigations into model interpretability and efficiency. Comprehensive reviews have detailed these issues while also mapping out future research directions to mitigate risks and optimize capabilities (Mohan et al., 2024; Naveed et al., 2024). As LLMs continue to evolve, balancing their remarkable potential with ethical and practical considerations remains a central focus of contemporary AI research.

Generative adversarial networks

A GAN consists of two NNs, the generator and the discriminator, which engage in an adversarial process to refine synthetic data generation (Zhong et al., 2024). The generator produces artificial samples, such as images or text, while the discriminator learns to distinguish these from real data (I. Goodfellow et al., 2014). Through continuous feedback, the generator enhances its output quality, while the discriminator sharpens its ability to identify inaccuracies. This interplay enables GANs to capture intricate patterns and structures within data, making them highly effective for applications such as image synthesis, data enhancement, and style transfer.

Building on this dynamic framework, GANs have become a cornerstone of generative modeling, driving significant advances in DL. Introduced by I. J. Goodfellow et al. (2014), the formulation of GANs as a two-player minimax game has established a foundation for ongoing innovation. The generator's iterative refinement process, guided by adversarial training, has led to increasingly sophisticated models capable of generating high-fidelity data samples from complex distributions. As research continues to explore their potential, GANs remain central to the evolution of AI-driven content generation.

Building on this foundation, subsequent advancements have focused on improving the performance and robustness of GANs. Mirza and Osindero (2014) extended the original framework by proposing conditional GANs, which incorporates auxiliary information to guide the generation process toward specific class labels or attributes. In parallel, Radford et al. (2016) introduced deep convolutional GANs, leveraging convolutional architectures to stabilize training and produce sharper and more realistic images, thus broadening their practical applicability.

To address inherent training challenges such as mode collapse and instability, Arjovsky et al. (2017) developed the Wasserstein GAN, redefining the loss function using the Wasserstein distance. This approach improved training convergence while providing a more meaningful metric for generator performance. More recently, Karras et al. (2019) proposed a style-based generator architecture that enables finer control over the synthesis process, facilitating the creation of high-resolution images with intricate details and diverse visual styles.

Dimensionality reduction

Feature selection techniques often serve as an intermediate step by simplifying high-dimensional data into interpretable lower-dimensional representations. This process not only improves the performance of clustering and association analyzes, but also facilitates the visualization and interpretation of the latent spaces learned by advanced generative models, such as LLMs and GANs.

The goal of dimensionality reduction is to decrease the number of features in high-dimensional datasets while preserving essential information. This technique is especially useful when handling large and complex datasets, which may consist of thousands or even millions of features (Ayesha et al., 2020).

Two primary approaches are used to achieve dimensionality reduction: feature selection and feature extraction. Feature selection involves selecting a subset of original features based on specific criteria, such as correlation or mutual information with the target variable. In contrast, feature extraction consists of transforming the original features into a new set using linear or non-linear methods (Pandit, 2023).

PCA is a widely used technique for feature extraction. Identifies a set of orthogonal components ordered by the variance they capture, thus encapsulating the most significant information from the original dataset. PCA is particularly effective in reducing dimensionality in highly correlated data and facilitates visualization in lower-dimensional spaces (Jolliffe, 2011).

Another notable technique is t-distributed stochastic neighbor embedding (t-SNE), which reduces the dimensionality while preserving the local structure of the data. As a non-linear method, t-SNE accentuates differences between data points, enabling visualization in two or three dimensions (Van der Maaten & Hinton, 2008).

In the present research, PCA is employed solely for visualization to represent the most influential features in the training data, providing a compact depiction of the primary variance. Subsequently, t-SNE is used to project the data into two- and three-dimensional spaces, allowing for a detailed exploration of the intrinsic structure and relationships within the training dataset.

2.3.3 Reinforcement learning

Supervised and unsupervised learning build predictive and descriptive models from pre-collected, static datasets, while RL focuses on enabling agents to learn optimal actions through interactive and dynamic environments. It is a subfield of ML that emphasizes optimizing decision making through iterative trial and error, a paradigm thoroughly detailed by Kaelbling et al. (1996).

In an RL framework, an agent interacts with its environment by performing actions and receiving corresponding feedback, typically in the form of rewards or penalties (R. S. Sutton, Barto, et al., 1998). This cyclical process enables the agent to adjust its behavior over time to maximize the long-term cumulative rewards. By balancing exploration (i.e., testing new actions) with exploitation (i.e., leveraging known strategies), RL effectively navigates complex and dynamic environments where conditions continuously evolve (H. Wang et al., 2019). This method's inherent adaptability in such settings has spurred its successful application in fields such as robotics and autonomous systems, as reviewed by Ghasemi et al. (2025).

RL also enables autonomous task learning without explicit human guidance, which makes it particularly suitable for applications where labeled data are scarce or costly, such as in robotics or gaming. Additionally, RL algorithms can optimize complex non-linear objectives that often exceed the capabilities of traditional methods (R. S. Sutton, Barto, et al., 1998); this is especially valuable in scenarios that prioritize profit maximization or risk minimization. Furthermore, their ability to adapt to changing environments makes them ideal for dynamic applications, such as autonomous driving (Izquierdo & Izquierdo, 2012).

However, RL algorithms tend to be computationally intensive, particularly in complex environments or when navigating large action spaces (Kumar et al., 2023). In addition, their sensitivity to hyperparameter settings can complicate the optimization process (Eimer et al., 2023), and limited training data may impede their ability to generalize effectively to new tasks or environments (Korkmaz, 2024).

3. Theoretical concept

3.1 Related work

Fuzzing is a widely used technique for uncovering vulnerabilities in software systems. The method involves generating input designed to cause errors or crashes in a program, followed by an analysis of the results to detect potential security flaws. However, traditional fuzzing approaches are limited by low efficiency and code coverage, as random input generation often fails to exercise specific program paths or trigger certain bugs. Recent studies have explored ML techniques to overcome these limitations. This chapter examines the relevant literature that has influenced the research presented in this dissertation.

Mutational gray-box fuzzing depends on valid seed samples to initiate the testing process. Common approaches for constructing a test corpus include using an empty file, a single instance of the target input format, or a large set of samples collected from the Internet. Herrera et al. (2021) analyzed the impact of seed selection on mutation-based gray-box fuzzing, emphasizing its critical but often overlooked role in fuzzing effectiveness. Through a comprehensive review of seed selection strategies and an extensive empirical evaluation, the authors evaluated six distinct approaches, including three corpus minimization techniques. Their findings confirm that the initial fuzzing corpus has a significant influence on the results, affecting the efficiency of the fuzzers in discovering vulnerabilities.

Although using a single, small, representative seed may seem like a reasonable baseline for fuzzer comparison, the study revealed that this approach introduces high variance, potentially leading to misleading performance evaluations. Moreover, it does not align with real-world fuzzing practices, where large seed corpora are commonly employed. To improve fuzzing results, the study advocates for a deliberate and well-documented seed selection process to collect and minimize diverse seed sets, ensuring that fuzzing campaigns begin with optimal input corpora.

Saavedra et al. (2019) provides a comprehensive review of research that integrates ML technologies with fuzz testing. The study examines various paradigms, including supervised, unsupervised, and reinforcement learning, to enhance the effectiveness of fuzzing. The review further investigates the application of ML at various stages of the fuzzing process and addresses the challenges associated with integrating ML techniques. The authors conclude that unsupervised methods, which detect patterns and similarities between data points, are particularly promising. Tools such as `AFL` have incorporated unsupervised algorithms into their workflows to improve fuzzing effectiveness.

The authors also discuss the potential of DL and RL to enhance the fuzzing process.

Although these methods have not been widely adopted, they offer promising avenues for improvement; however, challenges such as insufficient training data and computationally prohibitive training times have constrained their application in certain aspects of fuzz testing. The paper concludes that, although ML has significantly improved the effectiveness of fuzzing, many open research challenges remain. The authors suggest that further research will likely lead to a broader application of ML to remove bottlenecks in the fuzzing process.

According to Lyu et al. (2018), modern applications require advanced fuzzing techniques for security testing; however, such techniques restrict the flexibility and adaptability of fuzzers. Researchers have addressed these challenges by focusing on two primary areas: developing advanced fuzzing tools and improving seed sets for fuzz testing.

Generation-based fuzzing techniques create well-structured input files that conform to the target program's format, thereby augmenting crash detection. In contrast, mutation-based methods modify existing seed sets but frequently contend with rudimentary crash detection strategies. To further optimize performance, techniques such as static analysis, taint analysis (Clause et al., 2007), and symbolic execution (Xie et al., 2020) are integrated. Moreover, initiating fuzzing campaigns with an improved seed set has been shown to improve efficiency (Rebert et al., 2014). In this dissertation, I adopted the latter approach by designing a generic seed selection method that uses ML as an alternative to current techniques.

Recent research has demonstrated the potential of ML to improve the effectiveness of fuzzing techniques. As software systems become increasingly complex and challenging to secure, such approaches may become essential tools for identifying and mitigating security vulnerabilities. The following sections examine previous work that contributed to the identification of the research problem and provided a direct justification for the research questions.

3.1.1 Grammar-based white-box fuzzing

White-box fuzzing is a powerful dynamic test generation method for security testing, yet its applicability is often limited to programs that require highly structured inputs, such as compilers and interpreters (Godefroid et al., 2007). These applications process data in multiple stages, resulting in an overwhelming number of control paths that hinder deeper exploration. To mitigate this limitation, Godefroid, Kiezun, and Levin (2008) introduced grammar-based white-box fuzzing, an approach that enhances traditional white-box fuzzing by integrating a grammar-based specification of valid inputs. By generating constraints via symbolic execution and solving them using a custom grammar-based constraint solver, this method ensures that inputs remain contextually valid, preventing dead-end paths and improving test efficiency.

Unlike black-box fuzzing, which modifies inputs randomly to discover vulnerabilities, white-box fuzzing systematically explores program paths using symbolic execution and constraint solving. However, applications that process complex input formats present challenges due to excessive path complexity and limitations in symbolic execution, such as hash-based keyword detection, which obstructs effective test case generation. Grammar-based white-box fuzzing addresses these issues by generating constraints at the level of grammar tokens rather than raw input bytes. A custom constraint solver ensures that the generated inputs comply with a specified grammar, facilitating deeper exploration while eliminating ineffective testcases resulting from inputs that the application cannot understand. Although grammar-based specifications enable structured input applications to progress beyond initial processing stages, they do not fully overcome the inherent constraints of symbolic execution in reaching the deepest layers of complex programs. Certain constructs within compilers and interpreters may still pose unresolved challenges that cannot be adequately addressed solely through grammar-based approaches.

The effectiveness of this method was demonstrated through extensive evaluation in the JavaScript interpreter of Internet Explorer 7, where grammar-based white-box fuzzing significantly improved test coverage and efficiency. The coverage of the code generation module increased from 53% to 81%, while the number of required tests was reduced by a factor of three. In addition, deeper program layers became more accessible and reachability improved from 72% to 80%. These results highlight the ability of the approach to navigate complex application structures effectively. By integrating the strengths of constraint-based white-box testing and grammar-based black-box techniques, grammar-based white-box fuzzing emerges as a more effective tool for automated security testing of applications with highly structured inputs, addressing key limitations of conventional fuzzing techniques.

Despite its advantages, grammar-based white-box fuzzing remains inherently tailored to applications that process very complex inputs. Its reliance on symbolic execution and constraint generation introduces precision limitations, particularly in deeper program layers, where grammar specifications alone may not be sufficient. Furthermore, the need for a custom-built grammar-based constraint solver can complicate adaptation to diverse contexts.

3.1.2 Optimizing seed selection for fuzzing

Rebert et al. (2014) explores the optimization of seed selection strategies in fuzzing to maximize the number of software bugs found. Random sampling, used as the baseline method, exhibits substantial variability in seed quality and performance. This high variance underscores the limitations of relying solely on random selection and motivates the use of more systematic approaches. The authors propose and evaluate the following five approaches.

- **Hot set:** Selecting seeds that produced the highest number of unique bugs after a short fuzzing period under the assumption that a buggy program is more likely to have more bugs.
- **Unweighted minimal set:** Choosing the smallest set of seeds that achieves maximum code coverage, which means the fewest seed files needed to reach the most instructions.
- **Time-weighted minimal set:** Preferring seeds that execute quickly, allowing for higher fuzzing throughput, meaning more mutations and testcases, consequently, a higher likelihood of bug detection.
- **Size-weighted minimal set:** Selecting smaller seeds under the assumption that many small seed files that cover a few code blocks may be more beneficial than a single large file that covers numerous code blocks.
- **Peach minimal set:** Following the seed selection method used by the Peach fuzzer, which is based on code coverage, but does not necessarily minimize the seed set.

One of their key findings is that the standard coverage-guided unweighted minimal seed selection method consistently outperformed other approaches in terms of bugs found, confirming the effectiveness of using coverage-driven selection methods. Furthermore, the study demonstrates the advantages of strategic selection as a well-chosen subset of seed files leads to more efficient bug discovery than simply fuzzing with all available inputs. Selecting fewer, higher-quality seeds not only increases effectiveness but also reduces computational costs.

Another important conclusion is that optimized seed sets for one program often work well for other applications that process the same file format. This suggests that precomputing seed sets can be a practical way to enhance fuzzing efficiency across different software. Furthermore, the study highlights the role of scheduling algorithms, finding that Round-Robin scheduling, when combined with a strong seed selection strategy, helps maximize fuzzing performance.

Through extensive experiments, the study led to the discovery of 240 bugs in eight applications, providing strong evidence that proper seed selection significantly improves the effectiveness of fuzzing, reduces computational costs, and maximizes bug discovery. The findings support the broader adoption of coverage-driven strategies in fuzz testing.

3.1.3 Data-driven seed generation for fuzzing

J. Wang et al. (2017) introduces `Skyfire`, a novel data-driven seed generation approach designed to optimize fuzzing for programs that process highly structured inputs. Traditional techniques, such as mutation-based and generation-based fuzzing, struggle to reach deep execution stages because they are often rejected early during syntax parsing or fail during semantic checking. `Skyfire` addresses this limitation by taking advantage of a large corpus of existing samples to learn a probabilistic context-sensitive grammar

(PCSG), which captures both syntax features and semantic rules. This learned grammar enables the generation of well-distributed, semantically valid seed inputs, which increases the effectiveness of fuzzing.

By leveraging its learned probabilistic context-sensitive grammar, `Skyfire` not only improves seed generation but also refines the overall fuzzing process, focusing on three essential objectives: producing correct, diverse, and uncommon seeds. Correct seeds ensure that most generated inputs are syntactically and semantically valid, preventing fuzzing from getting stuck in early processing stages such as syntax parsing and semantic checking. This accelerates the exploration of the application execution stage, where deep bugs typically reside. Diverse seeds introduce a broad range of inputs with varying grammar structures and semantic properties, enabling fuzzers to cover different function modules that traditional mutation-based techniques struggle to reach. Uncommon seeds help fuzzers target less-tested code paths, increasing the likelihood of discovering previously unknown vulnerabilities.

The two-step process of `Skyfire` begins with parsing a corpus based on a predefined grammar to construct abstract syntax trees (ASTs) and subsequently extracting a PCSG that incorporates production rules conditioned on contextual probabilities. In the second step, seed inputs are generated by iteratively applying production rules while maintaining grammatical correctness. Additionally, `Skyfire` refines these seeds through selection mechanisms that eliminate redundancy and seed mutation strategies that introduce semantic variations while preserving structure.

Empirical evaluations demonstrate the efficacy of the tool in fuzzing open-source `XSLT` and `XML` engines, as well as closed-source components in Internet Explorer 11. The approach yielded a significant increase in code coverage and led to the discovery of 19 memory corruption bugs, including 16 previously unknown vulnerabilities, as well as 32 denial-of-service bugs.

3.1.4 Machine learning for input fuzzing

Traditional grammar-based fuzzing remains the most effective technique for testing software applications with complex structured input formats. However, it relies on manually crafted grammars, which are time-consuming and prone to errors. Godefroid et al. (2017) introduce a novel approach that integrates ML into fuzz testing to automate input generation, eliminating the need for format-specific customization. The proposed algorithm leverages character-level RNNs to generate new test inputs based on the probability distribution of non-binary Portable Document Format (PDF) data objects.

A detailed case study on the PDF format and the built-in PDF parser of the Microsoft Edge browser highlights the tension between learning and fuzzing objectives. While learning aims to preserve the structure of well-formed inputs, fuzzing seeks to disrupt that structure to uncover vulnerabilities. The algorithm strategically utilizes the learned prob-

ability distribution to guide the fuzzing process, outperforming traditional random fuzzing in terms of code coverage.

For each test execution, they measured code coverage, parsing errors, and the number of bugs found. All learning-based algorithms performed competitively compared to the baseline, with three achieving higher code coverage. A key insight from the experiments is the inherent tension between ML and fuzz testing: improving the ML model does not necessarily increase the effectiveness of fuzzing. Specifically, applying a random fuzzing algorithm to well-formed objects reduces parsing errors while increasing code coverage, as malformed inputs trigger error-handling code during execution. This trade-off between code coverage and the rate of parsing errors highlights the need to strike a balance between these objectives for optimal results. I observed a similar effect in my experiments: training the model on the complete dataset improved performance, but the predicted seed samples became overly focused on the learned patterns, limiting the fuzzer's ability to further explore the application under test. One possible explanation for my observation is overfitting, a well-known phenomenon in ML, where the model becomes overly specialized to the training data, leading to reduced generalization and effectiveness.

The study concludes that effective fuzzing requires generating both well-formed inputs (with a high pass rate) and malformed inputs (with a low pass rate) to maximize coverage of diverse parsing and error-handling code, ultimately increasing the likelihood of detecting vulnerabilities. A crucial implication is that more precise learning models do not necessarily yield better fuzzing results. Furthermore, the computational cost of training a model must be considered when evaluating its overall effectiveness. The study provides empirical evidence that high-accuracy models are not strictly necessary for successful fuzzing.

Building on this insight, subsequent work by Zakeri Nasrabadi et al. (2021) continues to explore how ML can be integrated into fuzzers without falling into the trap of overfitting or diminishing the exploratory nature that makes fuzzing effective. Their research reinforces the general lesson that fuzzing, unlike traditional prediction tasks, benefits from imperfect learning. Embracing uncertainty and introducing a degree of controlled error in generated input is not a drawback, but a powerful strategy for exposing software flaws. Although improved learning can enhance fuzzing, this only holds when it is paired with techniques that preserve input diversity and foster exploration; otherwise, the fuzzer may become too ”well-behaved” to uncover the most elusive bugs.

3.1.5 Reinitialization with deep neural networks

Nichols et al. (2017) presents an approach to improve fuzz testing using DNNs by integrating GANs into the `AFL` fuzzer to generate novel seed files to restart test campaigns. As usual, the primary metric for evaluation is the rate at which previously unseen code paths are discovered. Experimental results demonstrate that GAN-based reinitialization

outperforms both an long short-term memory (LSTM) model and random augmentation strategies, enabling the AFL fuzzer to uncover more unique and longer code paths within the same time frame. These findings suggest that GAN-based techniques hold promise in guiding fuzzers to deeper and potentially security-sensitive code regions.

The study builds on several software testing methodologies, including symbolic execution, property-based testing, and satisfiability solvers. The AFL fuzzer detects software flaws through random mutations and maintains seed files that trigger unique execution paths. This research proposes leveraging these native seed files as training data for deep generative models, thereby introducing novel and diverse inputs. Although ML techniques have been previously explored in fuzzing, this work specifically examines whether DNNs can enhance the effectiveness of AFL. By periodically restarting the fuzzer with GANs-generated seed files, the approach aims to accelerate the detection of vulnerabilities.

The results indicate that AFL benefits most when restarted mid-course with GANs-generated seed files. Although synthetic seed files yield comparable path lengths on average, GAN-based reinitialization consistently surpasses both random augmentation and LSTM models in discovering unique execution paths. The LSTM model, in particular, suffers from slower training times and reduced effectiveness in code exploration. Notably, neither GAN nor random augmentation methods require manual analysis or format-specific knowledge of the target program. Furthermore, the study suggests that exposing the internal program state could refine reinforcement learning (RL) strategies for fuzz testing.

3.1.6 Targeted mutation strategy for increasing code coverage

AFL has been effective in identifying bugs and security vulnerabilities in various software applications. However, its ability to thoroughly explore programs remains limited, particularly in sections constrained by specific input requirements, as it does not account for which input elements are essential for deeper analysis. To address this, Lemieux and Sen (2018) introduced FairFuzz, an enhancement that prioritizes input samples activating rare code paths while applying targeted mutations to preserve critical features and promote deeper exploration of the PUT. FairFuzz employs two key strategies to achieve this: first, it selects inputs from the queue that trigger rare code paths for mutation; second, it refines the mutation process to maximize the likelihood that modified inputs will continue reaching the targeted paths.

A comprehensive evaluation of FairFuzz against multiple AFL variants using real-world benchmarks measured its impact on the speed and extent of code coverage, specifically assessing whether FairFuzz increases the proportion of testcases targeting rare code paths. Results indicate that FairFuzz significantly improves code coverage or achieves similar gains at a faster rate, while performing particularly well in applications with deeply nested conditionals.

Despite its strengths, FairFuzz inherits a notable limitation: it cannot target program sections that AFL cannot reach. For instance, AFL struggles with multibyte comparisons unless progress leads to new coverage, a challenge FairFuzz retains. To address this, researchers propose incorporating Low Level Virtual Machine (LLVM) transformations, which decompose multibyte comparisons into byte-level evaluations, thereby improving the ability of AFL to track incremental progress. However, FairFuzz has yet to be tested in a parallel fuzzing setup, where such instrumentation improvements could be further examined.

By structuring its approach around the prioritization of rare paths, Lemieux (2021) demonstrates how guided input mutation strategies can boost fuzzing effectiveness and improve bug detection in software applications. In addition, it underscores the importance of evaluating fuzzing tools through repeated experiments and statistical variability analysis to ensure robust comparisons among competing techniques.

3.1.7 Coverage-based gray-box fuzzing as Markov chain

Böhme et al. (2019) examines coverage-guided graybox fuzzing (CGF) as a method of automated vulnerability discovery, which operates by mutating seed inputs and preserving those that reveal new execution paths. However, conventional fuzzers such as AFL, tend to overexplore high-frequency paths, diminishing efficiency. To address this, AFLFast implements a refined seed selection strategy that prioritizes low-frequency paths, thus accelerating vulnerability detection. By modeling fuzzing as a Markov chain, the study formalizes the probability of a seed mutation leading to a new execution path, leveraging a structured power schedule and search strategy to significantly improve the discovery rate of program crashes.

Symbolic execution systematically enumerates program paths, but incurs substantial computational overhead. Although it remains valuable for identifying vulnerabilities hidden deep in the program, gray-box fuzzing provides a scalable alternative for more efficient detection of software flaws, particularly with the optimizations of AFLFast. Empirical evaluations highlight the superiority of AFLFast in vulnerability discovery compared to AFL and KLEE, revealing numerous previously undiscovered bugs while increasing code coverage. These findings underscore the importance of refining fuzzing methodologies to broaden the scope of increasing code coverage while also maintaining practical feasibility.

AFLFast enhances the effectiveness of CGF by directing the fuzzer toward rarely executed code paths, leading to the faster identification of security vulnerabilities. The Markov chain modeling offers valuable insights regarding the inner workings of AFL and its extensions, demonstrating how strategic control over seed selection and mutation strategies optimizes fuzzing efficiency without the computational burden of symbolic analysis. In summary, while ML techniques in fuzzing primarily focus on guiding the generation and mutation of seed inputs to improve code coverage, alternative methods, such

as static analysis, often require manual verification. These observations underscore the need for further research to advance the practical effectiveness of approaches using ML in the security testing of software applications.

3.2 Proposed framework

3.2.1 Research motivation

Coverage-guided mutational fuzzers produce thousands, if not millions, of random mutations to uncover various software flaws and security vulnerabilities. However, the effectiveness of these mutations varies significantly, making it both challenging and time-consuming to identify the most promising inputs. Mutations that trigger error-checking routines in the target application may increase code coverage by uncovering new execution paths. However, this rarely results in the discovery of new bugs, which is the purpose of fuzzing. Consequently, there is growing interest in advanced automated techniques that improve the efficiency of fuzzing and optimize the overall testing process.

To further elaborate on this issue, existing fuzzing tools often treat all seed inputs equally and do not account for the security-related differences in the code paths they traverse (Y. Wang et al., 2020). Research indicates that software bugs are unevenly distributed, with approximately 80% concentrated in just 20% of the codebase (Ostrand & Weyuker, 2002; Shin & Williams, 2013). A study of `AFLGo` by Zong et al. (2020) found that, on average, more than 91.7% of the inputs failed to reach the buggy code, limiting the coverage of the test and reducing the effectiveness of the fuzzer. Furthermore, Herrera et al. (2021) emphasizes that the selection of initial seed inputs plays a crucial role in fuzzing. Addressing these challenges requires more sophisticated algorithms that prioritize seed input considering various perspectives and methodologies (Canakci et al., 2022), beyond those explored in previous studies, for example, based on the relative vulnerability of execution paths (Hayes et al., 2019; Herrera et al., 2020).

The present study introduces `SeedClass` as an alternative seed selection strategy designed to improve bug discovery by leveraging insights from previous test campaigns to prioritize valuable seed files. The proposed approach treats input as patterns and feeds them to a NN-based model to predict which seed files are the most likely to trigger program crashes. By training a model on a dataset of examples labeled according to past fuzzing outcomes, it can predict whether new, unseen samples will lead to program failure without the need to execute the target program.

The effectiveness of a gray-box fuzzer is closely tied to its execution speed. Therefore, any optimization must minimize the performance overhead to avoid affecting the fuzzing efficiency. Although training a NN can be time-intensive, this process occurs offline. It is only required once, making the time investment minimal compared to the long-term

benefits of increased accuracy and reliability in subsequent test campaigns.

A well-balanced and labeled dataset is critical for training a reliable model. However, discrepancies between the newly generated input and the training set can affect the accuracy of the prediction. In addition, minimizing the time required for both training and prediction is imperative. This work focuses on optimizing the effectiveness of existing coverage-guided fuzzers by directing them toward security-sensitive code paths, thereby allowing the identification of more vulnerabilities in less time while remaining computationally efficient enough to run on a common desktop PC[22].

The study by Lyu et al. (2018) examined the relationship between the number of seed files and the fuzzing performance, finding no clear correlation between the number of seed inputs used and the number of crashes detected. Their findings suggest that an initial seed set of 50 or more files is sufficient for fuzzing tools to trigger crashes and uncover new execution paths. Building on this research and my insights, I have opted for a smaller initial seed set, selecting 20 files from a large dataset for fuzzing.

During the development of the prototype, I continuously conducted fuzz testing campaigns against the Windows graphics component using EMF files, which uncovered several previously unknown security vulnerabilities that I subsequently reported confidentially to the vendor. Microsoft later addressed these vulnerabilities with the necessary security fixes, and I will discuss them in detail later in the dissertation.

3.2.2 Model justification

In this research, a NN is implemented to optimize the seed selection process in fuzz testing. I chose a NN because it offers a compelling balance between interpretability, computational efficiency, and robust generalization, even when training data is limited. By generating probability scores, the model helps researchers prioritize seed files that are more likely to expose software flaws.

Although alternative models, such as SVMs and ensemble methods like random forests and gradient boosting machines, can effectively capture more complex data patterns, they often require extensive fine-tuning and considerable computational resources. In contrast, simple linear models like logistic regression assume a linear relationship between features and outcomes, which may necessitate additional feature engineering in scenarios, such as fuzz testing, characterized by class imbalances and non-linear interactions. Therefore, the selected NN architecture, with its ability to model complex, nonlinear relationships through hidden layers and regularization techniques, is better suited for this context.

Furthermore, while ensemble methods may offer predictive performance improvements, they tend to compromise interpretability and are more susceptible to overfitting, especially when trained on small or imbalanced datasets. DNNs can require large datasets

[22] Fuzzing experiments were conducted on my personal desktop PC in my home lab. The complete hardware specifications are provided in later sections.

and may lack transparency, which poses challenges in security vulnerability research, where sample sizes are often limited. In this work, early stopping, dropout layers and regularization techniques, such as L1 and L2 penalties, are integrated to mitigate the risk of overfitting, ensuring that the NN remains effective even when data is scarce or noisy. However, the overall success of the model is heavily dependent on careful feature engineering, as inadequately designed features can impair its ability to accurately distinguish between valid and crashing sample files.

It is important to note that the main objective of this research is to optimize the fuzz testing process rather than to serve as a comprehensive exploration of ML methodologies. The binary classifier for seed selection is presented as an experimental approach within the broader scope of fuzzing, rather than a definitive solution supported by extensive empirical evidence. In some cases, hybrid approaches, such as the combination of linear models with heuristic-based selection or ensemble methods, may offer further enhancements. However, as implemented here, the NN, in conjunction with thoughtful feature engineering, strategies to address class imbalance, and proper feature scaling, serves as a viable and efficient method to optimize the seed selection process and improve the bug-finding capability of fuzzers.

I propose an alternative seed selection strategy that uses a discriminative ML model, employing binary classification to differentiate between sample files. This model identifies valuable seed input for fuzzers, as illustrated in Figure 14. This prediction-based seed selection strategy consists of the following four key steps:

- **Training data collection**: Establish criteria for evaluating input file utility and define a methodology for acquiring the training dataset.
- **Training data conversion**: Transform raw binary files into standardized matrix representations, ensuring compatibility across various file formats and sizes.
- **Model construction**: Develop a prototype discriminative ML model using binary classification, trained on the matrices generated in the previous step.
- **Model prediction**: Use the trained model to estimate the likelihood that a candidate seed file triggers a crash in the program under test.

The implemented `SeedClass` prototype aims to improve existing coverage-guided mutational fuzzers. I selected `WinAFL`, one of the most robust fuzzing frameworks for the Windows platform, to evaluate the effectiveness of the proposed prediction-based seed selection strategy and to assess the working prototype. `WinAFL` inherits its well-established methodologies and its proven ability to uncover security vulnerabilities from `AFL` while ensuring compatibility with Windows environments. The extensive track record of these fuzzers, supported by a comprehensive collection of identified vulnerabilities, demonstrates their reliability in security research. Additionally, `AFL` has contributed non-security improvements to various tools, further solidifying its impact beyond vulnerability detection.

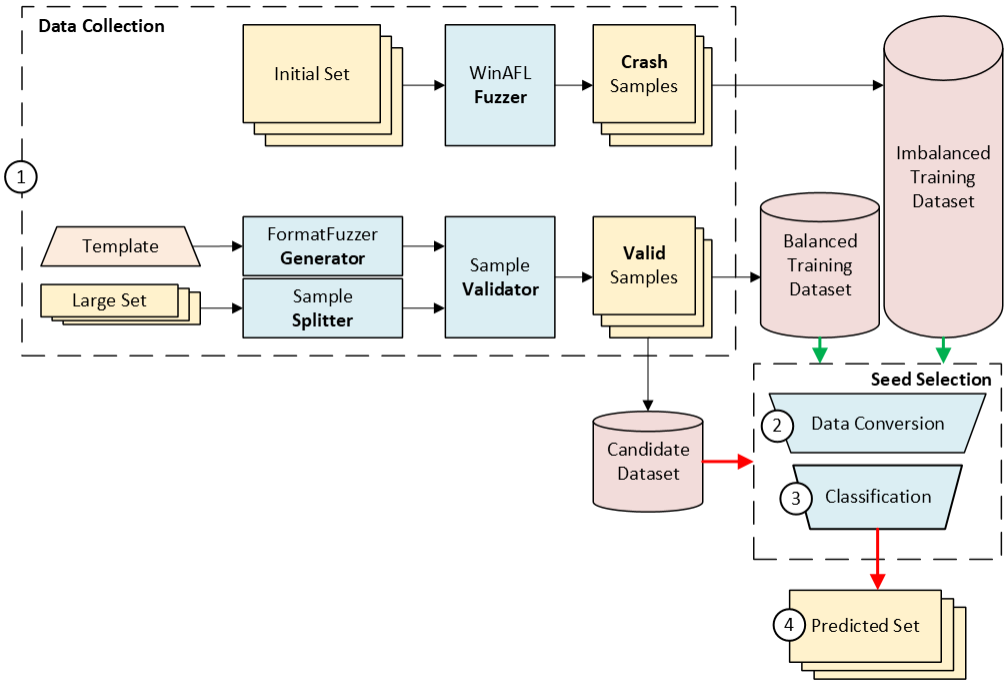


Figure 14: Overall architecture of the concept (author's own work)

3.2.3 Data collection

The data collection process must address two key questions: First, what criteria must a sample file meet to be deemed useful for the model? Second, how can we obtain high-quality samples in sufficient quantities for model training? The following sections will explore these topics in detail.

Establishing an initial training set is the most critical step in developing a ML model that can help a researcher in selecting optimal seed files. The effectiveness of `SeedClass` relies on the input files in this set that demonstrate genuine value. Therefore, it is essential to define the criteria for valuable input files. Without such criteria, `SeedClass` may fail to learn important features and, consequently, struggle to recognize them accurately.

The `SeedClass` prototype identifies valuable files as those that trigger unique program crashes or exercise distinct code paths. Since the primary goal of fuzzing is to detect crashes, input files that cause unique failures are particularly useful in this context. Furthermore, research indicates that increasing code coverage improves the likelihood of discovering new vulnerabilities. Existing seed selection algorithms, such as `AFL-cmin`, have been used to identify effective input files for the training set. However, research indicates that their effectiveness depends on the target program and the fuzzing strategy.

Two primary methods are available to acquire training data. The first method involves generating data from scratch, while the second method consists of mutating existing data, an approach that aligns with mutation-based fuzzers, such as `WinAFL`. Data generation requires a specialized generator program, such as `FormatFuzzer`, which produces syntactically valid data according to the specifications of the target file format.

Fuzzing tools such as `WinAFL` can store input files that trigger unique crashes or explore new code paths, making them effective for assembling training data. This process involves using valid input samples to start fuzzing the Windows graphics component, followed by extracting valuable input that reveals new execution paths or program crashes for the training set of `SeedClass`. The construction of this training set does not impose practical limitations on `SeedClass`. The evaluation of valuable input files is based on their capacity to trigger new code paths or program crashes, criteria that `WinAFL` inherently applies. The parallel execution of multiple fuzzer instances increases the speed of data collection, enabling a more rapid and efficient assembly of the training set.

Initial fuzz testing can be performed on outdated Windows builds that are known to be affected by specific vulnerabilities. Microsoft releases security updates for Windows on a monthly basis. The number of patches can vary significantly, but typically ranges from 50 to 100 updates, depending on the severity and scope of the vulnerabilities addressed. Although details on fixed vulnerabilities are limited, the sheer volume suggests a significant number of flaws within the Windows graphics component. I conducted fuzzing campaigns on Windows 10 build 1507 as part of my research. This build reached general availability in 2015, nearly a decade ago at the time of writing this dissertation. During this time, I identified more than 50 unique bugs in the Windows graphics component from thousands of crash samples, as detailed in the table provided in the Appendix 6.4.

This data collection method effectively evaluates the utility of input files. The training set comprises files that exercise previously unexplored code paths or trigger distinct crashes in the Windows graphics component, making them highly valuable for model training and development. Furthermore, corrupted files often lead to unique program failures and expose new execution paths; however, obtaining such files can be challenging. Fortunately, `WinAFL` reliably produces a substantial number of corrupted inputs that address this limitation.

Building on my initial strategy of using the `WinAFL` fuzzer to generate data, I incorporated additional methods to ensure a sufficient dataset for my ML experiments. I developed an enhanced binary template for the EMF file format, which enabled the `FormatFuzzer` generator to produce random files based on this template whenever a larger volume of samples was required.

Additionally, I implemented another strategy to split large metafiles, those exceeding 1K in size, which are unsuitable for effective fuzzing, into multiple smaller pieces, thereby generating dozens or even hundreds of manageable samples. Although the `WinAFL`-based strategy provided a solid foundation, these supplementary methods ensured a consistent, high-quality data supply. The following chapter provides a detailed discussion of the binary template, the `FormatFuzzer` generator, and the file splitting methodology.

3.2.4 Data conversion

The initial exploratory phase of the fuzzer operates without a NN, instead relying on an initial dataset collected from the Internet to obtain crash samples. The fuzzer identifies and separately stores unique testcases that trigger crashes in the application under test. A crash is considered unique if it involves previously unseen state transitions. However, multiple execution paths may lead to the same crash, generating distinct samples that trigger the same bug. Each crash sample is renamed according to its `SHA256` fingerprint to eliminate duplicates in different fuzzing campaigns.

The `SeedClass` tool must convert raw input files into a standardized matrix format to ensure compatibility with the NN. The preprocessing mechanism implemented in the prototype follows the conversion method described in Lyu et al. (2018). The transformation process consists of four steps:

1. Each input is read as a binary string and converted into a `Base64` character string.
2. Each character is mapped to a numerical value, yielding a sequence of integers derived from the raw file.
3. Every six consecutive integers are grouped into a larger decimal number and normalized.
4. The resulting matrix is padded with null bytes to ensure uniformity.

Initially, step 3 of the conversion method grouped every six integers by computing the product of the nonzero values and normalizing the result to the range [0, 0.75418890624]. However, experimental results indicated that an alternative approach, simply concatenating the integers to form the decimal feature, yielded improved predictive performance. Consequently, I incorporated this alternative conversion mechanism as an option in my model. Additionally, I train the NN to accommodate the largest input in accordance with its specific size requirements.

3.2.5 Model construction

The proposed ML model is a multi-layer perceptron (MLP), a type of feed-forward ANN comprising at least three layers: an input layer, a hidden layer, and an output layer. Each node, except those in the input layer, functions as a neuron utilizing a non-linear activation function. The MLP is trained through supervised learning using the backpropagation algorithm.

The NN was developed using a straightforward and pragmatic approach. The implemented prototype consists of a sequential (feed-forward) dense NN with a single fully connected hidden layer containing 256 neurons. This simple yet effective design serves as a straightforward starting point for constructing NNs, ensuring that the model remains both interpretable and manageable. As indicated in Table 6, which outlines common MLP

architectures for classification tasks, the architecture followed in this study adheres to a typical structure found in the literature.

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
hidden layers	Typically 1-5, depending on the task		
output neurons	1	1 per binary label	1 per class
output layer activation	Sigmoid	Sigmoid	Softmax
loss function	X-entropy	X-entropy	X-entropy

Table 6: Typical classification MLP architecture (Géron, 2022)

During the early phases of the project, the predictive model began as a simple logistic regression to establish baseline performance. After the initial experiments, the model transitioned to a basic NN incorporating a single hidden layer, which yielded encouraging results through the use of non-linear activation functions. However, as further research underscored the complexity of the task, the resulting architecture evolved into a more sophisticated design.

The final model is a six-layer NN. The sequential architecture of the model integrates multiple hidden layers with 256 neurons each, employing the `ReLU` activation function, and optional L2 regularization techniques, interleaved with `Dropout` layers to mitigate overfitting, and a single-neuron output layer with a sigmoid activation function to optimize the model for binary classification tasks.

- **`Input` layer:** Processes a fixed-size byte sequence of length 256.
- **`Dense` layers:** Two layers, each containing 256 neurons, using the `ReLU` activation function to mitigate the vanishing gradient problem and facilitate faster convergence. These layers may incorporate optional regularization techniques to enhance generalization.
- **`Droput` layers:** Two layers with a 50% dropout rate, which reduces overfitting by randomly deactivating half of the neurons during training (Panwar et al., 2025).
- **`Output` layer:** A single-neuron dense layer employing a `sigmoid` activation function, generating an output between 0 and 1. This value represents the likelihood that the input sample exhibits characteristics associated with program crashes based on patterns learned from the training data.

For binary classification problems, the logarithmic loss function, commonly known as `binary cross-entropy` (Versloot, 2021), is the preferred choice because it pairs effectively with the `sigmoid` activation function. This function evaluates the prediction accuracy when the `true` labels are binary (0 or 1), making it particularly useful for probability-based models. A lower binary cross-entropy value means improved model performance. Furthermore, the smooth gradient provided by this function enhances the learning efficiency. The `Adam` optimization algorithm, known for its effectiveness in gradient descent, is used to fine-tune the parameters of the NN. Training accuracy metrics are collected to

assess performance, and the model is periodically retrained and updated with new sample cases, either after accumulating a predefined number of new inputs or after completing a fuzzing campaign.

The evaluation was performed on a desktop machine equipped with an Intel Core i9-10900 processor and 32 GB of system memory, running a 64-bit version of Windows 10 Professional. This processor features 10 cores, each supporting two threads, which allows the simultaneous execution of 20 fuzzer instances to maximize fuzzing efficiency. No memory constraints were imposed on the fuzzer instances, and all experiments were performed on the same machine to ensure consistency and control over variables.

4. Experimental implementation

For this study, I have conducted vulnerability research on the Windows graphics component through fuzz testing the processing of EMF files. Evidence shows that the Windows Graphics Device Interface (GDI) has been a popular target for attackers in recent years, with the processing of various metafile formats as a common attack vector. I have developed a simple utility application to evaluate the process of opening and displaying EMF metafiles using the `WinAFL` fuzzer. Having many testcases is essential for experimenting with ML. Hence, I have created a binary template (SweetScape, 2021) for the `010 Editor` to generate example files with the `FormatFuzzer` tool. The following sections summarize the most important information on the technical basis of the research. The complete source code of the `SeedClass` prototype can be found in the Appendix 6.4.

4.1 Generating seed inputs

The following sections provide an overview of the problem of generating seed inputs for training the ML model, as well as for fuzzing the target system. Initially, the discussion covers the selection of the target component and the corresponding file format used for its security evaluation. Subsequently, I briefly describe the approach for generating example data tailored to this task.

4.1.1 Windows graphics component

GDI and its successor, Graphics Device Interface Plus (GDI+), are integral graphical subsystems of the Windows operating system (Microsoft, 2023). Originally developed for 16-bit Windows, GDI has undergone substantial updates to support 32-bit and 64-bit systems. It serves as a hardware abstraction layer for the Win32 platform and is implemented in C, although it lacks the object-oriented structure of GDI+. Polyakov and Brusentsev (2005) provides a comprehensive discussion of GDI+ capabilities.

Introduced in Windows XP as a replacement for GDI, GDI+ offers developers access to a collection of C++ classes that encapsulate flat C functions. Unlike GDI, GDI+ employs an object-oriented approach to simplify and enhance graphical operations. Designed for .NET, it seamlessly integrates into the `System.Drawing` namespace, providing developers with a more intuitive and efficient interface.

Among its advantages, GDI+ supports modern graphics file formats such as `JPEG` and `PNG`. Although generally slower than GDI, its ease of use, reduced likelihood of misuse, and ability to generate higher-quality images make it suitable for most applications.

GDI has historically been associated with security vulnerabilities due to its complexity. Several high-profile vulnerabilities have been identified in the past, including CVE-2008-2249, CVE-2016-3393, and CVE-2021-1665. CVE-2008-2249 enabled attackers to execute arbitrary code via a specially crafted Windows Metafile (WMF) image file. CVE-2016-3393 (Ivanov, 2016) allowed remote attackers to escape the browser sandbox through a specially crafted website. CVE-2021-1665 (Hardik, 2021) allowed remote execution of arbitrary code through a thumbnail preview of an EMF file containing a specially crafted `EmfPlusDrawString` metafile record.

4.1.2 Enhanced Metafile format

EMF is a vector-based graphics file format introduced by Microsoft in 1993 as the successor to WMF (Microsoft, 2022). Unlike its predecessor, EMF is 32-bit and supports 24-bit color depth, making it more suitable for high-quality graphics.

Designed for scalability, EMF stores vector graphics as mathematical equations that define shapes and lines, allowing images to be resized without loss of quality. Unlike raster-based graphics, which suffer from pixelation when enlarged, this approach enables seamless scaling. Although originally developed for Windows 2.0, the EMF files are intended for portability between applications and devices. They can contain both vector and raster elements.

EMF supports various graphical primitives, including lines, curves, polygons, and text, along with transparency features. It is compatible with editing tools such as Adobe Illustrator and CorelDRAW. Additionally, the relatively compact file size makes EMF files efficient for storage, transfer, and also ideal for fuzzing. However, the EMF format is not optimal for storing complex raster-based images, such as photographs, and lacks support for animation or interactivity, restricting its use in web applications. Compatibility issues may also arise, as not all software can open or modify EMF files. Despite these constraints, EMF remains widely used for vector-based graphics, as its compact size and scalability make it advantageous and can be easily converted to formats like Scalable Vector Graphics (SVG) and PDF, enhancing its versatility.

Security concerns surrounding the WMF format have drawn attention, as emphasized by Leonard (2016). In particular, the Metafile Image Code Execution (MICE) vulnerability, discovered in 2006 and awarded a 2007 Pwnie Award[23], arose from the ability of WMF files to manipulate the `SETABORTPROC` value within the `MetafileEscapes` enumeration via the `META_ESCAPE` record. This flaw allowed the execution of arbitrary code. Furthermore, the `META_ESCAPE_ENHANCED_METAFILE` record permits the embedding of EMF files within WMF files, making the security concerns of the EMF format relevant to WMF as well.

[23] The Pwnie Awards serve as an annual awards event that recognizes both outstanding achievements and notable shortcomings in information security.

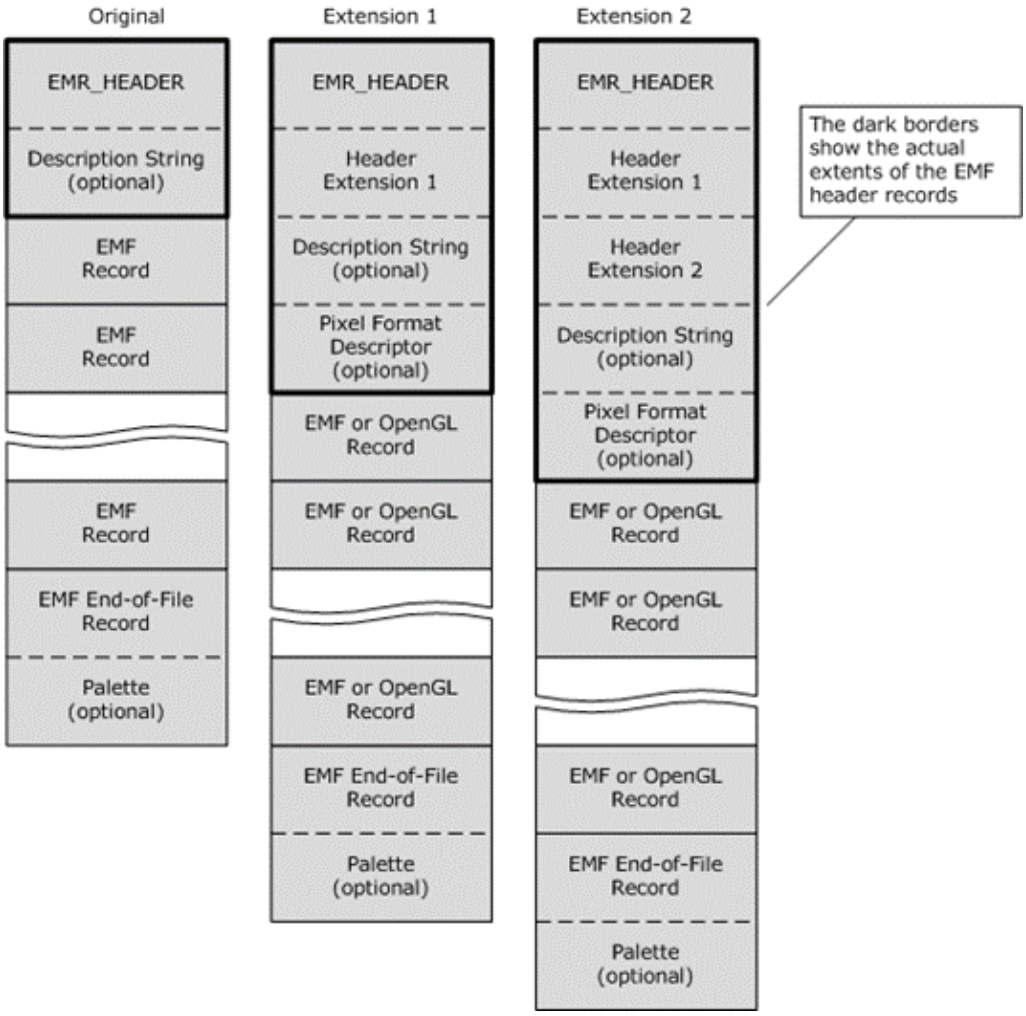


Figure 15: EMF metafile versions (Microsoft, 2022)

While the MICE vulnerability did not directly affect EMF, subsequent security flaws have been identified. Leonard (2016) noted that since EMF was designed to mimic GDI calls, vulnerabilities in GDI or its underlying display and printer drivers could be exploited via malicious EMF files. Custom implementations of EMF pose distinct risks, as metafiles contain numerous records that may not undergo the same level of security testing. Therefore, fuzzing such implementations could also uncover zero-day vulnerabilities. Additionally, an Enhanced Metafile Plus (EMF+)-aware application designed for GDI+ might discard unsafe data, while another implementation could remain vulnerable.

4.1.3 `FormatFuzzer` generator

`FormatFuzzer`, introduced by Dutra et al. (2023), is a generator designed to enhance the effectiveness of fuzzers that process structured binary inputs. While most existing tools are format-agnostic, meaning they lack inherent knowledge of specific file formats, this generality allows versatility but limits efficacy when valid format-specific inputs are required. `FormatFuzzer` addresses this challenge by leveraging pre-existing binary templates to

create or modify valid inputs, even in black-box environments where the target program provides no information. By combining the adaptability of format-agnostic fuzzers with the precision of format-specific fuzzers, it improves code coverage and increases the likelihood of uncovering software flaws or security vulnerabilities.

`FormatFuzzer` offers two key contributions. The first is its role as an efficient format-specific fuzzer, capable of generating outputs that conform to a given format, a capability that is particularly valuable in the context of the present research project. Second, it functions as a format-specific parser, capable of processing existing inputs. The parser can be seamlessly integrated into parser-augmented fuzzers, such as `AFLSmart` (Pham et al., 2021), enabling the evolution of format-specific fuzzers through seed input adaptation. Iteratively applying this approach across multiple templates results in a suite of coverage-guided, format-specific fuzzers.

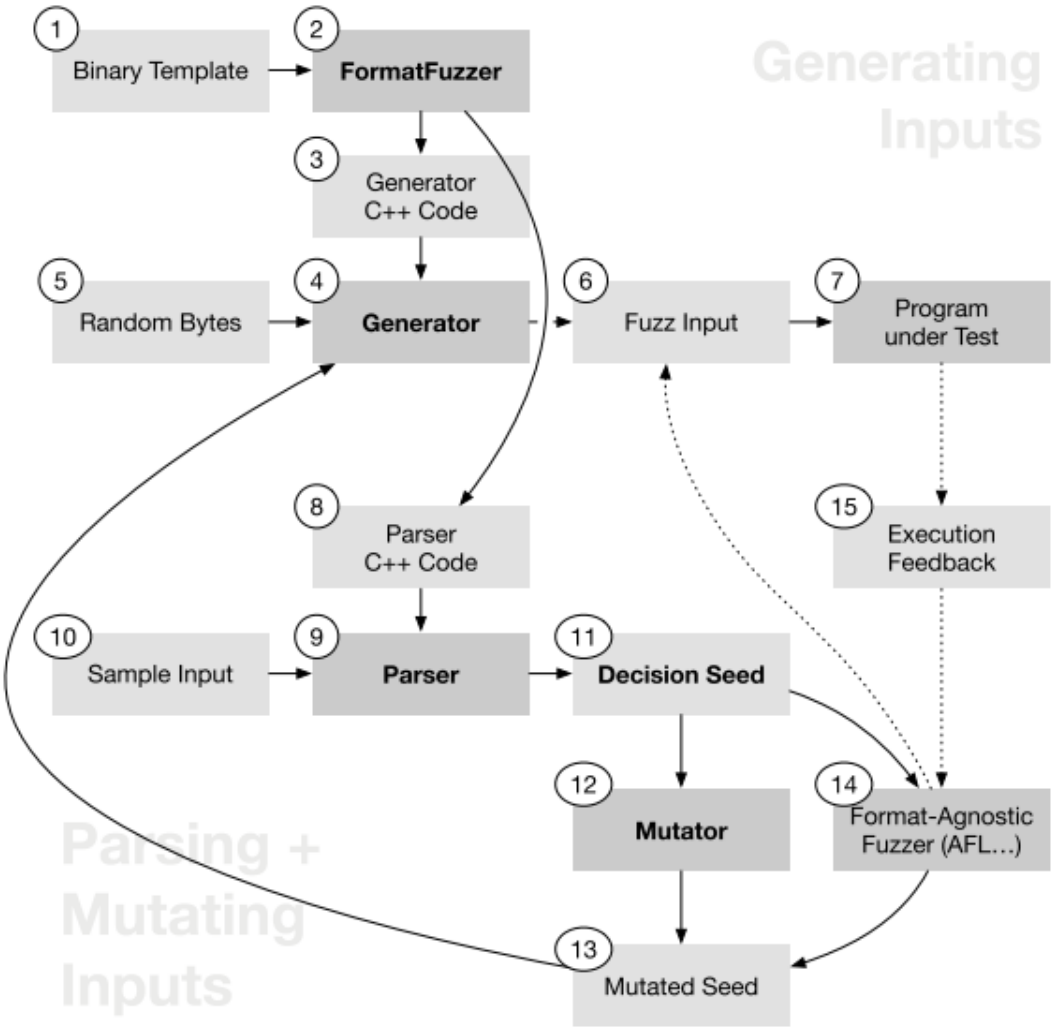


Figure 16: Overview of `FormatFuzzer` (Dutra et al., 2023)

Unlike conventional format-specific fuzzers, which require substantial effort to implement guidance and analysis features, `FormatFuzzer` streamlines the process. It converts an existing input specification into `C++` code, enabling the parsing, modification, and rapid

generation of inputs that comply with format rules. In addition, it can be integrated with format-independent fuzzers, such as AFL, thereby enhancing their ability to recognize and process structured inputs. Figure 16 provides an overview of FormatFuzzer.

For complex formats such as MP4 and ZIP, FormatFuzzer has demonstrated its effectiveness by generating valid inputs that uncovered previously unknown security vulnerabilities in the FFmpeg and timidity libraries. Furthermore, it improves code coverage by reaching code paths inaccessible to format-agnostic fuzzers, thereby increasing the likelihood of detecting bugs and vulnerabilities.

Consequently, FormatFuzzer presents a promising approach to enhancing the effectiveness of fuzzing programs for structured binary inputs. Its ability to generate substantial, format-compliant samples is crucial for the ML aspect of this project, providing the extensive data necessary for training the model, and it also complements the mutational fuzzing strategy of WinAFL by further enriching the overall suite of available testcases.

4.2 Fuzzing the Windows graphics component

The following sections detail the approach I adopted for fuzz testing the Windows graphics component. This methodology incorporates several key elements: instrumenting the target using DynamoRIO, employing the coverage-guided mutational fuzzer WinAFL for testing, and implementing the WinAFL Pet fuzz manager to remotely monitor test campaigns. Most importantly, a dedicated fuzzing harness was developed to establish a robust connection between the fuzzer and the target system, thereby ensuring seamless integration and effective execution of the testing process.

4.2.1 Dynamic instrumentation

As Bruening et al. (2003) and Bruening (2004) summarize, DynamoRIO was developed as the IA-32 version of Dynamo (Bala et al., 2000) and is available on both IA-32 Windows and Linux platforms. It efficiently handles large desktop applications by monitoring and modifying the application instructions prior to execution. Typically, this is achieved through an interpretation engine; however, emulation-based interpretation is computationally expensive, particularly on a complex instruction set such as IA-32. DynamoRIO mitigates this overhead by caching translations of frequently executed code, allowing direct access and avoiding performance bottlenecks (Garnett, 2003). Figure 17 illustrates the general architecture of DynamoRIO.

DynamoRIO copies basic blocks[24] into a code cache, where they are executed natively, although the extra code involved with this caching mechanism adds major overhead (Re-

[24] A basic block is a sequence of instructions ending with a control transfer operation. These blocks provide a linear view of control flow, characterized by a single entry point and one or more exits. (Chow et al., 2001; GCC Team, 2021)

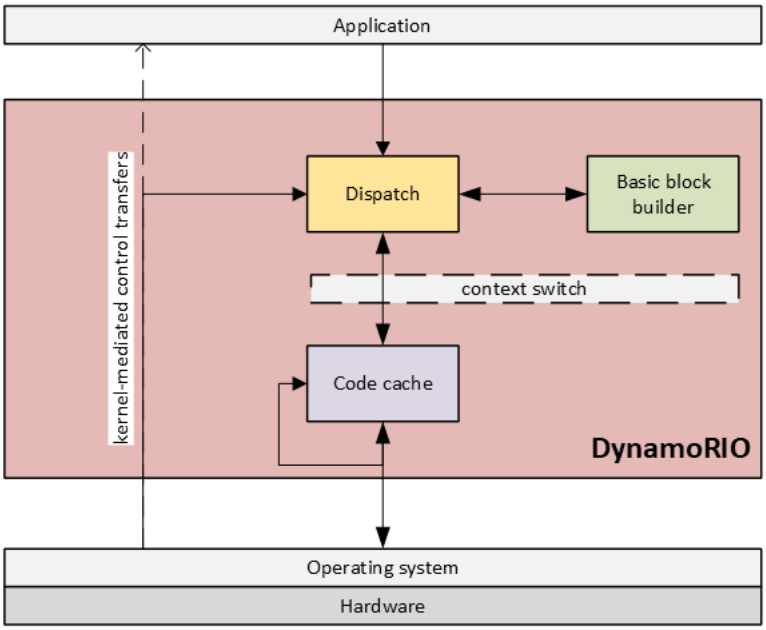


Figure 17: Architecture of DynamoRIO (based on Bruening, 2004)

ichelt et al., 2024; Zhao & Wong, 2005). After executing a block, the application's machine state is saved, and the context is switched to process the next basic block. When a target basic block already resides in the code cache and is accessed via a direct branch, DynamoRIO links the blocks using a direct jump, thereby avoiding costly context switching (Kiriansky et al., 2003).

DynamoRIO provides a robust API for creating client applications that collaborate with DynamoRIO to analyze and manipulate program execution. Clients utilize the API to implement custom modifications through hook functions invoked by DynamoRIO. This client interface abstracts low-level details, enabling focused transformations of application code stored in the code cache (Bruening et al., 2022).

I chose dynamic instrumentation with DynamoRIO for this study due to several constraints associated with alternative instrumentation methods supported by WinAFL. For instance, although hardware tracing with Intel PT [25] is available (Bakhvalov, 2019; Reinders, 2013), it requires specific hardware support and may preclude the use of virtual environments, which is an important consideration for this project. Additionally, while static instrumentation through syzygy (Google, 2018) is an option, it depends on full private symbols, but Microsoft only provides public symbols for Windows binaries, limiting its applicability. Finally, dynamic instrumentation using TinyInst was not yet available at the time of writing. These limitations collectively motivated the decision to employ DynamoRIO as a flexible and compatible solution for the fuzz testing approach.

[25] Intel PT is a low-overhead hardware feature that allows the recording and the reconstruction of the execution flow (Reinders, 2013).

4.2.2 Coverage-guided mutational fuzzing

This section explains the application of coverage-guided mutational fuzzing in this research. Initially, I considered employing the AFL fuzzer due to its well-documented success in driving code coverage. However, AFL is primarily designed for Linux-based systems. Given that the target platform for this study is Windows, I opted for WinAFL, a fork of AFL that is specifically developed to operate within Windows environments, thus ensuring compatibility and optimal performance during testing.

AFL fuzzer: The initial approach

AFL is a widely recognized coverage-guided gray-box fuzzing tool developed by Zalewski (2017). It operates by mutating an initial input set to activate new internal states in the target program. When an input uncovers a novel execution path, AFL stores it as an interesting input and continues mutating it to further explore the program. To track code coverage, AFL instruments the branch transitions of the target program, encoding the coverage information in a bitmap. It supports static instrumentation via the compiler and dynamic instrumentation through user-mode emulation using QEMU. When source code is proprietary, dynamic instrumentation remains the only viable option.

Since AFL repeatedly executes the target program during fuzzing, it optimizes performance by running the program until initialization completes and then repeatedly creating child processes using the Linux fork mechanism. As described in Zheng et al. (2019), this approach bypasses the overhead of setting up a new process for each test case. Figure 18 illustrates the general workflow of AFL and the critical functions invoked at each step.

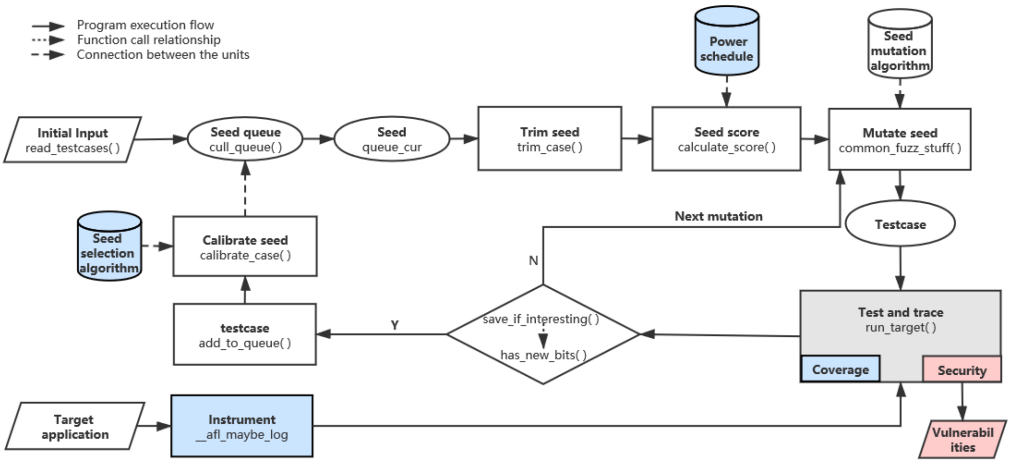


Figure 18: General workflow of AFL (Pang et al., 2021)

AFL generates diverse inputs via mutation, aiming to trigger bugs rather than adhering to a predefined strategy. As described by Zalewski (2014), it is essentially a collection of practical techniques implemented in a straightforward and robust manner.

Despite its success in uncovering numerous security vulnerabilities, AFL's reliance on code coverage alone may be insufficient. It does not provide initial seed inputs, and its blind mutation strategy lacks awareness of how to optimize seed files to discover vulnerabilities in code regions that are hard to reach.

WinAFL fuzzer: The Window-specific adaptation of AFL

WinAFL is an adaptation of the widely recognized AFL fuzzer, which uses compile-time instrumentation and genetic algorithms for fuzz testing. Developed by Fratric (2023) to extend AFL's functionality to Windows, WinAFL utilizes dynamic instrumentation via DynamoRIO and TinyInst, hardware tracing through Intel PT, and static instrumentation with Syzygy, opting for runtime analysis instead of compile-time instrumentation, facilitating fuzzing in the predominantly closed-source Windows ecosystem. This research specifically utilizes DynamoRIO, and the following sections will focus on its application.

While this approach introduces execution overhead, approximately twice the native execution speed, it remains comparable to the binary instrumentation mode of AFL. As an open-source fuzzing tool, WinAFL is highly effective at detecting software bugs and vulnerabilities in Windows applications. It follows coverage-guided fuzzing principles, leveraging code coverage feedback to refine input generation and mutation strategies.

A fundamental difference between Linux and Windows fuzzing environments is the absence of efficient process duplication on Windows systems, which is crucial for effective fuzzing. To address this limitation, Windows-based fuzzers such as WinAFL employ in-app persistent-mode fuzzing, mitigating the inefficiencies associated with process cloning.

As Y. Zhang and Du (2023) describes, WinAFL obtains coverage data through dynamic binary instrumentation using DynamoRIO. When the fuzzer identifies a new execution path, it logs the coverage information and places the corresponding input sample in the queue for subsequent mutations to expand coverage. WinAFL interacts with the target process via a named pipe and shared memory mapping; the pipe facilitates command exchange, while the mapped memory stores coverage data.

The instrumentation code wraps the target function, invoking `pre_fuzz_handler()` before and `post_fuzz_handler()` after each execution. When the target function is reached, `pre_fuzz_handler()` preserves contextual information, such as register states and runtime parameters. After execution, `post_fuzz_handler()` records the response context, and this process repeats iteratively for a predefined number of cycles.

I extended WinAFL with a feature called `bug bucket`, grouping crash samples based on the memory address of the exception, for faster and easier validation of the discovered program crashes. The modifications made within `afl-fuzz.c` are summarized in Listing 1. By default, the instrumentation client transmits only the occurrence and code of an exception to the fuzzer via the named pipe. My modification adds memory address transmission, processed within the `run_target()` function (line 22), allowing the fuzzer

to store the address for each detected crash. When a crash is associated with a previously observed memory address (line 5), it is saved separately in a `bucket` directory (line 42) within the `save_if_interesting()` function.

During my experiments with `WinAFL`, the `bug bucket` feature significantly reduced redundant crash reports, by a factor of ten, streamlining management within the `WinAFL Pet` fuzz manager, discussed in the next section. While all output generated by the fuzzer was preserved, redundant crash samples were collected separately. Although these crashes do not represent unique vulnerabilities, they remain valuable inputs for ML models, as they illustrate multiple pathways that can trigger the same program crashes.

```
1  PVOID ret_exception_addr = 0;
2  PVOID bug_bucket[BUG_BUCKET_SIZE] = { 0 };
3  int   bug_bucket_count = 0;
4
5  int in_bug_bucket(PVOID exception_addr) {
6    int i;
7    for (i = 0; i < BUG_BUCKET_SIZE; i++) {
8      if (bug_bucket[i] == exception_addr) return 1;
9    }
10
11   if (bug_bucket_count == BUG_BUCKET_SIZE) bug_bucket_count = 0;
12
13   bug_bucket[bug_bucket_count] = exception_addr;
14   bug_bucket_count++;
15
16   return 0;
17 }
18
19 static u8 run_target(char** argv, u32 timeout) {
20   if (ReadCommandFromPipe(timeout) == 'C') {
21     ret_exception_code = ReadDWORDFromPipe(timeout);
22     ret_exception_addr = ReadPVOIDFromPipe(timeout);
23     destroy_target_process(2000);
24     return FAULT_CRASH;
25   }
26 }
27
28 static u8 save_if_interesting(char** argv, void* mem, u32 len, u8 fault) {
29   s32 fd;
30   u8 keeping = 0;
31   u8 fn[MAX_PATH];
32   u8 exception_name[MAX_PATH];
33
34   switch (fault) {
35     case FAULT_CRASH:
36       snprintf(fn, MAX_PATH, "%s\\crashes\\id_%06llu_%02u_%s",
37         out_dir, unique_crashes, kill_signal, exception_name);
38
39       if (use_bug_bucket) {
40         snprintf(fn, MAX_PATH, "%s\\crashes\\id_%06llu_%p_%s",
41           out_dir, unique_crashes, ret_exception_addr, exception_name);
42         if (in_bug_bucket(ret_exception_addr)) {
43           snprintf(fn, MAX_PATH, "%s\\crashes\\bucket\\id_%06llu_%p_%s",
44             out_dir, unique_crashes, ret_exception_addr, exception_name);
```

```
45          }
46        }
47        break;
48    }
49
50    return keeping;
51  }
```

Listing 1: Storing duplicate crash samples separately

4.2.3 Remote fuzzing management

As part of this research, I developed `WinAFL Pet`, a custom tool designed to manage `WinAFL` fuzzer instances and simplify the remote administration through a user-friendly web interface. This tool facilitates the effortless monitoring of multiple test campaigns running on various remote machines. Given the specific scope of this research, an in-depth discussion of `WinAFL Pet`'s implementation is beyond the present study and would warrant a dedicated chapter or independent article. Here, I summarize only the key functionalities. However, its development was essential for enhancing the efficiency of conducting experiments.

`WinAFL Pet` follows a client-server model, with components communicating via a basic REST-like API. The lightweight agent, written in `Go`, runs efficiently as a system service on fuzzing machines. The server, built using the `Gin` web framework and deployed as a Docker container on Alpine Linux, serves as the control center. It can be easily deployed on standard network-attached storage (NAS) devices or a simple Raspberry PI, while agents operate on virtual machines. Figure 19 illustrates this typical deployment scenario.

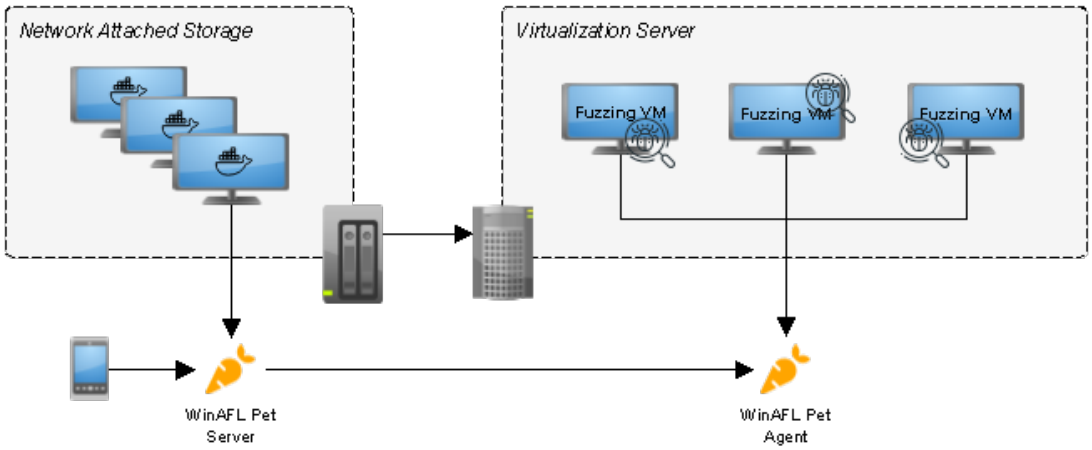


Figure 19: WinAFL Pet deployment schema (author's own work)

To start using `WinAFL Pet`, users first register an agent on the *Agents* page using a secret key manually generated during deployment. This step establishes communication

between the server and the agent to send instructions. Once registered, users navigate to the *Jobs* page to create a new fuzzing *job* linked to the previously registered *agent*. With a few exceptions, the tool supports most command-line options and parameters of the `WinAFL` fuzzer, simplifying the configuration of fuzzing jobs. Upon initiating a fuzzing job, the server signals the agent to start fuzzing the target application by executing `afl-fuzz.exe` on the target machine with the specified parameters. Depending on the complexity and size of the target application, the initialization process may take some time.

Users can monitor the progress of the fuzzing session via the *Jobs* page, where they can access detailed statistics, track active fuzzing instances and retrieve data on crashes found by the fuzzer. The tool may also periodically request crash reports from the agent and can send notification emails. In addition, users can download the input files that trigger crashes for further analysis to determine whether they represent genuine bugs. `WinAFL Pet` significantly reduces administrative overhead, facilitating efficient fuzzing job management across multiple machines. It allows researchers to monitor fuzzing sessions, view statistics, inspect running instances and collect crash data. However, it is limited to working exclusively with `WinAFL`.

4.2.4 Dedicated testing harness

A *fuzzing harness* is a program designed to facilitate and execute fuzzing tests. Its primary role is to inject data, generated or mutated by the fuzzer, into the target system during the fuzzing process. In addition, a fuzzing harness can provide extra functionality to enhance efficiency and effectiveness. For example, in the case of testing a file parser, the harness retrieves input data and forwards it to the parser for processing. The complete source code of the fuzzing harness developed for this research can be found in the Appendix 6.4.

The code snippet in Listing 2 summarizes the main functionality of the testing harness used to fuzz the Windows graphics component with the EMF image file format. The harness is a `Win32` application that implements the `WinMain()` function rather than the conventional approach of developing a console application that implements the `main()` function. Since Windows 10, console applications spawn a child process called `conhost.exe`, introducing performance overhead that slows down the fuzzing process. As noted by Lotfi (2020), developing the harness as a `Win32` application increases the speed of fuzzing.

The `Fuzz` function is a critical component that performs various operations on the metafile while also evaluating command line parameters, such as setting up shared memory. During testing, the fuzzer instruments this function and monitors the execution of each instruction that invokes the target system. The `dllexport` keyword indicates the `Fuzz` function as exported, allowing `WinAFL` to locate the function by name rather than offset address. Additionally, the `noinline` keyword is a function attribute instructing the compiler not to inline the `Fuzz` function, thus preserving the exact function boundaries, which is particularly important in fuzzing.

The program first initializes an `Image` object (line 4), followed by a `Metafile` object (line 5) from either a file on disk or data in shared memory. Shared memory improves fuzzing performance, whereas file-based input management is crucial for post-fuzzing crash analysis.

```
1  extern "C" __declspec(noinline dllexport) int Fuzz(HDC hDC) {
2    memcpy(sample_bytes, shm_data + sizeof(uint32_t), sample_size);
3    IStream* stream = SHCreateMemStream(sample_bytes, sample_size);
4    image = Gdiplus::Image::FromStream(stream);
5    metafile = new Gdiplus::Metafile(stream, hDC);
6
7    thumbnail = image->GetThumbnailImage(100, 100, NULL, NULL);
8
9    hEmf = metafile->GetHENHMETAFILE();
10   for (size_t x = 1; x <= 8; x++) {
11     for (size_t y = 0; y <= 4; y++) {
12       Gdiplus::Metafile::EmfToWmfBits(hEmf, 0, NULL, x, y);
13     }
14   }
15
16   graphics = Graphics::FromImage(image);
17   graphics->DrawImage(image, 0, 0);
18
19   return 0;
20 }
```

Listing 2: Testing harness for fuzzing metafiles

The harness performs several operations on the image data, including generating thumbnails (line 7), converting EMF files to WMF format with various mapping modes (line 12) and creating a `Graphics` object to render the image (line 17). A critical final step is the cleanup process, which prevents memory leaks by deleting created objects and freeing allocated memory. Additionally, the program includes some error-handling mechanisms, which are not discussed here for brevity.

4.3 Selecting seed inputs with classification

This chapter presents a streamlined version of the toolset developed specifically for this research. For those interested in a more exhaustive implementation, the project's GitHub repository contains detailed, updated versions of the ML model and additional helper scripts beyond what is described here. The approach involves collecting the dataset, followed by loading and preprocessing the data, and visualizing the training examples. Then, the classification model is constructed, trained, and evaluated, thereby facilitating the prediction of valuable seed inputs.

4.3.1 Collecting the dataset

The first step in implementing `SeedClass` was assembling the training dataset. Considering training effectiveness, large sample files require longer feature vectors, leading to longer training times and slower fuzzing performance. However, even small metafiles can contain a sufficient number of records for effective ML. Thus, determining an appropriate file size for training and fuzzing is essential.

Balancing fuzzing efficiency and training effectiveness, the prototype employs files smaller than 1KB to construct the discriminative model of `SeedClass`. 256-length feature vectors can adequately represent files up to this size. However, overly sparse vectors, those with numerous zero elements, carry limited meaningful information. Consequently, the prototype processes input files smaller than 1152 bytes, aligning with `AFL` guidelines on seed file sizes while ensuring a sufficient number of diverse seeds in the preprocessed corpora. This setting optimizes efficiency, accelerates training, facilitates data collection and maintains adequate information density.

The raw data conversion mechanism proposed by Lyu et al. (2019) provides extensibility by allowing adjustments to input file size based on Base64 character-to-decimal conversions or varying feature vector lengths. This flexibility provides greater control over the input data.

To construct the dataset, I collected 421 sample files from publicly accessible repositories on platforms such as GitHub (Cosentino et al., 2016; GitHub, 2008) and GitLab (GitLab, 2011; Ríos et al., 2019). Of these, 390 were too large for effective fuzz testing. To address this limitation, I developed a helper script utilizing the `pyemf3` library (Sanders & McMullen, 2022) to split large metafiles into smaller segments, each containing 10 metafile records. This process successfully partitioned 119 files into 24,264 smaller samples, collectively occupying 22MB of storage.

To obtain *crash* samples, I conducted an initial test campaign on Windows GDI using a single physical machine equipped with an Intel i9 10900 CPU, 32GB of memory and Windows 10 version 1809 (Build 17763.107). I used `WinAFL` to fuzz an initial set of EMF metafiles in 20 parallel threads, resulting in 4,630 *crash* samples ranging from 136 bytes to 34 kilobytes.

The final dataset consists of 2,606 valid metafiles and 1,702 samples that trigger program crashes. To improve the integrity of the dataset and reduce redundancy, both *crash* and *valid* samples underwent the following data preprocessing steps:

- *Valid* samples were filtered using the `SeedFilter.py` helper script, detailed in Appendix 6.4. The files were grouped by size, and samples exhibiting sufficient byte-level variance were selected to ensure representative diversity.
- *Crash* samples were sorted using the `CrashSort.py` helper script, detailed in Appendix 6.4, to mitigate the overrepresentation of specific bugs, producing a more balanced dataset.

Supervised discriminative models train on labeled datasets, learning from known classifications to make predictions on unseen data. The ability to make predictions or decisions without being explicitly programmed to perform a task is made possible through this learning mechanism. In the context of fuzz testing, the model should train on metafiles labeled as *crash*, *valid*, *hanging*, or *interesting*. The prototype of `SeedClass` learns the characteristics of *crash* and *valid* metafiles to predict whether a newly encountered candidate is likely to cause a program crash.

Correctly labeled data is essential for a functioning supervised ML model. Mislabeling can lead to inaccurate predictions or classifications. To verify the validity of the training set, sample files marked as *valid* were confirmed to be correct metafiles by opening them in Microsoft Paint, a simple raster graphics editor available in all versions of Microsoft Windows.

This verification process was implemented using a code snippet shown in Listing 3, which takes advantage of the `pyWinAuto` module (Mc Mahon & Herrmann, 2019) to simulate user interactions with dialogs and controls through mouse and keyboard inputs. The addition of this verification step effectively filtered out metafiles from the training set that Microsoft Paint was unable to open, improving the integrity of the dataset.

To efficiently validate thousands of files concurrently across multiple parallel threads, I created the `SeedValidate.py` helper script, which is detailed in Appendix 6.4. The `is_emf_valid_via_ui` function sets a timeout for window detection, launches Paint with the given file, and checks for an error dialog indicating that Paint cannot read the file. If such a dialog appears, the metafile is invalid.

```
from pywinauto import timings
from pywinauto.application import Application
from pywinauto.findwindows import ElementNotFoundError

def is_emf_valid_via_ui(path):
  timings.Timings.window_find_timeout = 10
  app = None
  try:
    app = Application(backend='uia').start(f'mspaint "{path}"', wait_for_idle=False)
    dlg = app.window(title='Paint cannot read this file.', enabled_only=True)
    if dlg.exists() or dlg.is_visible():
      return False  # File is explicitly invalid
  except ElementNotFoundError:
    return True
  except Exception as e:
    return False
  finally:
    if app is not None: app.kill()
  return False
```

Listing 3: Verification script for *valid* samples

I implemented the prototype of `SeedClass` using the training data collected. The core functionality of the tool runs on version 3.11.9 of Python (Van Rossum, 2024), while the

discriminative model utilizes version 2.19.0 of TensorFlow (Abadi et al., 2025) with some helper functions from version 1.6.1 of Scikit-Learn (Pedregosa et al., 2011). The model uses `Adam` as the optimization algorithm (Z. Zhang, 2018), with hyperparameters, such as the number of training cycles, determined through extensive experimentation. Training the discriminative model using k-fold cross-validation on the same machine requires only a few seconds. The following sections summarize the key components of the seed selection tool, which uses a NN to predict whether an input file exhibits characteristics likely to trigger a crash during fuzz testing.

4.3.2 Loading and preprocessing the dataset

The `load_data()` function, shown in Listing 4, initializes an empty dictionary named `data` and processes the labeled input files from user-specified directories that contain `valid` metafiles and `crash` files that cause program failures. Internally, the `process()` function reads the binary contents of each file, encodes them using Base64, and maps each byte to predefined characters in `CHARMAP`. The encoded values are then segmented into chunks of six elements and transformed into numerical features based on either the product of nonzero values or simple concatenated string representations, depending on the `ENCODING` setting. The resulting numerical sequence is padded to a standardized length of 256 elements.

Each processed file is assigned a label as `LABEL_NEGATIVE` (0) or `LABEL_POSITIVE` (1), inserted as the first element of its feature representation. Files exceeding the predefined feature size constraint are discarded. The extracted features are stored in the `data` dictionary, where filenames serve as dictionary keys and the calculated feature vectors as values.

The `data` dictionary is then converted into a `DataFrame`, with filenames as the index. The names of the feature columns are prefixed with `f`, and the column `f0` is removed before storing the remaining data in `X`. The `.values` attribute converts the `DataFrame` into a `numpy` array, while the `f0` column values are stored separately in `y`, representing the target variable for the binary classification model.

Finally, the data is split into feature inputs (`X`) and labels (`y`), then further partitioned into training (`X_train`, `y_train`) and testing (`X_test`, `y_test`) subsets with 20% of the data reserved for testing, using the `train_test_split` function from Scikit-Learn.

```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3
4 def load_data(data, dir, label):
5     def process(path):
6         # Yield successive n-sized chunks from a list.
7         def chunks(lst, n):
8             return [lst[i:i + n] for i in range(0, len(lst), n)]
9
```

```
        with open(path, 'rb') as file:
            data = file.read()
        # Encode file content via Base64 and map each byte using CHARMAP.
        encoded = [CHARMAP.get(chr(i), 0) for i in b64encode(data)]

        # Group every 6 numbers into a single decimal feature.
        decimal = [
            np.prod(np.array(chunk)[np.array(chunk) != 0]) if ENCODING == 'product'
            else int(''.join(map(str, chunk)))
            for chunk in chunks(encoded, 6)
        ]

        # Pad the feature list to 256 elements.
        while len(decimal) < 256:
            decimal.append(0)

        return decimal

    for file in [f for f in os.listdir(os.path.join(os.getcwd(), dir)) if f.lower().
    endswith('.emf')]:
        features = process(os.path.join(os.getcwd(), dir, file))
        features.insert(0, label)
        if len(features) == 257: # Ignore too big files
            data.update({file: features})

    return data

dd = {}
dd = load_data(dd, args.negative, LABEL_NEGATIVE)
dd = load_data(dd, args.positive, LABEL_POSITIVE)

df = pd.DataFrame.from_dict(dd, orient='index').add_prefix('f')

X = df.drop('f0', axis=1).values
y = df['f0'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Listing 4: Data loading and preprocessing

Since my dissertation proposal, I have modified the `load_data()` function to increase the overall performance of data processing. The updated version now supports in-memory processing of the dataset compressed in a `ZIP` archive, thereby eliminating the time-consuming file operations associated with the traditional approach. This enhancement significantly speeds up the data loading process, contributing to a more efficient and responsive analysis workflow. The detailed implementation has been omitted here because it does not affect the core functionality of the tool. The most recent version, including all implementation details, is available in the code repository of the project.

4.3.3 Visualizing the training data

Visualization plays a crucial role in identifying patterns that can inform the development of effective classifiers. Dimensionality reduction techniques such as PCA and t-SNE help uncover the structure of the dataset and the separability of different classes. PCA, a widely used technique, simplifies large datasets by transforming them into lower-dimensional spaces. In this binary classification problem, where the goal is to distinguish between *valid* and *crash* inputs, PCA offers a useful perspective.

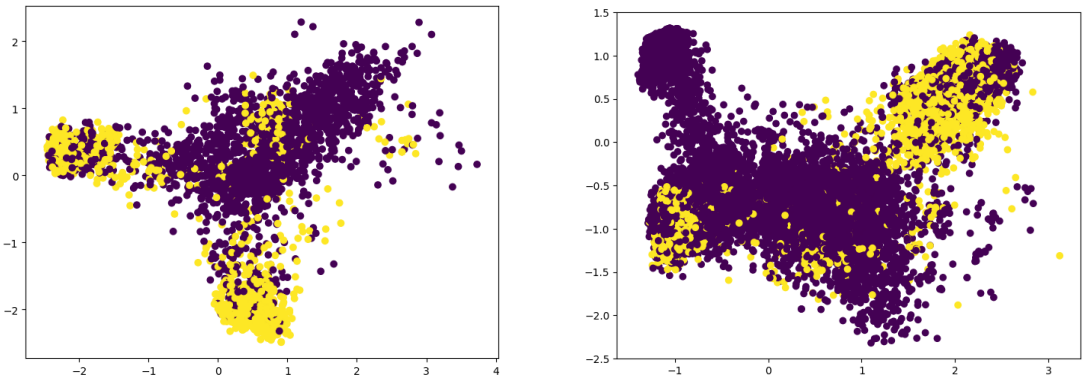
The code snippet shown in Listing 5 applies PCA to reduce the dimensionality of the training data to two components, which are then plotted to visualize the separation between classes. Each point in the plot represents an input file from the training set, positioned based on the first two principal components. The color of the point indicates the class label of the input file. First, an instance of the `PCA` class is created specifying the number of components. The model is then fitted to `X_train`, transforming the data into `X_pca`. A scatter plot is generated using the first principal component for the x-axis (`X_pca[:, 0]`) and the second for the y-axis (`X_pca[:, 1]`), with points colored according to the class label `y_train`.

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_train)
for i in range(0, 1, 2):
    plt.scatter(X_pca[:, i], X_pca[:, i+1], c=y_train)
plt.show()
```

Listing 5: Visualizing the data with a PCA scatter plot

The scatter plots shown in Figure 20 illustrate the training data under two scenarios: one based on a balanced dataset and the other on an imbalanced dataset. The balanced plot shows a relatively equal distribution of *valid* and *crash* samples, while the imbalanced plot highlights that *valid* samples predominate, as *crash* samples are scarce. In both cases, distinct clusters can be observed, although with some overlap, indicating that the features capture useful discriminatory information. Additionally, the presence of outliers, or points located far from the main distribution, may indicate potential anomalies or errors in the data. Furthermore, the varying spread of data points across the plots offers further insight into class distribution. For example, if one class is concentrated in a small region while the other is spread out, it might suggest that classifying the spread-out class could be more challenging for the model.

t-SNE is another powerful data visualization technique that projects high-dimensional data into lower-dimensional spaces while preserving local relationships. It is particularly effective in revealing clusters. The code snippet in Listing 6 initializes an instance of `TSNE` with two components, fits it to `X_train`, and stores the transformed data in `X_tsne`. A



(a) Balanced (b) Imbalanced

Figure 20: Two-dimensional PCA scatter plot of the data (author's own work)

scatter plot is then created, mapping the first component to the x-axis (`X_tsne[:, 0]`) and the second to the y-axis (`X_tsne[:, 1]`), with the class label `y_train` determining the color of each point.

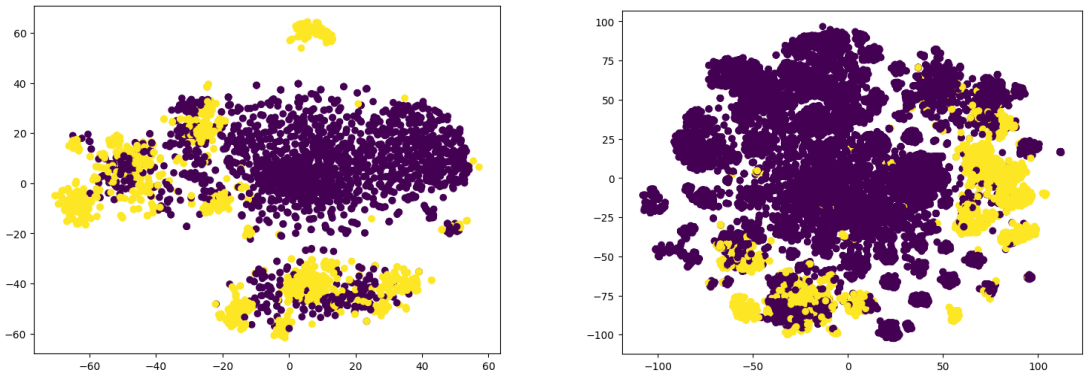
```
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=30)
X_tsne = tsne.fit_transform(X_train)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y_train)
plt.show()
```

Listing 6: Visualizing the data with a t-SNE scatter plot

The t-SNE visualization shown in Figure 21 offers an alternative nonlinear perspective on the dataset by emphasizing local relationships among samples. Unlike linear methods, t-SNE emphasizes finer structural details that may reveal subtle subclusters within each class. In the balanced scenario, clear local groupings of *valid* and *crash* samples emerge. In contrast, in the imbalanced case, *valid* samples form dense assemblies with only scattered *crash* instances. This intricate view of the feature space not only highlights potential transitional regions between classes but also identifies areas that may be more vulnerable to misclassification, which is helpful in guiding further model refinements.

Both visualizations complement each other, providing a holistic view of the dataset's structure. The PCA plots provide a global perspective, revealing distinct clusters with some degree of overlap, as well as highlighting outliers and differences in class distribution between balanced and imbalanced scenarios.

In contrast, the t-SNE plots offer a nonlinear view that emphasizes local relationships and finer subclusters, underscoring subtle differences between `valid` and `crash` samples and hinting at potential decision boundaries. Together, these plots confirm that the selected features effectively capture the inherent distinctions within the data while also highlighting areas that may present challenges for classification.



(a) Balanced (b) Imbalanced

Figure 21: t-SNE visualization of the similarity distribution in data (author's own work)

4.3.4 Normalizing the training data

Feature normalization is a common preprocessing step in ML, ensuring that features with different scales are transformed into a uniform range (Priyambudi & Nugroho, 2024). The code snippet in Listing 7 applies `Min-Max` scaling to normalize both training and test datasets, preventing information leakage between them.

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, FEATURE_RANGE_MAX))
X_train = pd.DataFrame(scaler.fit_transform(X_train)).values
X_test = pd.DataFrame(scaler.transform(X_test)).values
```

Listing 7: Normalizing the training data with the Min-Max method

`MinMaxScaler` is a utility class in Scikit-Learn that linearly transforms features to a given range, commonly between 0 and 1, so that each feature's minimum value is mapped to the specified lower bound and its maximum value to the upper bound (Baka, 2017). In this example, the scaler is instantiated with a feature range of `0` to approximately `0.68719476736`. This transformation standardizes the scale across features, ensuring that all values in both the training and test datasets fit within the chosen range without altering the relative differences between data points.

To tailor the scaler to the training dataset (`X_train`), the minimum and maximum values required for scaling are computed during the fitting process. The `fit_transform()` method simultaneously determines these values and transforms the data. The processed training data is then converted back into a `DataFrame`.

The test dataset (`X_test`) undergoes transformation using the same fitted scaler, ensuring consistency in scaling without introducing test data into the training set. The same minimum and maximum values previously computed are applied with the `transform()` method, thereby preventing information leakage from the test set into the training set.

Since both `X_train` and `X_test` datasets are structured with samples as rows and

features as columns, `MinMaxScaler` is applied directly without transposition. This format is commonly used for ML datasets in Python, especially when using `pandas.DataFrame` or `numpy.array`.

The value of `FEATURE_RANGE_MAX` can be adjusted according to the encoding method used to create decimal numbers from Base64-encoded strings based on the provided `CHARMAP`. By default, in standard Base64 encoding, each character represents a value from `0` to `63`. However, in my custom mapping, I assign `0` to the padding character `=` and `64` to the character `A`. For example, the `simple` encoding method, which concatenates six characters, produces a maximum decimal of `646464646464`, expressed as `0.646464646464` to represent the upper bound for normalization.

In contrast, the `product` encoding method calculates the final decimal as the product of the six numbers, resulting in a maximum value that will differ and should be defined accordingly (Lyu et al., 2018). This flexibility ensures that the normalization process aligns with the upper bound produced by the chosen encoding scheme.

4.3.5 Building the classification model

The code snippet in Listing 8 presents the implementation of the NN model using `Keras` (Chollet, 2015), a high-level API for building ML models in Python. `Keras` operates on top of frameworks such as `TensorFlow` and provides modular components that facilitate efficient model development. The implementation begins with the initialization of a `Sequential` model in line 4, which allows the construction of models layer-by-layer in a step-by-step fashion. This is followed by adding an `Input` layer of size 256 in line 5, which defines the number of input features.

The next step in line 6 is to add a `Dense` layer to the model. A `Dense` layer is a regular, deeply connected NN layer, which means that each neuron in this layer receives input from all neurons of its previous layer. Almost every NN model has `Dense` layers. In this case, the `Dense` layer has 256 neurons and uses the `ReLu` activation function that outputs the input directly if it is positive; otherwise, it outputs zero (Rasamoelina et al., 2020). It has become the default activation function for many types of NNs because the models that use it are easier to train and often perform better (Banerjee et al., 2019).

Line 7 adds a `Dropout` layer to the model, which randomly sets 50% of the neurons in the previous layer to zero during training. This helps prevent overfitting by reducing reliance on specific neurons, which encourages the model to learn more generalizable features. The model comprises two sets of `Dense` and `Dropout` layers to enhance the learning process while mitigating overfitting by introducing dropout and optional regularization controlled by the `REGULARIZATION` flag at multiple points in the network.

```
1 from tensorflow.keras.models import Sequential
2 from tensorflow.keras.layers import Dense
3
4 model = Sequential()
```

```
5  model.add(Input(shape=(256,)))
6  model.add(Dense(256, activation='relu', kernel_regularizer=REGULARIZATION))
7  model.add(Dropout(0.5))
8  model.add(Dense(256, activation='relu', kernel_regularizer=REGULARIZATION))
9  model.add(Dropout(0.5))
10 model.add(Dense(1, activation='sigmoid'))
```

Listing 8: Building the neural network model

The final layer created in line 10 is another `Dense` layer, which has only one neuron and uses the `sigmoid` activation function, which is commonly used in binary classification problems that transform the input into a range between 0 and 1. This layer is typically the last, and its single unit holds the binary prediction of the model (0 or 1). The choice of the `sigmoid` function is significant because it addresses the binary classification nature of the problem, providing a probability output that the given input point belongs to class 1. In this context, class 1 represents a prediction that the input sample has characteristics associated with causing a program crash based on patterns learned from training data. The model is then trained on labeled data to make predictions on new, unseen samples.

4.3.6 Training the classification model

The `model.compile()` function on line 15 in the code snippet shown in Listing 9 configures the model's learning process before training it.

```
1  from tensorflow.keras import Input
2  from tensorflow.keras.regularizers import L2
3  from tensorflow.keras.models import Sequential
4  from tensorflow.keras.layers import Dense, Dropout
5  from tensorflow.keras.callbacks import EarlyStopping
6
7  model = Sequential()
8  model.add(Input(shape=(256,)))
9  model.add(Dense(256, activation='relu', kernel_regularizer=REGULARIZATION))
10 model.add(Dropout(0.5))
11 model.add(Dense(256, activation='relu', kernel_regularizer=REGULARIZATION))
12 model.add(Dropout(0.5))
13 model.add(Dense(1, activation='sigmoid'))
14
15 model.compile(
16     optimizer='adam',
17     loss='binary_crossentropy',
18     metrics=['accuracy', Recall(), Precision()]
19 )
20
21 kfold = KFold(n_splits=4, shuffle=True)
22 for train, test in kfold.split(X_train, y_train):
23     model.fit(
24         x=X_train[train],
25         y=y_train[train],
26         validation_data=(X_train[test], y_train[test]),
27         callbacks=[EarlyStopping(monitor='val_loss', restore_best_weights=True)]
```

```
    )
```

Listing 9: Training the neural network model

`Adam` is the optimizer algorithm that adjusts the weights and biases of the model to minimize the loss function. It is a popular optimizer because it combines the advantages of two other extensions of stochastic gradient descent (Z. Zhang, 2018). The `binary_crossentropy` loss function measures the model's efficacy during training and should be minimized. It is a typical loss function in binary classification tasks, where the model's output represents the probability that each input instance belongs to the positive class (Ruby, Yendapalli, et al., 2020).

Accuracy, recall, and precision are the basic metrics selected for evaluation during training, validating, and testing. Accuracy measures the overall proportion of correct predictions, recall assesses the model's ability to identify all relevant instances, and precision evaluates how many of the predicted positive cases are actually correct. Together, these metrics provide a comprehensive assessment of the performance of the model.

After the configuration, the model is ready to be trained using the `model.fit()` function. Line 23 trains the model for a fixed number of epochs (iterations on a dataset). It uses the training data `X_train` and `y_train` and validates the model after each epoch using the test data `X_test` and `y_test`. The `EarlyStopping` callback stops the training when a monitored quantity has stopped improving. Monitoring the `val_loss` of the model helps to prevent overfitting by stopping the training process when the model starts to over-learn the training dataset, thereby optimizing the model's performance.

4.3.7 Evaluating the classification model

The code snippet shown in Listing 10 creates a plot of the metrics recorded during the model's training and validation against the number of epochs. The plot helps to visualize the model's learning process and understand whether the model is learning, overfitting, or underfitting from the training data.

The x-axis represents the number of epochs, and the y-axis represents the metric values. The x-axis is labeled `epochs` ranging from `0` to around `17.5`. An epoch is one complete pass through the entire training dataset (Sari et al., 2023). The y-axis represents normalized scales for losses and other metrics, ranging from `0.0` to `1.0` and it is explicitly labeled for each subplot. Each line in the plot corresponds to a different metric.

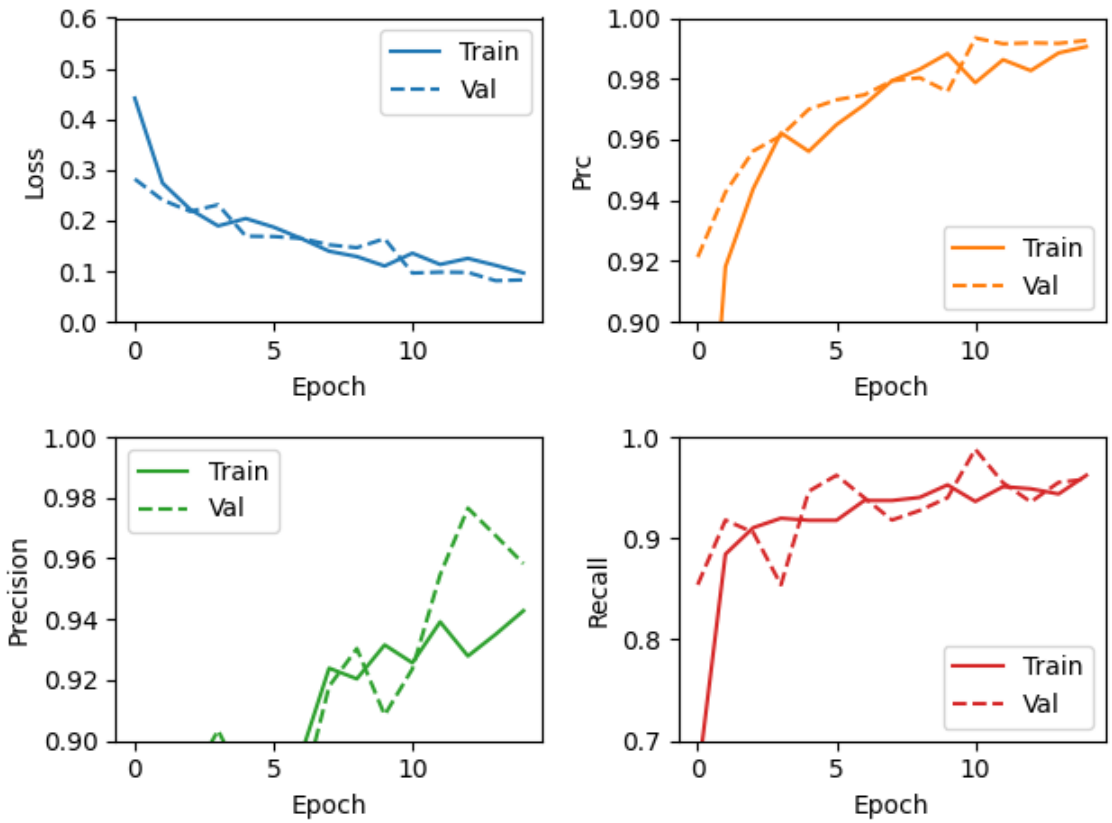
```
import pandas as pd
import matplotlib.pyplot as plt

metrics = ['loss', 'prc', 'precision', 'recall']
for n, metric in enumerate(metrics):
    train_scores = []
    val_scores = []

    name = metric.replace('_',' ').capitalize()
```

```
    plt.subplot(2,2,n+1)

    for history in histories:
        for train_score in history.history[metric]:
            train_scores.append(train_score)

        for val_score in history.history['val_'+metric]:
            val_scores.append(val_score)

    plt.plot(train_scores, color='C{}'.format(n), label='Train')
    plt.plot(val_scores, color='C{}'.format(n), linestyle='--', label='Val')
    plt.legend()

plt.suptitle('Essential Metrics')
plt.tight_layout()
plt.show()
```

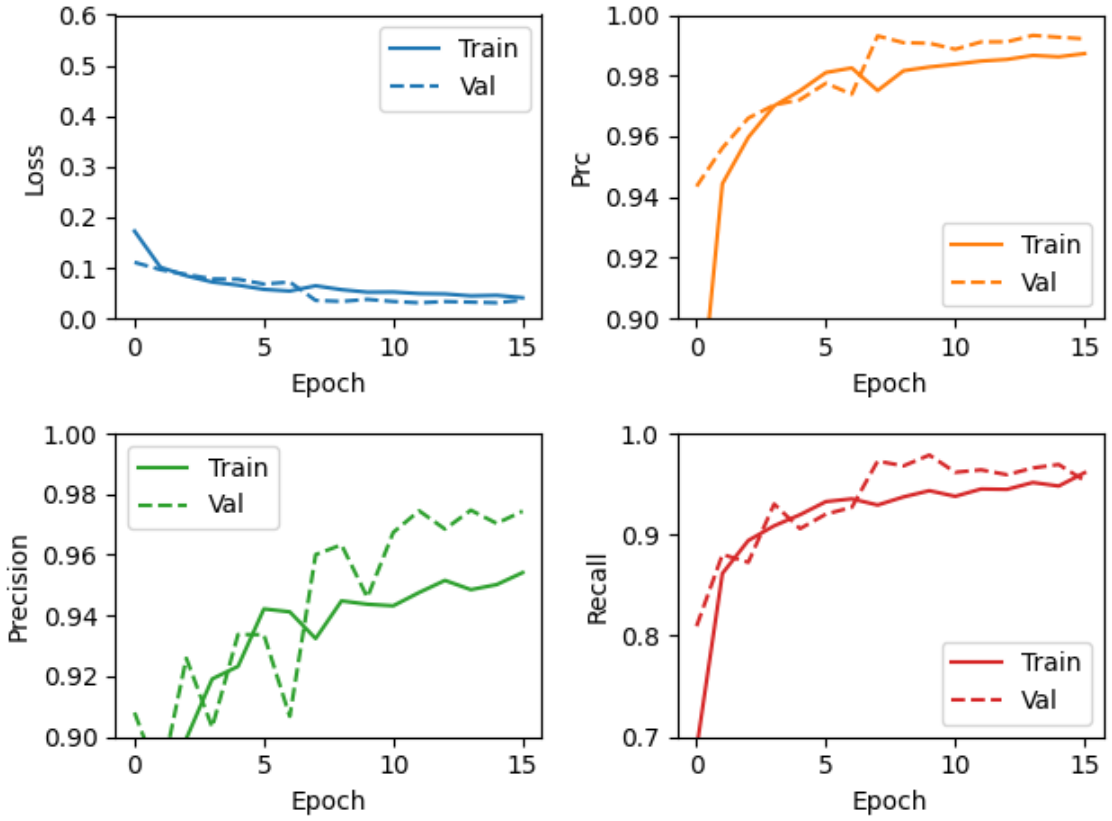
Listing 10: Plotting the learning history of the model

The plot shown in Figure 22 is a line graph that represents the learning history of the model over epochs. An epoch is one complete pass through the entire training dataset. During an epoch, the model learns by adjusting its internal parameters in response to the loss function, which measures its performance. The number of epochs determines how many times the learning algorithm works across the entire training dataset. Too few epochs can result in underfitting the model, while too many epochs can result in overfitting. Therefore, selecting the correct number of epochs is crucial for the performance of the model. The plot presents four essential metrics: loss, precision, recall, and precision-recall.

- **Loss:** The solid **blue** line represents the training loss, a measure of how well the model's predictions match the actual labels in the training data (Elharrouss et al., 2025). It starts high and gradually declines over epochs, indicating that the model is learning and adjusting its internal parameters to minimize prediction errors. The validation loss, indicated by the dashed blue line, follows a similar downward trend but exhibits more fluctuations, reflecting the variability in generalization.
- **Precision:** The solid **green** line represents the training precision (Ioualalen & Martel, 2019), which starts lower and gradually increases, demonstrating improved accuracy in identifying positive instances. The dashed green line for validation precision also shows an increasing trend, but exhibits more variability.
- **Recall:** The solid **red** line illustrates training recall (Winkler et al., 2019), which begins relatively high and maintains stability over epochs. The dashed red line, representing validation recall, starts similarly high, dips slightly, and then stabilizes, reflecting some fluctuations before reaching consistency.
- **PRC:** The solid **orange** line represents the training precision-recall curve (Davis & Goadrich, 2006; J. Miao & Zhu, 2022), which starts lower and increases over epochs, suggesting an improved balance between precision and recall. The dashed orange line for validation follows a similar trend, but with noticeable fluctuations.



(a) Balanced



(b) Imbalanced

Figure 22: Learning progress over epochs (author's own work)

In general, optimizing for recall reduces the number of false negatives, allowing a model to capture a larger proportion of actual positive instances. Although this approach may lead to an increase in false positives, the trade-off is acceptable in the context of seed selection for fuzz testing. This focus on recall is appropriate because the goal is to identify and prioritize files that resemble crash samples associated with known bugs or security vulnerabilities.

Since all candidate files have been confirmed to be valid metafiles, the task is to select those samples that closely mirror the characteristics of samples that triggered program crashes in previous test campaigns. In this context, false negatives are files with potentially relevant characteristics of software flaws, thereby diminishing the comprehensiveness of the analysis. In contrast, false positives merely add additional files for subsequent examination.

4.3.8 Predicting valuable seed inputs

The code snippet shown in Listing 11 loads new and unseen data, preprocesses it, and makes binary predictions using the trained model. Files predicted as *crash* samples may be considered potential seed inputs for new fuzz testing campaigns. This prototype is a starting point for predicting valuable seed inputs for fuzz testing.

The `load_data()` function in line 3 loads unseen input files from the `candidates` directory to make predictions. Note that this is the same function that is used to load the labeled data; however, in this case, the label has no meaning and will be removed from the dataset. The `from_dict()` method in line 5 converts the loaded data dictionary into a `pandas.DataFrame`, a two-dimensional labeled data structure. The `orient='index'` argument transposes (swaps rows with columns) the `pandas.DataFrame` and specifies that the keys in the dictionary should be rows. The `add_prefix()` method adds the prefix `f` to all the column names in the `pandas.DataFrame`.

```
1  import pandas as pd
2
3  data = load_data(data, 'candidates')
4
5  df = pd.DataFrame.from_dict(data, orient='index').add_prefix('f')
6
7  X_pred = df.drop('f0', axis=1).values
8  X_pred = pd.DataFrame(scaler.transform(X_pred))
9
10 preds = (model.predict(X_pred) > THRESHOLD).astype('int32')
```

Listing 11: Making predictions with the trained model

Line 7 prepares the data for prediction by dropping the `f0` column containing the labels from the `pandas.DataFrame` and then converting it into a `numpy.array`. The resulting `X_pred` contains the features the model will use to make predictions. Line 8 scales the features using the previously defined `MinMaxScaler` scaler. The `transform()` method

scales `X_pred` and converts it back to a `pandas.DataFrame`. The `model.predict()` method shown in line 10 is used to make predictions on `X_pred`.

Since this is a binary classification problem, the output is a probability, translated into binary predictions. If the prediction of the model is more significant than the `THRESHOLD`, it will be considered `True` (or `1`); if it is less than or equal to the `THRESHOLD`, it will be considered `False` (or `0`). This is a common approach in binary classification problems, where data is divided into two distinct categories. Finally, the `astype()` method converts the boolean values (`True`/`False`) to integer values (`1`/`0`), specifying the desired data type for the output as a 32-bit integer. The variable `preds` stores the results as an array of 1s and 0s representing the model's predictions for the input data `X_pred`.

5. Empirical assessment

5.1 Setup and execution

The most effective approach to assessing the application of ML in fuzzing is to conduct actual fuzz testing of current software. However, this method has several limitations and is highly time-consuming. During the development of the concept presented in this dissertation and while implementing the working prototype of the `SeedClass` tool, I identified multiple zero-day vulnerabilities in the Windows graphics component. In this context, the vulnerabilities uncovered already serve as a reliable indicator of the tool's usefulness.

Furthermore, additional experiments presented in this chapter also confirm that leveraging the results of previous test campaigns enables a more systematic approach to guiding the fuzzer via seed file selection. To demonstrate the impact of selecting seed files based on the predictions of the binary classification model presented in this work, I conducted several tests on an outdated version of the Windows 10 operating system, known to be affected by several security vulnerabilities, to compare the results of multiple fuzzing campaigns, each using a different strategy to select seed files from the same candidate dataset. These tests compared the performance of four distinct seed selection approaches, which are detailed in the following list.

- **Randomization:** By using the randomization strategy, my aim was to demonstrate the default state in which no attempt is made to improve the effectiveness of the fuzz testing process through the selection of initial test data. I shuffled the entire corpus and then selected 20 files at random using the `SeedGroup.py` helper script, which I created for this experiment.
- **Minimization:** The minimization strategy aims to select the testcases from the candidate dataset that provide the best code coverage. I minimized the corpus using the `winafl-cmin.py` tool from the `WinAFL` project. This tool drastically reduces the number of testcases while maintaining code coverage by eliminating redundant inputs. Because it is designed to maximize code coverage with a given corpus, I slightly modified it to return only the top 20 files for this specific scenario in the experiment.
- **Prediction:** The goal of the prediction strategy is to take advantage of the knowledge gained from previously identified program crashes to select seed files, thereby improving the bug-finding capability of the fuzzer. I used a working prototype of `SeedClass` to predict valuable seed inputs. I trained the binary classification model on the dataset collected for this research, which is documented in previous chapters and is available in the code repository of the project. I selected the seed files based

on the predictions from the trained model.

- **Combination:** The combination strategy aims to leverage the advantages of the different approaches collectively while mitigating their specific disadvantages. This strategy involves the joint application of the minimization and prediction strategies. First, I reduced the candidate dataset to a minimal set of 121 files using the original `winafl-cmin.py` tool without any modifications. Second, I used the predictions of the `SeedClass` prototype to select the top 20 files for fuzzing.

I used the `SeedGroup.py` helper script to compile a group of 1000 files for these experiments as a candidate dataset, randomly picked from the entire corpus, which consisted of 24,245 files in total. The selected files have been removed from the training dataset to prevent any adverse effects on the model's training. Using the various seed selection strategies described above, I selected 20 seed files from this separated dataset for each experiment. Then, I started fuzz testing campaigns using the selected files as initial seeds. Each campaign consisted of 20 parallel fuzzer instances, which ran for approximately 24 hours. I conducted the experiments on the same physical device with identical configuration settings over consecutive days without restarting the machine.

Each fuzz testing campaign ran across 20 threads in parallel, amounting to approximately 20 days of cumulative testing time. Mutated testcases were transmitted between the test harness and the fuzzer using shared memory mode, which was enabled with the `-s` command line option. For dynamic instrumentation, I configured `edge` coverage tracking for the `GdiPlus.dll` module with 5,000 execution iterations. To maintain consistent instrumentation settings, I applied the mentioned parameters using the `-e` command line option of the `WinAFL` fuzzer. The same 32-bit fuzzing harness was used for all tests, providing access to the targeted Windows GDI functions via the exported `Fuzz` method, which expects a single parameter. I manually started and stopped each testing campaign. As a result, the total runtime typically varied by approximately 10 minutes instead of running for a full 24 hours. However, this slight variation does not significantly affect the results.

5.2 Results and insights

After 24 hours, I gathered the complete output folder for each campaign. This folder contains the files that caused a program crash, as well as the `fuzzer_stats` and `plot_data` files, which include essential metrics recorded during the tests. Based on the collected output folder, I retrieved the detailed metrics using the `CrashResults.py` helper script for each fuzzing campaign, partially based on the original `winafl-whatsup.py` script provided with the `WinAFL` project.

Table 7 compares the results of the campaigns according to the seed selection strategies used, based on key performance metrics, with the best value for each metric highlighted in bold to indicate the most effective strategy for that specific measure. Experiments regard-

Strategy	Execs	Speed	Paths	Pending	Coverage	Stability	TTC	Crashes
Randomize	182M	2.1k	**19k**	131k	38.81%	68.11%	01:17	46
Minimize	**252M**	**2.9k**	18k	117k	39.11%	65.48%	01:11	59
Predict (balanced)	212M	2.5k	17k	118k	39.04%	64.66%	**00:11**	**108**
Predict (imbalanced)	188M	2.2k	16.2k	109k	38.51%	**70.17%**	00:54	73
Combine	154M	1.8k	17k	**137k**	**40.37%**	69.47%	04:27	62

Table 7: Summary of fuzzing output statistics

ing the prediction strategy involved using two different distributions of the same training dataset. I adjusted the balanced dataset using the `CrashDistribute.py` helper script to remove most redundant sample files, while the imbalanced dataset contains all available samples. The balanced dataset served the purpose of preventing the model from learning a specific bug that was overrepresented in the crash samples. The following are the key insights:

- **Speed:** The minimization strategy achieved the highest execution count (252M) at a speed of 2900 executions per second, suggesting extensive testing. The randomization strategy had 182 million executions, running at 2100 executions per second, while the imbalanced prediction strategy operated at 2200 executions per second.
- **Progress:** The randomization strategy discovered the most unique paths (19k), slightly outperforming the minimization (18k) and the combination (17k) strategies. The combination strategy had the highest number of pending inputs (137k), indicating a large number of interesting testcases.
- **Coverage:** The combination strategy achieved the highest code coverage (40.37%), surpassing other strategies. The imbalanced prediction strategy exhibited the highest stability at 70.17%, ensuring consistent execution conditions.
- **Bug-finding** The balanced prediction strategy was the most effective at bug detection, producing 108 crashes with a time to crash (TTC) of just 11 seconds. In contrast, the randomization strategy recorded 46 crashes with a TTC of 1 minute and 17 seconds, while the combination strategy had 62 crashes, taking 4 minutes and 27 seconds to detect the first bug.

Based on the results of the experiments, where the TTC and the number of crashes were the most important metrics, the balanced prediction strategy proved to be the most effective in identifying bugs. It produced many crashes with a short TTC, demonstrating better efficiency in bug discovery. In contrast, the minimization strategy resulted in fewer program crashes, taking longer to detect the first bug, indicating a slower and less effective approach. Although the combination strategy achieved the highest code coverage, its longer TTC and fewer crashes suggest that broader exploration does not necessarily lead to faster bug identification. These findings reinforce that leveraging prior knowledge of program crashes can enhance fuzz testing efficiency for bug discovery. The ability of the various seed selection techniques to identify bugs is summarized in Table 8, which

also highlights the advantages and disadvantages of each approach in identifying program crashes.

The *randomization* strategy demonstrated satisfactory path exploration, finding the most execution paths (19k), while having a relatively lower coverage rate (38.81%), suggesting that it starts with a modest initial coverage, but the rate increases as testing progresses. Despite identifying the fewest crashes (46), it still uncovered a unique crash in the `SetROP2()` function, along with the minimization strategy, that was undetected by the more targeted approaches. This result highlights its ability to effectively explore more code paths, potentially revealing edge cases that more systematic strategies may overlook. However, its lower crash detection rate overall suggests that while it excels in exploration, it may benefit from refinements to enhance the bug-finding capabilities of the fuzzer.

The *minimization* strategy excels in execution speed and efficiency, achieving the highest number of executions (252M) and the fastest cumulative speed (2.9k execs/sec). It also detected more crashes (59) than the randomization strategy. However, the fuzzer's performance across other metrics, such as path discovery, pending testcases, and coverage, did not show significant improvement. While it discovered a bug in the `NamedEscape()` function alongside the *randomization* strategy, it identified fewer unique bugs overall. This result suggests that while testcase minimization optimizes the fuzzing throughput, it does not substantially enhance the fuzzer's ability to discover new bugs. Its primary benefit lies in maximizing execution rate rather than improving vulnerability detection, making it ideal for rapid test cycles but potentially limited in uncovering deeper flaws in the system.

BugId	Randomization	Minimization	Prediction (balanced)	Prediction (imbalanced)	Combination
0e5.7fb	11	16	16	4	3
2b1.cf1					1
3ce.9b5	35	10	17	10	13
520.506	1	7			
5ab.a57					5
6e7.9de				1	
8ed.260			1261	847	806
8ed.13f			3	4	2
999.7fb	1				
be9.1bc			1		
d05.7fb		6	76	2	
db3.f44	28	1	1		
eb3.7fb	11	23	34	32	16
efe.7fb	3	9	11	1	2
f33.8ed			2603	2464	1726
f33.cf7			158	104	70

Table 8: Distribution of unique bugs identified

The *prediction* strategy using a *balanced* dataset demonstrated the best bug-finding efficiency, identifying the most crashes (108) and achieving the fastest time-to-crash (00:11).

This result is consistent with the methodology of using historical crash data to guide the fuzzer toward inputs likely to replicate known failure patterns. Compared to the *randomization* strategy, which excelled in path discovery but found fewer crashes, the *prediction* strategy prioritized crash detection while maintaining competitive execution speed. Unlike the *minimization* strategy, which optimizes throughput but has limited impact on bug discovery, the *prediction* strategy offers a more effective trade-off between execution speed and crash identification. The results suggest that predictive techniques could enhance testing efficiency by directing the fuzzer towards known vulnerable code paths without compromising execution speed or broader code coverage.

Although training the model with the *imbalanced* dataset of all available samples improved its performance and helped identify a new bug in the `GlyphDataElement<...>::GetExistingGlyph()` function, it did not significantly affect other important metrics. The distribution of the bugs also indicates that the predicted samples directed the fuzzer toward bugs in the `DpRegion::Set` and `DpRegion::And` functions that were overrepresented in the dataset. The observed behavior suggests that training the model to be less accurate is beneficial in preserving the fuzzer's ability to explore new, previously unseen code paths.

The *combination* strategy strikes a balance between enhancing crash detection and improving code coverage. The coverage and stability metrics indicate that this approach is better suited for systematic bug discovery, as they demonstrate adequate path execution exploration while maintaining stability.

Multiple seed selection strategies consistently detected some crashes, confirming that most methods can reliably identify certain bug patterns. Only the *randomization* and *minimization* strategies were able to detect the bug in the `NamedEscape()` function, indicating their potential for identifying unseen bugs. The *combination* strategy successfully identified bugs in the `CopyGindices()` and `CreateFontIndirectWImpl()` functions, showing that combining different seed selection methods can also increase the bug-finding capability of the fuzzer.

In conclusion, the *prediction* strategy was the best for directed fuzz testing using failure patterns from historical crash data. The *combination* strategy balanced testing stability and code coverage while maintaining bug discovery rates. The *randomization* strategy performed well in increasing code coverage and successfully identified a bug in the `SetROP2()` function. At the same time, the *minimization* strategy optimized testing efficiency but had limited effectiveness in uncovering novel program crashes.

5.3 Vulnerabilities discovered

Through multiple fuzz testing campaigns, each lasting approximately 10 to 14 days, this investigation uncovered several unique crashes in the Windows graphics component. A

detailed review of the results identified more than 50 distinct crash samples, with potential impacts ranging from information disclosure and elevation of privilege to remote code execution. Seven of these crashes have been confirmed to be previously unknown zero-day vulnerabilities, which allow information disclosure.

These vulnerabilities arise when the Windows graphics component processes EMF files containing specially crafted metafile records. This can result in memory reads beyond the boundaries of an allocated buffer, potentially exposing both initialized and uninitialized heap memory. An attacker could leverage these vulnerabilities, along with others, to execute code in the context of the current process. Successful exploitation requires user interaction, meaning that the victim must open a malicious file or visit a compromised website. Stack backtraces for the identified vulnerabilities are included in Appendix 6.4.

Security researchers often rely on MITRE (1999) [26] to identify vulnerabilities in open-source software. However, closed-source firms like Microsoft typically have their own procedures for receiving and handling vulnerability reports. As soon as I triaged the identified crashes and recognized a potential vulnerability, I promptly forwarded the details to the Microsoft Security Response Center (MSRC)[27]. Following a thorough investigation, MSRC confirmed my findings and resolved the issues through security fixes. The following sections outline the key details of the technical analysis of these vulnerabilities.

5.3.1 CVE-2022-21904

The first vulnerability may lead to the exposure of small portions of heap memory when the Windows graphics component converts an EMF metafile containing a specially crafted `EMR_SETDIBITSTODEVICE` record to the WMF format (Selján, 2021c).

The `Metafile::EmfToWmfBits()` method converts an EMF metafile to a WMF metafile and stores the transformed records in a buffer. The root cause of this vulnerability lies in the `DoSetDIBitsToDevice()` function, which calls `memcpy()` with an invalid size parameter. Before executing the actual device-independent bitmap (DIB) setting operation, the `DoSetDIBitsToDevice()` function attempts to verify the validity of the provided EMF metafile record by first calling the `IsValidEnhMetaRecordBitmapEx()` function, which initiates a series of validation checks.

```
1 int bCheckBitmapInfoHeader(...) {
2   if (cbBmiSrc >= 0xC) {
3     srcBmi = (emr + offBmiSrc);
4     biSize = *(emr + offBmiSrc);
5     if ( cbBmiSrc >= biSize
6       && CalculateColorTableSize(...)
7       && (!UsageSrc || ULongAdd(...) >= 0 && (biSize = v13, cbBmiSrc >= v13) ) {
```

[26] MITRE is a non-profit organization responsible for maintaining and managing the Common Vulnerabilities and Exposures (CVE) system, a public database that assigns unique identifiers to known vulnerabilities (MITRE, 1999).

[27] MSRC is Microsoft's dedicated team for identifying, analyzing, and responding to security threats across its products and services.

```
...SNIP...
      dstBmi = LocalAlloc(0, biSize); // 0x28
      if (dstBmi) {
        memcpy(dstBmi, srcBmi, biSize);
        dstBmi->bmiHeader.biClrUsed = colorTableSize;
        LOBYTE(UsageSrc) = DIB_PAL_COLORS;
        BitmapInfoPtr::Attach(bmiPtr, dstBmi, UsageSrc);
        return 1;
      }
    }
  }
  return 0;
}
```

Listing 12: Decompiled source code of the `bCheckBitmapInfoHeader()` function

Among these validation routines, the `bCheckBitmapInfoHeader()` function appears to allocate memory for a new `BITMAPINFO` structure using the value in the `biSize` member of the source bitmap's `BITMAPINFOHEADER` structure rather than the `cbBmiSrc` field of the `EMR_SETDIBITSTODEVICE` record, which is also passed to it as a function parameter through the following call tree.

```
bHandleSetDIBitsToDevice()
└── IsValidEnhMetaRecordBitmapEx()
    └── bCheckBitmap()
        └── bCheckBitmapInfo()
            └── bCheckBitmapInfoHeader()
                └── LocalAlloc(0, biSize)
```

During the conversion from EMF to WMF, the process reads beyond the allocated buffer that holds the source `BITMAPINFO` structure. This occurs because the allocated buffer is only `0x28` bytes, which is smaller than the size of the `BITMAPINFO` structure specified in the `cbBmiSrc` field of the `EMR_SETDIBITSTODEVICE` record. Microsoft addressed the vulnerability by allocating a buffer that corresponds to the size specified in the `cbBmiSrc` field of the `EMR_SETDIBITSTODEVICE` record (MSRC, 2022a).

5.3.2 CVE-2022-21915

The second vulnerability related to the handling of `EMR_STRETCHDIBITS` records could potentially expose uninitialized heap memory when the Windows graphics component processes EMF files (Selján, 2021d).

When the `bcSize` member of the `BITMAPCOREHEADER` structure is not set to the minimum header size of `0xC`, the `MRBDIB::vInit()` function calls the `memcpy()` function using the provided size parameter. However, the function only checks the existence of the parameter without validating its value. Consequently, the function may pass an excessively large value, such as `0x1c0098`, to the `memcpy()` function as the size parameter.

```
int bMetaGetDIBInfo(HDC hdc, 0, LPBITMAPINFO lpbmi, int &size, int &sizeImage, UINT
    ColorUse, int cLines, 0) {
```

```
...SNIP...
  biBitCount = lpbmi->bmiHeader.biBitCount;
  if (biBitCount == 16 || biBitCount == 32) {
...SNIP...
  } else if (biBitCount != 24 &&
            lpbmi->bmiHeader.biCompression != BI_JPEG &&
            lpbmi->bmiHeader.biCompression != BI_PNG)
  {
    biClrUsed = lpbmi->bmiHeader.biClrUsed; // 0x7001c
    if (biClrUsed) {
      // 0x1c0070    =        4        *  0x7001c
      colorTableSize = sizeof(RGBQUAD) * biClrUsed;
      goto LABEL_28;
    }
    if (biBitCount < 16) {
      colorTableSize = cbBits * (1 << biBitCount);
LABEL_28:
      // 0x28 +     0x1c0070     =  0x1c0098
      biSize += colorTableSize;
      goto LABEL_12;
    }
  }
LABEL_12:
...SNIP...
  biWidth = lpbmi->bmiHeader.biWidth;
  if (biWidth >= 0)
    res = CJSCAN(biWidth,
                 lpbmi->bmiHeader.biPlanes,
                 lpbmi->bmiHeader.biBitCount,
                 &cbBits);
  if (res) {
    if (!cLines) {
      cLines = lpbmi->bmiHeader.biHeight;
      if (cLines < 0)
        cLines = -cLines;
    }
    if (ULongLongToULong(cbBits * cLines, cbBits * cLines >> 32) >= 0) {
      biSizeImage = cbBits;
LABEL_21:
      *size       = biSize;      // 0x1c0098
      *sizeImage  = biSizeImage; // 0x8b80
      return 1;
    }
  }
  return 0;
}
```

Listing 13: Pseudocode of the `bMetaGetDIBInfo()` function

The value of `0x1c0098` is derived from the `bMetaGetDIBInfo()` function, which is called by the `EMF_AnyDIBits()` function prior to the execution of the `MRBDIB::vInit()` function. In this context, the `bMetaGetDIBInfo()` function assigns the `&size` parameter based on the `biClrUsed` member of the `BITMAPINFOHEADER` structure. The underlying issue that causes the vulnerability lies in the fact that the `biClrUsed` member should not exceed 256, as the `biBitCount` member has already established this limit.

An attacker can leverage this vulnerability by using an EMR_STRETCHDIBITS record that includes a BITMAPINFOHEADER structure with an excessively large biClrUsed value. This manipulation allows the attacker to invoke the memcpy() function with an arbitrary size parameter. Microsoft addressed this vulnerability by introducing a new GetBitmapFromRecordEx() function, which incorporates additional validation logic to check the biClrUsed member of the BITMAPINFOHEADER structure (MSRC, 2022b).

5.3.3 CVE-2022-26934

The third vulnerability occurs in the handling of EMR_CREATEDIBPATTERNBRUSHPT records when the Windows graphics component processes EMF metafiles. This flaw may result in reading beyond the end of an allocated buffer, which could potentially expose either initialized or uninitialized heap memory (Selján, 2021b).

The root cause of the vulnerability lies in the MfEnumState::ModifyDib() function, which reads beyond the BITMAPINFO structure. This occurs because of an invalid value in the biWidth member of the BITMAPINFOHEADER structure, which indicates the width of the bitmap in pixels. Below is an example of the affected BITMAPINFOHEADER structure:

```
BITMAPINFOHEADER bmiHeader = {
  .biSize          = 0x28,    // Header size in bytes
  .biWidth         = 0x901,   // Bitmap width
  .biHeight        = 0x1,     // Bitmap height
  .biPlanes        = 0x1,     // Number of color planes
  .biBitCount      = 0x1,     // Bits per pixel
  .biCompression   = 0x0,     // Compression method (0 = BI_RGB)
  .biSizeImage     = 0x4,     // Image size (may be 0 for BI_RGB)
  .biXPelsPerMeter = 0x0,     // Horizontal resolution (pixels per meter)
  .biYPelsPerMeter = 0x0,     // Vertical resolution (pixels per meter)
  .biClrUsed       = 0x0,     // Number of colors in the color palette
  .biClrImportant  = 0x0      // Number of important colors used
};
```

The GetDibByteWidth() function determines the size parameter for the memcpy() function, which in turn invokes the GetDibBitsSize() function. The GetDibByteWidth() function calculates the number of bytes from one row of pixels to the next using the formula (((biWidth*biBitCount)+0x1F)&0x1FFFFFFC)>>0x3)=stride, which results in ((((0x901*0x1)+0x1F)&0x1FFFFFFC)>>0x3)=0x124 after substituting the appropriate values.

```
int EmfEnumState::CreateModifiedDib(...) {
  result = 0;
  if ( Src->biSize < 0x28
    || !GetDibNumPalEntries(0, Src->biSize, Src->biBitCount, Src->biCompression, Src->
    biClrUsed, &biSize)
    || !GetDibBitsSize(Src, &Size)
    || Size <= 0 ) {
    return result;
```

```
  }

  if (biSize != 2 || rop == SRCCOPY || *(&Src->biSize + 0x28) || *(&Src->biWidth + 0
    x28) != 0xFFFFFF) {
    v9 = *a4;
    dstSize = MfEnumState::GetModifiedDibSize(Src, biSize, Size, a4);
    if ( dstSize > 0 ) {
      if ( MfEnumState::CreateRecordToModify(dstSize) ) {
        Dst = *(this + 0x98);
        MfEnumState::ModifyDib(this, Src, v9, Src, a3, Dst, biSize, Size, 1); // Crash
     here!
      }
    }
    return Dst;
  }

  return 0;
}
```

Listing 14: Pseudocode of the `EmfEnumState::CreateModifiedDib` function

The issue arises because the `EmfEnumState::CreateModifiedDib()` function does not adequately verify the validity of the source bitmap prior to invoking the `MfEnumState::ModifyDib()` function. This latter function conducts a bit-block transfer of the color data by directly copying the source shape to the destination shape. As a result, it may attempt to copy more bitmap data than the actual size of the source bitmap allows.

An attacker can leverage this vulnerability by manipulating the `biWidth` member in the `BITMAPINFOHEADER` structure, allowing them to invoke the `memcpy()` function with an arbitrary size parameter. To address this issue, Microsoft implemented a mitigation that verifies the record size using the `IsValidBitmapRecordSize()` function before continuing with the `EmfEnumState::CreateModifiedDib()` function (MSRC, 2022c).

5.3.4 CVE-2022-29112

The fourth vulnerability occurs in the `ValidateBitmapInfo()` function when the Windows graphics component processes a specially crafted `EMR_BITBLT` record within an EMF file (Selján, 2021a).

The `ValidateBitmapInfo()` function converts `RGB` bitmap colors into a palette of `ARGB` color values. It attempts to loop through the `bmiColors` member of the `BITMAPINFO` structure based on the `biClrUsed` member's value, which may lead to reading memory beyond the actual array boundaries.

```
int ValidateBitmapInfo(...) {
...SNIP...
  if ( nColors ) {
...SNIP...
    pltEntry = palette->Entries;
    if (&palette->Entries[nColors] >= palette->Entries ? nColors >> 2 : 0) {
      i = 0;
      clrsUsed = &palette->Entries[nColors] >= palette->Entries ? nColors >> 2 : 0;
```

```
9       do {
10        blue = bmiColor->rgbBlue;
11        arg = (bmiColor->rgbGreen | (((bmiColor->rgbRed | 0xFFFFFF00) << 8)) << 8; //
      Crash here!
12        ++bmiColor;
13        ++i;
14        pltEntry->argb = blue | arg;
15        ++pltEntry;
16 ...SNIP...
17      }
18      while ( i < clrsUsed );
19    }
20  }
21 ...SNIP...
22 }
```

Listing 15: Pseudocode of the `ValidateBitmapInfo()` function

The `ValidateBitmapInfo()` function only checks that the value of the `biClrUsed` member is less than the number of color entries indicated by the `biBitCount` member. However, it does not detect when the value of `biClrUsed` does not match the actual number of entries set in the `bmiColors` member of the `BITMAPINFO` structure.

The `BITMAPINFO` structure defines the dimensions and color information for a DIB. Within this structure, the `bmiColors` member contains an array of `RGBQUAD` entries. The size of this array is determined by the values of the `biBitCount` and `biClrUsed` members found in the `BITMAPINFOHEADER` structure.

```
1  void MfEnumState::OutputDIB(...) {
2  ...SNIP...
3    entrySize = 2 * (iUsageSrc == 0) + 2; // 0x2 * (0x20000b1 == 0x0) + 0x2 = 0x2
4    if (palEntries <= 0xFFFFFFFF / entrySize) {
5      palSize = palEntries * entrySize;   // 0x12 * 0x2  = 0x24
6      Size = biSize + palSize;            // 0x28 + 0x24 = 0x4c
7      if (Size >= palSize) {
8        Dst = GpMallocEx(Size, 0);        // 0x4c
9        palEntries = Dst;
10       if (Dst) memcpy(Dst, Src, Size);
11     }
12   }
13 ...SNIP...
14 }
```

Listing 16: Pseudocode of the `MfEnumState::OutputDIB()` function

The root cause of the issue lies in the `MfEnumState::OutputDIB()` function, which miscalculates the size of the `BITMAPINFO` structure due to a corrupted `0x20000b1` value in the `iUsageSrc` field of the `EMR_BITBLT` record. As a result, the calculated size of `0x4c` is smaller than the actual size of the bitmap. Microsoft addressed this vulnerability by adding a check at the beginning of the `MfEnumState::OutputDIB()` function to confirm that the value of the `iUsageSrc` field in the `EMR_BITBLT` record is either `DIB_RGB_COLORS` or `DIB_PAL_COLORS` (MSRC, 2022d).

5.3.5 CVE-2022-34728

The fifth vulnerability involves the processing of a specially crafted `EMR_SETPIXELV` record within an EMF metafile, which may lead to an out-of-bounds read access violation in the `ModifyRecordColor()` function (Selján, 2022b).

The `EmfEnumState::ProcessRecord()` function calls the appropriate handler function for each record based on its type. For `EMR_SETPIXELV` records, this handler is the `EmfEnumState::ModifyRecordColor()` function. This function sets the pixel at the specified coordinates to the closest approximation of the chosen color.

```
1  void EmfEnumState::ModifyRecordColor(int this, int ColorAdjustTypeBrush, int
       ColorAdjustTypePen) {
2    pixel = *(this + 0x5C);
3    if (pixel) {
4      if (*(this + 0x60) >= (4 * ColorAdjustTypeBrush)) {   // 0x10 >= 0x8
5        oldPalEntry = *(4 * ColorAdjustTypeBrush + pixel);  // Crash here!
6        newPalEntry = MfEnumState::ModifyColor(this, oldPalEntry, ColorAdjustTypePen);
7        if (newPalEntry != oldPalEntry) {
8          if (EmfEnumState::CreateCopyOfCurrentRecord(this, Size))
9            *(*(v3 + 0x98) + 4 * ColorAdjustTypeBrush + 8) = newPalEntry;
10 }}}}
```

Listing 17: Pseudocode of the `EmfEnumState::ModifyRecordColor` function

The root cause of the vulnerability lies in the `ModifyRecordColor()` function's failure to verify if the value `0x10` specified in the Size field of the `EMR_SETPIXELV` record corresponds to the actual size of the record, which is `0x14`. As a result, the `GetEmf()` function allocates memory for the metafile based on the `nBytes` field of the `EMR_HEADER` record. This allocation creates a buffer that is 4 bytes smaller, effectively truncating the `crColor` field of the `EMR_SETPIXELV` record. Consequently, this oversight can lead to the leakage of several bytes from the adjacent memory area.

An attacker who successfully exploits the vulnerability could leak small portions of the process's heap memory, because the `ModifyRecordColor()` function retrieves the next four bytes after the record, followed by the `ptlPixel` field. However, the attacker can only leak three useful bytes, as the operation sets the high-order byte to zero.

5.3.6 CVE-2022-35837

This sixth vulnerability occurs when the Windows graphics component processes a specially crafted `EMR_STARTDOC` record within an EMF metafile, potentially leading to several arbitrary memory read operations in the `StartDocWImpl()` function (Selján, 2022a).

```
1  int StartDocWImpl(HDC hdc, _DOCINFOW *lpdi) {
2  ...SNIP...
3    if (lpdi && lpdi->cbSize == 0x14 &&
4              lpdi->lpszDatatype &&
5              lpdi->fwType <= 1 &&
6             __wcsicmp(L"emf", lpdi->lpszDatatype) ) // Crash here!
```

```
  {
...SNIP...
  }
...SNIP...
}
```

Listing 18: Pseudocode of the `StartDocWImpl()` function

The `StartDocWImpl()` function performs a string comparison using user-controlled data. The following decompiled source code demonstrates how an attacker can directly manipulate the second parameter of the `__wcsicmp()` string comparison function:

```
DOCINFO di = {
  .cbSize       = 0x14,               // Structure size in bytes
  .lpszDocName  = (LPCSTR)0x1e,       // Pointer value for the document name
  .lpszOutput   = (LPCSTR)0xb4,       // Pointer value for the output destination
  .lpszDatatype = (LPCSTR)0x210000e6, // Address passed to __wcsicmp()
  .fwType       = 0x1                 // Flag indicating the document type
};
```

Listing 19: Example `DOCINFO` structure

The `lpdi` field of the `EMR_STARTDOC` record points to a `DOCINFO` structure, which contains information for the `StartDoc()` function that initiates a print job. An attacker can manipulate the `lpszDocName`, `lpszOutput`, and `lpszDatatype` members of the `DOCINFO` structure to read arbitrary memory locations.

If an attacker successfully exploits these arbitrary read primitives, they could reveal sensitive information, such as memory layout. Additionally, these primitives may serve as an oracle that an attacker could use to scan the memory address space of the process, potentially circumventing memory protections, such as address space layout randomization (ASLR).

5.3.7 CVE-2022-38006

The seventh vulnerability occurs in the `MfEnumState::OutputDIB()` function when the Windows graphics component processes metafile containing an `EMR_STRETCHDIBITS` record with a specially crafted `BITMAPINFOHEADER` structure (Selján, 2022c).

The `EmfEnumState::ProcessRecord()` function calls the appropriate handler function for each record based on its type. For `EMR_STRETCHDIBITS` records, this handler is the `EmfEnumState::StretchDIBits()` function. This function is responsible for copying the color data for a specified rectangle of pixels to the designated destination rectangle and subsequently calls the `MfEnumState::OutputDIB()` function.

```
void MfEnumState::OutputDIB(...) {
...SNIP...
          if (!GetDibNumPalEntries(
            1,                        // flag?
            biSize,                   // 0x28
            Src->biBitCount,          // 0x20
```

```
            Src->biCompression,      // 0x0
            Src->biClrUsed,          // 0x22222222
            &palEntries) )
            goto FAIL;
          if (Src & 3 || iUsage) { // DIB_PAL_COLORS
            palSize = sizeof(WORD) * (iUsage == DIB_RGB_COLORS) + 2;
            // 0x03 * 0x02 = 0x06
            if (UIntMult(palEntries, palSize, &dwBytes) < 0 )
              goto FAIL;
            // 0x03 * 0x04 = 0x0c
            if (UIntMult(palEntries, sizeof(DWORD), &palSize) < 0 )
              goto FAIL;
            // 0x06 + 0x28 = 0x2e
            if (SizeTAdd(dwBytes, Src->biSize, &Size) < 0 )
              goto FAIL;
            // 0x0c + 0x28 = 0x34
            if (SizeTAdd(palSize, Src->biSize, &dwBytes) < 0 )
              goto FAIL;
            Dst = GpMallocEx(dwBytes, 8); // 0x34
            if (!Dst)
              goto FAIL;
            memcpy(Dst, Src, Size); // Crash here!
          }
...SNIP...
}
```

Listing 20: Pseudocode of the `MfEnumState::OutputDIB()` function

The vulnerability stems from the `GetDibNumPalEntries()` function, which is called multiple times during the process, and the initial flag parameter influences its return value.

```
int GetDibNumPalEntries(int flag, uint biSize, int biBitCount, uint biCompression,
    uint biClrUsed, uint *palEntries) {
  nSize = biSize;
  // This will override the biCompression parameter.
  if ((biBitCount == 0x10 || biBitCount == 0x20) && flag)
    biCompression = BI_BITFIELDS;
...SNIP...
  palEntries = 0;
  if (biCompression) {
    if (biCompression--) {
      if (biCompression--) {
        if (biCompression != 1 || biBitCount != 0x10 && biBitCount != 0x20)
          return 0;
        // The color table consists of three DWORD color masks.
        biClrUsed = nSize > 0x28 ? 0 : 3;
        palEntries = biClrUsed;
        goto LABEL_11;
      }
...SNIP...
      return 0;
    }
  }
  if (biBitCount == 1 || biBitCount == 4 || biBitCount == 8)
    goto LABEL_18;
  if (biBitCount != 0x18 && biBitCount != 0x10 && biBitCount != 0x20)
    return 0;
```

```
LABEL_11:
  if (biClrUsed && biClrUsed <= palEntries)
    palEntries = biClrUsed;
  if (palEntries > 0x100)
    palEntries = 256;
  result = 1;
  return result;
}
```

Listing 21: Pseudocode of the `GetDibNumPalEntries()` function

If the flag is set to 1, the function assumes or enforces `BI_BITFIELDS` compression, which indicates that three color masks should be present, and returns a value of 3.

```
EmfEnumState::StretchDIBits()
├── GetBitmapFromRecordEx()
│     └── GetDibNumPalEntries(0, ...) -> 0
└── MfEnumState::OutputDIB()
      └── GetDibNumPalEntries(0, ...)    -> 0
      └── GetDibNumPalEntries(1, ...)    -> 3
      └── GetDibNumPalEntries(1, ...)    -> 3
```

The root cause of the issue lies in the enforcement of `BI_BITFIELDS` compression on a `BI_RGB` bitmap. The `GetBitmapFromRecordEx()` function extracts only `0x28` bytes of the bitmap from the record. In contrast, the `MfEnumState::OutputDIB()` function incorrectly assumes that additional color masks or 2-byte indices accompany the bitmap.

6. Results and conclusion

6.1 Summary

This section provides a comprehensive summary of the research by outlining its objectives and key findings, while also describing its outputs and limitations, which are discussed in detail in subsequent sections.

The primary objective of this dissertation was to propose a solution to a real-world issue in security vulnerability research, drawing inspiration from the literature on the potential applications of ML methods in cybersecurity. One of my key objectives was to develop a prototype of `SeedClass`, an ML-based tool designed to leverage the extensive knowledge gained from long-term fuzzing campaigns through a NN model for binary classification.

Data is the fundamental requirement of any ML project. To obtain the necessary data for training the classifier, I relied on files scraped from the Internet, as well as interesting mutated testcases produced by the `WinAFL` fuzzer or testcases generated using the `FormatFuzzer` tool. For the latter, I developed an enhanced binary template for EMF files, achieving approximately 80% coverage of the graphic operations exposed via metafile records.

However, the research did not end with the development of the prototype for the ML model. A key aspect of the study was the practical application of this theoretical concept, which was realized through the fuzz testing of a selected software component. This case study allowed me to explore further how the approach could be implemented in real-world scenarios.

The case study required the creation of a dedicated fuzzing harness, which serves as an interface between the fuzzer and the targeted software component. Furthermore, I integrated a new functionality into the `WinAFL` fuzzer called the `bug bucket` to streamline the triaging of numerous program crashes. This feature groups crash samples according to the memory address of the instruction that triggered them.

Since fuzz testing aims to identify new security vulnerabilities that manifest themselves as program crashes, the primary evaluation metric was the number of crashes identified by the fuzzer. Previous research has shown that increasing code coverage correlates with an increased number of crashes, making the number of distinct code paths explored within a given timeframe a secondary evaluation criterion for assessing fuzzing effectiveness.

Taking these factors into account, the evaluation looked at test campaigns using different ways to select seed inputs based on two main points: (1) time costs, which refers

to how long it takes to pick and examine the initial inputs, and (2) test efficiency, which compares how many code paths were explored and how many crashes were found with different methods of selecting inputs.

The results of the experiments demonstrate that knowledge from prior tests can be leveraged to enhance the fuzzing process through the use of a binary classification model. In addition, the research uncovered several zero-day vulnerabilities, which Microsoft publicly acknowledged and addressed through security updates. In general, this study has contributed to the development of efficient testing tools for identifying and mitigating security threats through fuzz testing.

6.2 Outputs

This research utilized ML to enhance the fuzz testing methodology, focusing on a case study of the Windows graphics component using the EMF file format. This section details the results of the investigation.

This study introduced an enhanced binary template of the EMF file format for the `010 Editor` to improve the coverage of graphic operations. The created template provides approximately 80% coverage of available metafile records, making it a valuable tool in the overall approach to improve the fuzz testing of the Windows graphics component through metafiles. Security researchers can use the binary template with the `FormatFuzzer` tool to generate input files for ML experiments and fuzz testing as well.

Additionally, this work presented a fuzzing harness designed to retrieve and transmit a metafile to the target system, specifically the Windows graphics component, and it has been demonstrated to be effective for fuzz testing. The harness performs various operations on the input file, such as creating a thumbnail or converting an EMF metafile to WMF format. These operations are likely to trigger various program behaviors, thus increasing the code coverage of the target program. The fuzzing harness is a crucial component of the overall approach, as it ensures that the testing system can reveal genuine errors in the program.

The development of a remote management and monitoring application for fuzzing campaigns was essential to efficiently execute the numerous experiments conducted during the research. I developed `WinAFL Pet` to simplify the monitoring and configuration of multiple fuzzer instances running on remote machines, enhancing both scalability and manageability. This application played a crucial role in ensuring an effective testing process by providing a centralized platform for overseeing multiple fuzzing campaigns.

Since the research aimed to increase the number of identified program crashes, it was necessary to extend the `WinAFL` fuzzer with a functionality that facilitates the evaluation of detected bugs; hence, the dissertation introduced the `bug bucket` feature. By keeping a record of the memory address associated with each crash, this feature allows grouping

samples based on the memory address of the instruction that caused the crash, making it easier to identify unique cases. This feature significantly assisted in the evaluation process.

Furthermore, the dissertation introduced `SeedClass`, a tool that uses a NN for binary classification to select valuable inputs from large datasets. This tool is a prototype of an ML-based approach using `TensorFlow`. This is achieved by training a binary classifier on a set of labeled examples and then using this classifier to predict the value of unlabeled samples. The expected value indicates the file's potential to trigger a program crash, thereby providing an effective way to select valuable seed files from large datasets and improve the overall efficiency of the fuzzing process.

In addition to the artifacts outlined in this chapter, I made the dataset used to train the ML model publicly available, along with a suite of helper scripts for preprocessing the data. During the data collection process, I identified more than 50 distinct samples capable of reliably reproducing bugs and security vulnerabilities, as summarized in the table included in Appendix 6.4. These samples are valuable assets for security researchers studying vulnerabilities within the Windows graphics component. To further support future vulnerability research, I also published relevant software artifacts, including the binary template for EMF metafiles, the testing harness to fuzz the Windows graphics component, and the prototype implementation of the binary classifier. By making these resources available on GitHub, this work promotes transparency and reproducibility in vulnerability research while providing useful tools for fellow security researchers. I hope that sharing these materials will foster further innovation and collaboration in the field.

In conclusion, this work advances vulnerability research while reaffirming the broad potential of applying ML in this domain. The dissertation highlights the real-world impact of the study, as demonstrated by the discovery of seven zero-day vulnerabilities in the Windows graphics component. This discovery underscores the practical viability and real-world implications of the approach, providing a compelling case for the further integration of ML techniques into vulnerability research methodologies.

6.3 Limitations

Like all research projects, the prediction-based seed selection strategy discussed in this dissertation also has limitations. In this section, I outline the constraints of the overall research and the potential limitations associated with the application of the proposed seed selection approach.

The strength of the presented method also serves as its main limitation. The dissertation focused on supervised ML, which means that the prediction-based seed selection strategy relies on accurately labeled examples to train the NN. Consequently, this approach is unsuitable if known vulnerabilities and their corresponding input samples, capable of reproducing the program crash, are unavailable.

Furthermore, to ensure that the method does not hinder the ability of the fuzzer to discover new execution paths, proper preprocessing of the training data is necessary, which can limit the scalability of this approach. The experiments carried out during the research confirmed that the seed selection strategy presented can effectively guide the fuzzer. However, the result significantly depends on the sample files used to train the model.

This project concentrated on optimizing the seed selection in fuzz testing to increase the likelihood of discovering novel program crashes. However, assessing the exploitability of these crashes, whether they constitute security vulnerabilities or simply benign software bugs, falls outside the scope of this research. Notably, exploitability analysis has emerged as a distinct research domain, with prior studies demonstrating that ML methods can be used effectively to assess software exploitability (Suciu et al., 2021; Yan et al., 2017) and predict the probability of a vulnerability being exploited (Bhatt et al., 2021). These findings reinforce the notion that exploitability prediction requires specialized methodologies separate from the seed selection strategies examined in this work. Consequently, the terms *bug* and *vulnerability* were used interchangeably throughout the dissertation.

A critical consideration in designing the seed selection method was to avoid any negative impact on the fuzzer, whether through additional overhead or the introduction of errors, especially given the measurable performance penalty already imposed by dynamic instrumentation. Consequently, the prototype implemented can be utilized as a separate step at the beginning of the fuzz testing process. However, this approach also leaves the question of whether closer integration with any particular fuzzer is feasible. In my opinion, one of the most promising integration possibilities would be to filter newly discovered interesting testcases and use the sample files identified in the model predictions as a seed corpus when restarting a fuzzing campaign.

From an evaluative perspective, this dissertation aimed to demonstrate the applicability of a binary classification model for seed selection. As Klees et al. (2018) pointed out, relying on isolated evaluations of seed selection strategies and fuzz testing methodologies can lead to misleading conclusions and hinder meaningful comparisons between studies. Thus, a more comprehensive assessment of a binary classifier should encompass a broader array of seed selection strategies and include fuzz testing across multiple file formats.

The data encoding mechanism, which determines the features that carry information for the model, is another critically important aspect of implementation, along with compiling the dataset required for training the model. M. Wang and Robinson (2023) have proposed another approach that could be a potential future research direction. In their recent work, the authors introduce a process that converts input data into Unicode bytes, transforms them into a sequence of integer values, and then normalizes them. Padding the remaining entries with null bytes to meet input size requirements without altering the input itself can potentially improve the model's performance. Figure 23 illustrates the procedure. However, further research is needed to evaluate the effectiveness of this approach.

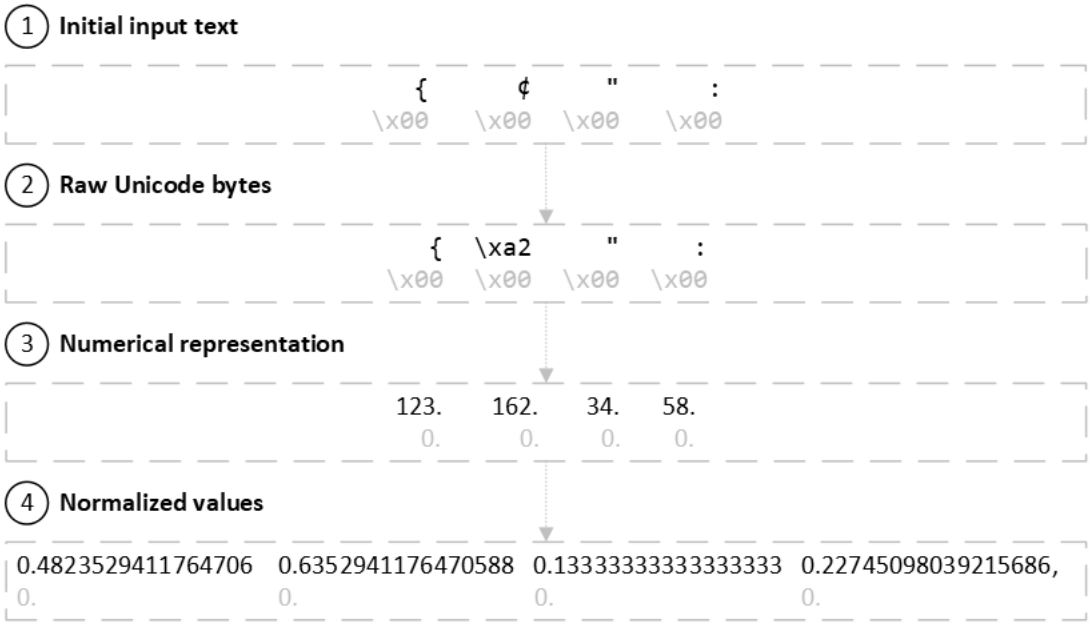


Figure 23: Alternative input encoding process (based on M. Wang and Robinson, 2023)

Finally, although fuzz testing the Windows graphics component served only as a case study and was not the primary focus of the dissertation, the EMF+ file format is more complex and includes several previously unexplored metafile records, which represent an additional attack surface. The application of the outlined approach to the testing of the EMF+ format presents a promising opportunity for future research.

6.4 Conclusion

This dissertation bridges the gap between ML and security vulnerability research by introducing an ML-powered seed selection strategy for directed fuzz testing to prioritize input samples exhibiting characteristics of known crash patterns and demonstrating how the output of prior fuzzing campaigns can serve as training data for a NN. The resulting binary classification model, `SeedClass`, predicts which input samples can guide the fuzzer toward security-sensitive code paths that have been previously affected by known security vulnerabilities.

I have developed several software artifacts to support this core contribution. I implemented a fuzzing harness specifically for the Windows graphics component; however, its main functionality can also be reused for other targets. Additionally, I developed a remote management tool and modified `WinAFL` to allow large-scale experiments and support crash triage. I also created an improved binary template to enable richer input generation for ML and fuzzing. I published the complete training dataset, data preprocessing scripts, and the `SeedClass` prototype on GitHub to facilitate further research.

Experimental results on the Windows graphics component confirm that the proposed

prediction-based seed selection strategy consistently directs the fuzzer toward code paths that trigger known program failures. Compared to conventional seed selection methods, prediction-driven fuzzing campaigns were faster in discovering the first crash and identified more bugs within the same time frame. In particular, this approach led to the disclosure of seven previously unknown vulnerabilities, demonstrating its practical impact.

Despite these successes, the method's reliance on supervised learning introduces certain constraints. The classifier's performance depends on the availability and quality of labeled crash samples, and the current workflow remains an offline, one-shot preprocessing step. Broader validation across diverse file formats, tighter integration into the fuzzing loop, and exploration of alternative encoding schemes, such as Unicode-based feature representations, are necessary to generalize the approach.

Looking ahead, integrating `SeedClass` into an online feedback loop could enable continuous model refinement based on newly discovered paths. Extending the framework to semi-supervised or unsupervised learning methods may reduce dependence on labeled training samples. By combining ML with vulnerability discovery, this dissertation contributes to the development of more adaptive, data-driven fuzzing frameworks and provides a basis for further advancements in security vulnerability research.

Bibliography

Abada, R., Abubakar, A. M., & Bilal, M. T. (2022). An Overview on Deep Leaning Application of Big Data. *Mesopotamian Journal of Big Data*, *2022*, 31–35. https://doi.org/10.58496/mjbd/2022/004 (cit. on p. 35).

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2025, March 13). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* (Version 2.19). https://www.tensorflow.org/ (cit. on p. 75).

Ables, J., Childers, N., Anderson, W., Mittal, S., Rahimi, S., Banicescu, I., & Seale, M. (2024). Eclectic Rule Extraction for Explainability of Deep Neural Network based Intrusion Detection Systems. https://doi.org/10.48550/arXiv.2401.10207 (cit. on p. 37).

Abraham, T., & de Vel, O. (2017, October). A Review of Machine Learning in Software Vulnerability Research. https://www.dst.defence.gov.au/publication/review-machine-learning-software-vulnerability-research (cit. on p. 8).

Acemoglu, D., & Restrepo, P. (2018, January). *Artificial Intelligence, Automation and Work* (Working Paper No. 24196). National Bureau of Economic Research. https://doi.org/10.3386/w24196 (cit. on p. 3).

Aggarwal, C. C., & Reddy, C. K. (Eds.). (2018). *An Introduction to Cluster Analysis*. Chapman & Hall/CRC. https://www.taylorfrancis.com/chapters/edit/10.1201/9781315373515-1/introduction-cluster-analysis-charu-aggarwal (cit. on p. 40).

Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining Association Rules Between Sets of Items in Large Databases. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 207–216. https://doi.org/10.1145/170035.170072 (cit. on p. 41).

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A. I., et al. (1996). Fast Discovery of Association Rules. *Advances in Knowledge Discovery and Data Mining*, *12*(1), 307–328. http://www.cs.bme.hu/~marti/adatbanya/apriori_hashtree.pdf (cit. on p. 41).

Alpaydin, E. (2020). *Introduction to Machine Learning*. MIT press. https://mitpress.mit.edu/9780262043793/introduction-to-machine-learning/ (cit. on p. 37).

Alrawi, O., Wong, M. Y., Avgetidis, A., Valakuzhy, K., Adjibi, B. V., Karakatsanis, K., Ahamad, M., Blough, D., Monrose, F., & Antonakakis, M. (2024). SoK: An Essential Guide For Using Malware Sandboxes In Security Applications: Challenges, Pitfalls, and Lessons Learned. https://arxiv.org/abs/2403.16304 (cit. on p. 18).

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. https://doi.org/10.48550/arXiv.1701.07875 (cit. on p. 43).

Arya, A., Chang, O., Moroz, M., Barbella, M., & Metzman, J. (2019, February). *Open sourcing ClusterFuzz*. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html (cit. on p. 7).

Auburn, L. (2023, November). *Audio deepfake detective developing new sleuthing techniques*. Retrieved June 22, 2025, from https://www.rochester.edu/newscenter/audio-deepfake-detective-developing-new-sleuthing-techniques-573482/

Austin, A., & Williams, L. (2011). One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. *2011 International Symposium on Empirical Software Engineering and Measurement*, 97–106. https://doi.org/10.1109/ESEM.2011.18 (cit. on p. 21).

Ayesha, S., Hanif, M. K., & Talib, R. (2020). Overview and Comparative Study of Dimensionality Reduction Techniques for High Dimensional Data. *Information Fusion*, *59*, 44–58. https://doi.org/10.1016/j.inffus.2020.01.005 (cit. on p. 43).

Babuta, A., Oswald, M., & Janjeva, A. (2020, April). Artificial Intelligence and UK National Security: Policy Considerations. https://rusi.org/explore-our-research/publications/occasional-papers/artificial-intelligence-and-uk-national-security-policy-considerations (cit. on p. 15).

Baka, B. (2017). *Python Data Structures and Algorithms*. Packt Publishing Ltd. https://www.oreilly.com/library/view/python-data-structures/9781786467355/ (cit. on p. 79).

Bakhvalov, D. (2019). *Enhance performance analysis with Intel Processor Trace*. https://easyperf.net/blog/2019/08/23/Intel-Processor-Trace (cit. on p. 66).

Bala, V., Duesterwald, E., & Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 1–12. https://doi.org/10.1145/349299.349303 (cit. on p. 65).

Banerjee, C., Mukherjee, T., & Pasiliao, E. (2019). An Empirical Study on Generalizations of the ReLU Activation Function. *Proceedings of the 2019 ACM Southeast Conference*, 164–167. https://doi.org/10.1145/3299815.3314450 (cit. on p. 80).

Barriga, J. J., & Yoo, S. G. (2017). Malware Detection and Evasion with Machine Learning Techniques: A Survey. *International Journal of Applied Engineering Research*, *12*(18), 7207–7214. https://api.semanticscholar.org/CorpusID:31201315 (cit. on p. 18).

Bhatt, N., Anand, A., & Yadavalli, V. S. (2021). Exploitability prediction of software vulnerabilities. *Quality and Reliability Engineering International*, *37*. https://doi.org/10.1002/qre.2754 (cit. on p. 106).

Binosi, L., Rullo, L., Polino, M., Carminati, M., Zanero, S., et al. (2023). Rainfuzz: Reinforcement Learning Driven Heat-Maps for Boosting Coverage-Guided Fuzzing. *Proceedings of the 12th International Conference on Pattern Recognition Applications and Methods - ICPRAM*, 39–50. https://doi.org/10.5220/0011625300003411 (cit. on p. 22).

Black Duck Editorial Staff. (2017). *Navigating Responsible Vulnerability Disclosure Best Practices*. Retrieved June 22, 2025, from https://www.blackduck.com/blog/responsible-vulnerability-disclosure-best-practices.html

Böhme, M., Pham, V.-T., Nguyen, M.-D., & Roychoudhury, A. (2017). Directed Greybox Fuzzing. *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2329–2344. https://doi.org/10.1145/3133956.3134020 (cit. on p. 31).

Böhme, M., Pham, V.-T., & Roychoudhury, A. (2019). Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, *45*(5), 489–506. https://doi.org/10.1109/TSE.2017.2785841 (cit. on pp. 31, 52).

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., … Amodei, D. (2020, May). Language Models are Few-Shot Learners. https://doi.org/10.48550/arXiv.2005.14165 (cit. on p. 41).

Bruening, D., Garnett, T., & Amarasinghe, S. (2003). An Infrastructure for Adaptive Dynamic Optimization. *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 265–275. https://doi.org/10.1109/CGO.2003.1191551 (cit. on p. 65).

Bruening, D. (2004, September). *Efficient, transparent, and comprehensive runtime code manipulation* [Doctoral dissertation, Massachusetts Institute of Technology]. https://www.burningcutlery.com/derek/phd.html (cit. on pp. 65, 66).

Bruening, D., Kleckner, R., & Zhao, Q. (2022, February 14). *DynamoRIO* (Version 9.0.1). https://github.com/DynamoRIO/dynamorio/releases/tag/release_9.0.1 (cit. on p. 66).

Bruneau, G. (2001). The History and Evolution of Intrusion Detection. *SANS Institute*. https://www.sans.org/white-papers/344/ (cit. on p. 19).

Brunner, C. (2020). *Intrusion detection by machine learning* [Doctoral dissertation, Corvinus University of Budapest]. https://doi.org/10.14267/phd.2020026 (cit. on p. 20).

Camargo, M., Dumas, M., & González-Rojas, O. (2020). Automated discovery of business process simulation models from event logs. *Decision Support Systems*, *134*, 113284. https://doi.org/10.1016/j.dss.2020.113284 (cit. on p. 4).

Canakci, S., Matyunin, N., Graffi, K., Joshi, A., & Egele, M. (2022). TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers. *Proceedings of the 2022 ACM on*

Asia Conference on Computer and Communications Security, 561–573. https://doi.org/10.1145/3488932.3501276 (cit. on p. 53).

Cassidy, D., Wales, C., Turetzky, K., Buck, A., Kshirsagar, D., & Mabrey, H. (2022). *Foundations of Assessing Harm*. Retrieved June 20, 2025, from https://learn.microsoft.com/en-us/azure/architecture/guide/responsible-innovation/harms-modeling/ (cit. on p. 3).

Ceschin, F., Botacin, M., Gomes, H. M., Oliveira, L. S., & Gregio, A. (2020). Shallow Security: on the Creation of Adversarial Variants to Evade Machine Learning-Based Malware Detectors. *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*. https://doi.org/10.1145/3375894.3375898 (cit. on p. 19).

Chapman, A. D. (2023). *Neural networks*. The Autodidact's Toolkit. https://www.amazon.com/Neural-Networks-Autodidacts-Toolkit-Chapman-ebook/dp/B0CSPBNXGV (cit. on p. 36).

Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., & Liu, W. (2018). A Systematic Review of Fuzzing Techniques. *Computers & Security*, *75*, 118–137. https://doi.org/10.1016/j.cose.2018.02.002 (cit. on p. 27).

Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., & Liu, Y. (2018). Hawkeye: Towards a Desired Directed Grey-box Fuzzer. *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2095–2108. https://doi.org/10.1145/3243734.3243849 (cit. on p. 31).

Chen, J., Wu, K., & Xiu, D. (2025). DUE: A Deep Learning Framework and Library for Modeling Unknown Equations. https://arxiv.org/abs/2504.10373 (cit. on p. 35).

Cheng, L., Zhang, Y., Zhang, Y., Wu, C., Li, Z., Fu, Y., & Li, H. (2019). Optimizing Seed Inputs in Fuzzing with Machine Learning. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 244–245. https://doi.org/10.1109/ICSE-Companion.2019.00096 (cit. on p. 8).

Chilton, J. (2023, April). *The New Risks ChatGPT Poses to Cybersecurity*. Retrieved June 22, 2025, from https://hbr.org/2023/04/the-new-risks-chatgpt-poses-to-cybersecurity (cit. on p. 16).

Chollet, F. (2015, March). *Keras* (Version 3.10.0). https://keras.io (cit. on p. 80).

Chow, S., Gu, Y., Johnson, H., & Zakharov, V. A. (2001). An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In G. I. Davida (Ed.), *Information Security* (pp. 144–155, Vol. 2200). Springer. https://doi.org/10.1007/3-540-45439-X_10 (cit. on p. 65).

Clause, J., Li, W., & Orso, A. (2007). Dytan: A Generic Dynamic Taint Analysis Framework. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 196–206. https://doi.org/10.1145/1273463.1273490 (cit. on p. 46).

Cohen, S., Bitton, R., & Nassi, B. (2025). Here Comes The AI Worm: Unleashing Zero-click Worms that Target GenAI-Powered Applications. https://doi.org/10.48550/arXiv.2403.02817 (cit. on p. 19).

Conn, A. (2015, November). Benefits and Risks of Artificial Intelligence. Retrieved June 22, 2025, from https://futureoflife.org/ai/benefits-risks-of-artificial-intelligence (cit. on p. 2).

Cosentino, V., Luis, J., & Cabot, J. (2016). Findings from GitHub: methods, datasets and limitations. *Proceedings of the 13th International Conference on Mining Software Repositories*, 137–141. https://doi.org/10.1145/2901739.2901776 (cit. on p. 73).

CrowdStrike. (2019). *Red Team / Blue Team Exercise*. Retrieved May 16, 2024, from https://www.crowdstrike.com/services/prepare/red-team-blue-team-exercise/ (cit. on p. 16).

Csáki, C. (2022). Social Sustainability in Medicine: The Role of Artificial Intelligence in Future Doctor–Patient Communication. A Methodological Experiment. *Acta Universitatis Sapientiae, Communicatio*, (9), 90–107. https://doi.org/10.2478/auscom-2022-0007 (cit. on p. 2).

Da San Martino, G. (2009). *Kernel Methods for Tree Structured Data* [Doctoral dissertation, University of Bologna]. https://amsdottorato.unibo.it/id/eprint/1400/

Davis, J., & Goadrich, M. (2006). The relationship between Precision-Recall and ROC curves. *Proceedings of the 23rd International Conference on Machine Learning*, 233–240. https://doi.org/10.1145/1143844.1143874 (cit. on p. 83).

Dela Cruz, G., & Rubi, J. (2022). The Effects of AI in Various Spheres of Life. *Technoarete Transactions on Advances in Computer Applications*, *1*, 1–8. https://www.researchgate.net/publication/373338553_The_Effects_of_AI_in_Various_Spheres_of_Life (cit. on p. 3).

Dempsey, K. L., Chawla, N. S., Johnson, L. A., Johnston, R., Jones, A. C., Orebaugh, A. D., Scholl, M. A., & Stine, K. M. (2011). *SP 800-137. Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations* (tech. rep.). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-137 (cit. on p. 3).

Dinh, T., Hauchi, W., Lisik, D., Koren, M., Tran, D., Yu, P. S., & Torres-Sospedra, J. (2025). Data clustering: an essential technique in data science. https://arxiv.org/abs/2412.18760 (cit. on p. 40).

Drake, J. J., & Coley, S. C. (2016). Vulnerabilities 101: How to Launch or Improve Your Vulnerability Research Game. https://doi.org/10.5446/36258 (cit. on p. 7).

Dreiseitl, S., & Ohno-Machado, L. (2002). Logistic regression and artificial neural network classification models: a methodology review. *Journal of Biomedical Informatics*, *35*(5), 352–359. https://doi.org/10.1016/S1532-0464(03)00034-0 (cit. on pp. 37, 38).

Dutra, R., Gopinath, R., & Zeller, A. (2023, September). FormatFuzzer: Effective Fuzzing of Binary File Formats. https://doi.org/10.48550/arXiv.2109.11277 (cit. on pp. 34, 63, 64).

Dynamic Instrumentation Tool Platform (Version 9.0.1). (2022, February 14). https://dynamorio.org

Dzeroski, S., Dehaspe, L., Ruck, B. M., & Walle, W. (1970). Classification Of River Water Quality Using Machine Learning. *WIT Transactions on Ecology and the Environment*, *5*. https://www.witpress.com/elibrary/wit-transactions-on-ecology-and-the-environment/5/11448 (cit. on p. 35).

Eimer, T., Lindauer, M., & Raileanu, R. (2023). Hyperparameters in reinforcement learning and how to tune them. *International conference on machine learning*, 9104–9149. https://proceedings.mlr.press/v202/eimer23a (cit. on p. 44).

Elharrouss, O., Mahmood, Y., Bechqito, Y., Serhani, M. A., Badidi, E., Riffi, J., & Tairi, H. (2025). Loss Functions in Deep Learning: A Comprehensive Review. https://arxiv.org/abs/2504.04242 (cit. on p. 83).

Emmert-Streib, F., Yang, Z., Feng, H., Tripathi, S., & Dehmer, M. (2020). An Introductory Review of Deep Learning for Prediction Models with Big Data. *Frontiers in Artificial Intelligence*, *3*, 4. https://doi.org/10.3389/frai.2020.00004 (cit. on p. 2).

Evans, C., Moore, M., & Ormandy, T. (2011, August). *Fuzzing at scale*. https://security.googleblog.com/2011/08/fuzzing-at-scale.html (cit. on p. 1).

Fahrmeir, L., Kneib, T., Lang, S., & Marx, B. D. (2022). *Regression*. Springer. https://doi.org/10.1007/978-3-662-63882-8 (cit. on p. 38).

Fan, R., & Chang, Y. (2018). Machine Learning for Black-Box Fuzzing of Network Protocols. In S. Qing, C. Mitchell, L. Chen, & D. Liu (Eds.), *Information and Communications Security* (pp. 621–632). Springer International Publishing. https://doi.org/10.1007/978-3-319-89500-0_53 (cit. on p. 30).

Fang, R., Bindu, R., Gupta, A., & Kang, D. (2024, April). LLM Agents can Autonomously Exploit One-day Vulnerabilities. https://doi.org/10.48550/arXiv.2404.08144

Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From Data Mining to Knowledge Discovery in Databases. *AI magazine*, *17*(3), 37. https://doi.org/10.1609/aimag.v17i3.1230 (cit. on p. 3).

Fell, J. (2017). A review of fuzzing tools and methods. *PenTest Magazine*. https://pentestmag.com/download/pentest-intelligent-fuzzing-techniques/ (cit. on pp. 26, 27).

Feuz, K. D., & Cook, D. J. (2015). Transfer Learning across Feature-Rich Heterogeneous Feature Spaces via Feature-Space Remapping (FSR). *ACM Trans Intell Syst Technol*, *6*(1), 3. https://doi.org/10.1145/2629528 (cit. on p. 36).

Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M. (2020). AFL++ : Combining Incremental Steps of Fuzzing Research. *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi

Flach, P. (2012). Machine Learning. In *Machine Learning: The Art and Science of Algorithms that Make Sense of Data* (pp. 367–382). Cambridge University Press. https://doi.org/10.1017/CBO9780511973000.017

Foreman, P. (2019, May). *Vulnerability Management*. Auerbach Publications. https://doi.org/10.1201/9780429289651 (cit. on p. 20).

Fratric, I. (2023, September 4). *WinAFL: A fork of AFL for fuzzing Windows binaries* (Version 1.17). https://github.com/googleprojectzero/winafl (cit. on p. 68).

Ganesh, V., Leek, T., & Rinard, M. (2009). Taint-based Directed Whitebox Fuzzing. *2009 IEEE 31st International Conference on Software Engineering*, 474–484. https://doi.org/10.1109/ICSE.2009.5070546 (cit. on p. 30).

Garnett, T. (2003). *Dynamic optimization if IA-32 applications under DynamoRIO* [Doctoral dissertation, Massachusetts Institute of Technology]. https://dspace.mit.edu/bitstream/handle/1721.1/87411/53471363-MIT.pdf (cit. on p. 65).

Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., & Rieck, K. (2015). Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST* (pp. 330–347, Vol. 164). Springer-Verlag. https://doi.org/10.1007/978-3-319-28865-9_18 (cit. on p. 30).

GCC Team. (2021, January). *Basic Blocks*. https://gcc.gnu.org/onlinedocs/gcc-4.6.2/gccint/Basic-Blocks.html (cit. on p. 65).

Géron, A. (2022, October). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition*. O'Reilly Media. https://www.oreilly.com/library/view/hands-on-machine-learning/9781098125967/ (cit. on p. 59).

Ghasemi, M., Moosavi, A. H., & Ebrahimi, D. (2025). A Comprehensive Survey of Reinforcement Learning: From Algorithms to Practical Challenges. https://doi.org/10.48550/arXiv.2411.18892 (cit. on p. 44).

Gipiškis, R., Joaquin, A. S., Chin, Z. S., Regenfuß, A., Gil, A., & Holtman, K. (2024, October). Risk Sources and Risk Management Measures in Support of Standards for General-Purpose AI Systems. https://doi.org/10.48550/arXiv.2410.23472

Girhepuje, S., Verma, A., & Raina, G. (2024, September). A Survey on Offensive AI Within Cybersecurity. https://doi.org/10.48550/arXiv.2410.03566 (cit. on pp. 5, 15).

GitHub. (2008). GitHub. https://github.com/ (cit. on p. 73).

GitLab. (2011). GitLab. https://gitlab.com/ (cit. on p. 73).

Godefroid, P., Kiezun, A., & Levin, M. Y. (2007, November). *Grammar-based Whitebox Fuzzing* (tech. rep. No. MSR-TR-2007-154). Massachusetts Institute of Technology. https://www.microsoft.com/en-us/research/publication/grammar-based-whitebox-fuzzing/ (cit. on p. 46).

Godefroid, P., Kiezun, A., & Levin, M. Y. (2008). Grammar-based Whitebox Fuzzing. *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 206–215. https://doi.org/10.1145/1375581.1375607 (cit. on p. 46).

Godefroid, P., Levin, M. Y., & Molnar, D. (2008, November). Automated Whitebox Fuzz Testing. https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/ (cit. on p. 30).

Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, *55*(3), 40–44. https://doi.org/10.1145/2093548.2093564 (cit. on pp. 7, 30).

Godefroid, P., Peleg, H., & Singh, R. (2017). Learn&Fuzz: Machine learning for input fuzzing. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 50–59. https://doi.org/10.1109/ASE.2017.8115618 (cit. on p. 49).

Goertzel, B. (2014). Artificial General Intelligence: Concept, State of the Art, and Future Prospects. *Journal of Artificial General Intelligence*, *5*(1), 1–48. https://doi.org/10.2478/jagi-2014-0001 (cit. on p. 2).

Goncalves, J. d. S., Manduchi, L., Vandenhirtz, M., & Vogt, J. E. (2024). Structured Generations: Using Hierarchical Clusters to guide Diffusion Models. https://doi.org/10.48550/arXiv.2407.06124

Goodfellow, I., Bengio, Y., & Courville, A. (2014). *GAN (Generative Adversarial Network) | Envisioning Vocab*. Retrieved June 22, 2025, from https://www.envisioning.io/vocab/gan-generative-adversarial-network (cit. on p. 42).

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. *Advances in Neural Information Processing Systems*, *3*. https://doi.org/10.1007/978-3-658-40442-0_9 (cit. on p. 42).

Google. (2018). *Syzygy Transformation Toolchain*. https://github.com/google/syzygy (cit. on p. 66).

Grakovski, A., Krivchenkov, A., & Misnevs, B. (2022). Feature Selection Method for ML/DL Classification of Network Attacks in Digital Forensics. *Transport and Telecommunication Journal*, *23*(2), 131–141. https://doi.org/10.2478/ttj-2022-0011 (cit. on p. 20).

Grammarly. (2024, November). *Regression in Machine Learning: What It Is and How It Works*. https://www.grammarly.com/blog/ai/what-is-regression/

Gregersen, E. (2025). *ChatGPT*. https://www.britannica.com/technology/ChatGPT (cit. on p. 16).

Group, A. S. (2023). *Now is the Time to Integrate AI/ML into Your Organization's Cybersecurity Plan*. https://www.linkedin.com/pulse/now-time-integrate-aiml-your-organizations-cybersecurity (cit. on p. 15).

Guembe, B., Azeta, A., Misra, S., Osamor, V. C., Fernandez-Sanz, L., & Pospelova, V. (2022). The Emerging Threat of AI-driven Cyber Attacks: A Review. *Applied Artificial Intelligence*, *36*(1), 2037254. https://doi.org/10.1080/08839514.2022.2037254 (cit. on p. 15).

Hamadah, S., & Aqel, D. (2020). Cybersecurity Becomes Smart Using Artificial Intelligent and Machine Learning Approaches: An Overview. *ICIC Express Letters, Part B: Applications*, *11*(12), 1115–1123. https://doi.org/10.24507/icicelb.11.12.1115 (cit. on pp. 6, 16).

Han, J., Pei, J., & Tong, H. (2012). *Data Mining: Concepts and Techniques*. Elsevier. https://doi.org/10.1016/C2009-0-61819-5 (cit. on p. 35).

Hansen, T. B. S., & Jensen, M. T. (2023). *Buzzy: An Unguided Smart-Strategy Generation-Based Blackbox Fuzzer for eBPF Technologies* [Master's thesis, Aalborg University]. https://vbn.aau.dk/ws/files/538309795/buzzy_cs_23_ds_10_05_master.pdf (cit. on p. 33).

Hardik, S. (2021, June). *Analyzing CVE-2021-1665 – Remote Code Execution Vulnerability in Windows GDI+*. Retrieved June 22, 2025, from https://www.mcafee.com/blogs/other-blogs/mcafee-labs/analyzing-cve-2021-1665-remote-code-execution-vulnerability-in-windows-gdi/ (cit. on p. 62).

Hastie, T., Tibshirani, R., & Friedman, J. (2009). Unsupervised Learning. In *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (pp. 485–585). Springer New York. https://doi.org/10.1007/978-0-387-84858-7_14 (cit. on p. 39).

Hayes, L., Gunadi, H., Herrera, A., Milford, J., Magrath, S., Sebastian, M., Norrish, M., & Hosking, A. L. (2019). MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation. *arXiv preprint*. https://doi.org/10.48550/arXiv.1905.13055 (cit. on p. 53).

Herrera, A., Gunadi, H., Hayes, L., Magrath, S., Friedlander, F., Sebastian, M., Norrish, M., & Hosking, A. L. (2020). Corpus Distillation for Effective Fuzzing: A Comparative Evaluation. *arXiv preprint*. https://doi.org/10.48550/arXiv.1905.13055 (cit. on p. 53).

Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., & Hosking, A. L. (2021). Seed selection for successful fuzzing. *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 230–243. https://doi.org/10.1145/3460319.3464795 (cit. on pp. 45, 53).

Hodován, R., & Kiss, Á. (2018). Fuzzinator: An Open-Source Modular Random Testing Framework. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 416–421. https://doi.org/10.1109/icst.2018.00050

Holm, H. (2014). Signature Based Intrusion Detection for Zero-Day Attacks: (Not) A Closed Chapter? *2014 47th Hawaii International Conference on System Sciences*, 4895–4904. https://doi.org/10.1109/HICSS.2014.600 (cit. on p. 20).

Householder, A., & Foote, J. M. (2018, June). Probability-Based Parameter Selection for Black-Box Fuzz Testing. https://doi.org/10.1184/R1/6582245.v1

Huiras, W., Ong, I., Ziegler, J., & Labs, T. S. (2022, June). *Edge of the Art in Vulnerability Research Version 6* (tech. rep.). Two Six Labs. https://apps.dtic.mil/sti/citations/AD1172601 (cit. on p. 7).

Hutchins, E. M., Cloppert, M. J., Amin, R. M., et al. (2011). Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. *Leading Issues in Information Warfare & Security Research*, 80. https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/LM-White-Paper-Intel-Driven-Defense.pdf (cit. on p. 16).

Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., & Palomba, F. (2023). The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering*, *49*(1), 44–63. https://doi.org/10.1109/TSE.2022.3140868 (cit. on p. 21).

Ioualalen, A., & Martel, M. (2019). Neural Network Precision Tuning. In D. Parker & V. Wolf (Eds.), *Quantitative Evaluation of Systems* (pp. 129–143). Springer. https://doi.org/10.1007/978-3-030-30281-8_8 (cit. on p. 83).

Irfan, M. M., Ali, S., Yaqoob, I., & Zafar, N. (2021). Towards Deep Learning: A Review on Adversarial Attacks. *2021 International Conference on Artificial Intelligence (ICAI)*, 91–96. https://doi.org/10.1109/ICAI52203.2021.9445247

Iturbe, E., Rios, E., & Toledo, N. (2023). Towards trustworthy Artificial Intelligence: Security risk assessment methodology for Artificial Intelligence systems. *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 291–297. https://doi.org/10.1109/CloudCom59040.2023.00054

Ivanov, A. (2016, October). *Windows Zero-Day Exploit Used in Targeted Attacks by FruityArmor APT*. Retrieved June 22, 2025, from https://securelist.com/windows-zero-day-exploit-used-in-targeted-attacks-by-fruityarmor-apt/76396/ (cit. on p. 62).

Izquierdo, L. R., & Izquierdo, S. S. (2012). Reinforcement Learning. In N. M. Seel (Ed.), *Encyclopedia of the Sciences of Learning* (pp. 2796–2799). Springer. https://doi.org/10.1007/978-1-4419-1428-6_567 (cit. on p. 44).

Jajal, T. D. (2018). Distinguishing Between Narrow AI, General AI and Super AI. *Medium*. https://medium.com/mapping-out-2050/distinguishing-between-narrow-ai-general-ai-and-super-ai-a4bc44172e22 (cit. on p. 3).

James, G., Witten, D., Hastie, T., Tibshirani, R., et al. (2013). *An Introduction to Statistical Learning* (Vol. 112). Springer. https://www.statlearning.com/ (cit. on p. 39).

Jaquet-Chiffelle, D.-O., & Loi, M. (2020). Ethical and Unethical Hacking. In M. Christen, B. Gordijn, & M. Loi (Eds.), *The Ethics of Cybersecurity* (pp. 179–204). Springer International Publishing. https://doi.org/10.1007/978-3-030-29053-5_9 (cit. on p. 5).

Jarek, S. (2020). *Machine Learning Security Evasion Competition 2020 Invites Researchers to Defend and Attack*. Retrieved June 22, 2025, from https://msrc-blog.microsoft.com/2020/06/01/machine-learning-security-evasion-competition-2020-invites-researchers-to-defend-and-attack/ (cit. on p. 19).

Jimmy, F. (2021). Emerging Threats: The Latest Cybersecurity Risks and the Role of Artificial Intelligence in Enhancing Cybersecurity Defenses. *Valley International Journal Digital Library*, 564–574. https://doi.org/10.18535/ijsrm/v9i2.ec01 (cit. on p. 6).

Jing, L., & Tian, Y. (2021). Self-Supervised Visual Feature Learning With Deep Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *43*(11), 4037–4058. https://doi.org/10.1109/TPAMI.2020.2992393 (cit. on p. 41).

Jo, T. (2023). Supervised Learning. In *Deep Learning Foundations* (pp. 29–55). Springer International Publishing. https://doi.org/10.1007/978-3-031-32879-4_2 (cit. on p. 4).

Johannesson, P., & Perjons, E. (2021). *An Introduction to Design Science*. Springer International Publishing. https://doi.org/10.1007/978-3-030-78132-3 (cit. on pp. 10, 11).

Jolliffe, I. (2011). Principal Component Analysis. In M. Lovric (Ed.), *International Encyclopedia of Statistical Science* (pp. 1094–1096). Springer. https://doi.org/10.1007/978-3-642-04898-2_455 (cit. on p. 43).

Jurczyk, M. (2016, October). *Windows Metafiles: An Analysis of the EMF Attack Surface & Recent Vulnerabilities*. Retrieved June 22, 2025, from https://j00ru.vexillium.org/talks/ruxcon-windows-metafiles-an-analysis-of-the-emf-attack-surface-and-recent-vulnerabilities/ (cit. on p. 7).

Kabanda, G. (2020). Performance of Machine Learning and Other Artificial Intelligence Paradigms in Cybersecurity. *Oriental Journal of Computer Science and Technology*, *13*(1), 1–21. https://doi.org/10.13005/ojcst13.01.01 (cit. on p. 4).

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, *4*, 237–285. https://doi.org/10.48550/arXiv.cs/9605103 (cit. on p. 44).

Kaloudi, N., & Li, J. (2020). The AI-based Cyber Threat Landscape: A Survey. *ACM Comput. Surv.*, *53*(1). https://doi.org/10.1145/3372823 (cit. on pp. 5, 8, 17).

Karamcheti, S., Mann, G., & Rosenberg, D. (2018). Adaptive Grey-box Fuzz-testing with Thompson Sampling. *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, 37–47. https://doi.org/10.1145/3270101.3270108 (cit. on p. 33).

Karras, T., Laine, S., & Aila, T. (2019). A Style-Based Generator Architecture for Generative Adversarial Networks. https://doi.org/10.48550/arXiv.1812.04948 (cit. on p. 43).

Kaspersky. (2024). *Shadowy innovation: how cybercriminals experiment with AI on the dark web*. https://dfi.kaspersky.com/blog/ai-in-darknet (cit. on p. 16).

Katrenko, A. (2023). *Catching Sandbox-Evading Malware: Techniques, Principles & Solutions*. https://www.apriorit.com/dev-blog/545-sandbox-evading-malware

Kim, D., & Solomon, M. (2021). *Fundamentals of Information Systems Security*. Jones & Bartlett Learning. https://books.google.hu/books?id=DiVGEAAAQBAJ (cit. on p. 6).

King, T. C., Aggarwal, N., Taddeo, M., & Floridi, L. (2020). Artificial Intelligence Crime: An Interdisciplinary Analysis of Foreseeable Threats and Solutions. *Science and Engineering Ethics*, *26*(1), 89–120. https://doi.org/10.1007/s11948-018-00081-0

Kiriansky, V., Bruening, D., & Amarasinghe, S. (2003). Execution model enforcement via program shepherding. *LCS Technical Memos*. https://dspace.mit.edu/handle/1721.1/149325 (cit. on p. 66).

Klees, G., Ruef, A., Cooper, B., Wei, S., & Hicks, M. (2018). Evaluating Fuzz Testing. *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2123–2138. https://doi.org/10.48550/arXiv.1808.09700 (cit. on p. 106).

Konoiko, A., Kadhem, A., Saiful, I., Ghorbanian, N., Zweiri, Y., & Sahinkaya, M. N. (2019). Deep learning framework for controlling an active suspension system. *JVC/Journal of Vibration and Control*, *25*(17), 2316–2329. https://doi.org/10.1177/1077546319853070

Korkmaz, E. (2024). A Survey Analyzing Generalization in Deep Reinforcement Learning. https://doi.org/10.48550/arXiv.2401.02349 (cit. on p. 44).

Koschke, R., Girard, J.-F., & Wurthner, M. (1998). An intermediate representation for integrating reverse engineering analyses. *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, 241–250. https://doi.org/10.1109/WCRE.1998.723194 (cit. on p. 30).

Kotsiantis, S. B., Zaharakis, I., Pintelas, P., et al. (2007). Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, *160*(1), 3–24. https://informatica.si/index.php/informatica/article/viewFile/148/140 (cit. on pp. 38, 39).

Kramer, S., & Bradfield, J. C. (2010). A general definition of malware. *Journal in Computer Virology*, *6*. https://doi.org/10.1007/s11416-009-0137-1 (cit. on p. 18).

Kubovic, O., Kosinar, P., & Janosik, J. (2018). *Can Artificial Intelligence Power Future Malware*. https://web-assets.esetstatic.com/wls/2018/08/Can_AI_Power_Future_Malware.pdf (cit. on pp. 3, 4).

Kumar, S., Marklund, H., Rao, A., Zhu, Y., Jeon, H. J., Liu, Y., & Roy, B. V. (2023). Continual Learning as Computationally Constrained Reinforcement Learning. https://doi.org/10.48550/arXiv.2307.04345 (cit. on p. 44).

Kumari, R., & Srivastava, S. K. (2017). Machine Learning: A Review on Binary Classification. *International Journal of Computer Applications*, *160*(7). https://doi.org/10.5120/ijca2017913083 (cit. on p. 38).

Kurzweil, R. (2014). The Singularity is Near. In R. L. Sandler (Ed.), *Ethics and Emerging Technologies* (pp. 393–406). Palgrave Macmillan UK. https://doi.org/10.1057/9781137349088_26 (cit. on p. 2).

Laka, D. (2022). Malware: Types, Analysis and Classification. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4036836 (cit. on p. 18).

Lee, S., Han, H., Cha, S. K., & Son, S. (2020). Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. *Proceedings of the 29th USENIX Security Symposium*. http://arxiv.org/abs/2001.04107

Lemieux, C. (2021). *Expanding the Reach of Fuzz Testing* [Doctoral dissertation, University of California, Berkeley]. https://www.proquest.com/openview/e8680a49f9f44faf86afc01940 (cit. on p. 52).

Lemieux, C., & Sen, K. (2018). FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. https://doi.org/10.1145/3238147.3238176 (cit. on p. 51).

Leonard, S. (2016, September). Windows Image Media Types. https://doi.org/10.17487/RFC7903 (cit. on pp. 62, 63).

Li, J., Zhao, B., & Zhang, C. (2018). Fuzzing: A survey. *Cybersecurity*, *1*, 1–13. https://doi.org/10.1186/s42400-018-0002-y (cit. on pp. 29, 32, 33).

Li, Y., Ji, S., Lv, C., Chen, Y., Chen, J., Gu, Q., & Wu, C. (2019). V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing. https://doi.org/10.48550/arXiv.1901.01142 (cit. on pp. 22, 24, 25).

Liang, J., Jiang, Y., Wang, M., Jiao, X., Chen, Y., Song, H., & Choo, K.-K. R. (2021). DeepFuzzer: Accelerated Deep Greybox Fuzzing. *IEEE Transactions on Dependable and Secure Computing*, *18*(6), 2675–2688. https://doi.org/10.1109/TDSC.2019.2961339 (cit. on p. 31).

Libicki, M. C., Ablon, L., & Webb, T. (2015). *The Defender's Dilemma: Charting a Course Toward Cybersecurity*. Rand Corporation. https://doi.org/10.7249/RR1024 (cit. on p. 1).

Lin, J., Zhang, Q., Li, J., Sun, C., Zhou, H., Luo, C., & Qian, C. (2025). Automatic Library Fuzzing through API Relation Evolvement. *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS 2025)*. https://www.ndss-symposium.org/ndss-paper/automatic-library-fuzzing-through-api-relation-evolvement/

Liu, B., Shi, L., Cai, Z., & Li, M. (2012). Software Vulnerability Discovery Techniques: A Survey. *2012 Fourth International Conference on Multimedia Information Networking and Security*, 152–156. https://doi.org/10.1109/MINES.2012.202 (cit. on p. 21).

Liu, Y., Jin, M., Pan, S., Zhou, C., Zheng, Y., Xia, F., & Yu, P. (2022). Graph self-supervised learning: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *35*(6), 5879–5900. https://doi.org/10.1109/tkde.2022.3172903 (cit. on p. 37).

Lotfi, H. (2020, June). *Develop Your Harness as a Win32 Application*. Retrieved June 22, 2025, from https://x.com/hosselot/status/1273943606324314114 (cit. on p. 71).

Lyu, C., Ji, S., Li, Y., Zhou, J., Chen, J., & Chen, J. (2018). SmartSeed: Smart Seed Generation for Efficient Fuzzing. https://doi.org/10.48550/arXiv.1807.02606 (cit. on pp. 23, 46, 54, 58, 80).

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., & Beyah, R. (2019). MOPT: Optimized Mutation Scheduling for Fuzzers. *28th USENIX Security Symposium (USENIX Security 19)*, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu (cit. on pp. 33, 73).

Malatji, M., & Tolah, A. (2025). Artificial intelligence (AI) cybersecurity dimensions: a comprehensive framework for understanding adversarial and offensive AI. *AI and Ethics*, *5*(2), 883–910. https://doi.org/10.1007/s43681-024-00427-4 (cit. on p. 15).

Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2021). The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, *47*(11), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563 (cit. on p. 21).

Marshall, A., Rojas, R., Stokes, J., & Brinkman, D. (2022). Securing the Future of Artificial Intelligence and Machine Learning at Microsoft. *Microsoft Learn*. https://docs.microsoft.com/en-us/security/engineering/securing-artificial-intelligence-machine-learning (cit. on p. 6).

Mathis, B., Gopinath, R., & Zeller, A. (2020). Learning Input Tokens for Effective Fuzzing. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 27–37. https://doi.org/10.1145/3395363.3397348

Mc Mahon, M., & Herrmann, M. (2019, October 27). *pyWinAuto* (Version 0.6.8). https://github.com/pywinauto/pywinauto (cit. on p. 74).

Meshram, A. (2023). *Self-learning Anomaly Detection in Industrial Production*. KIT Scientific Publishing. https://doi.org/10.5445/KSP/1000152715 (cit. on p. 16).

Miao, J., & Zhu, W. (2022). Precision–recall curve (PRC) classification trees. *Evolutionary Intelligence*, *15*, 1545–1569. https://doi.org/10.1007/s12065-021-00565-2 (cit. on p. 83).

Miao, S., Wang, J., Zhang, C., Lin, Z., Gong, J., Zhang, X., & Li, J. (2022). Deep Learning in Fuzzing: A Literature Survey. *2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information (ICETCI)*, 220–223. https://doi.org/10.1109/ICETCI55101.2022.9832143

Microsoft. (2022, April 29). *[MS-EMF]: Enhanced Metafile Format*. Retrieved June 22, 2025, from https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-emf/91c257d7-c39d-4a36-9b1f-63e3f73d30ca (cit. on pp. 62, 63).

Microsoft. (2023, January). Windows GDI. https://learn.microsoft.com/en-us/windows/win32/gdi/windows-gdi (cit. on p. 61).

Miller, B. (1988). Fall 1988 CS736 Project List. https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf (cit. on pp. 21, 26).

Miller, B. P., Fredriksen, L., & So, B. (1990). An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, *33*(12), 32–44. https://doi.org/10.1145/96267.96279 (cit. on p. 26).

Miller, C. (2007). How smart is intelligent fuzzing-or-how stupid is dumb fuzzing [Video available at https://www.youtube.com/watch?v=k1rPhQzX1PA]. https://infocon.org/cons/DEF%20CON/DEF%20CON%2015/DEF%20CON%2015%20presentations/DEF%20CON%2015%20-%20miller.pdf (cit. on p. 28).

Miller, C., & Peterson, Z. N. J. (2007). Analysis of Mutation and Generation-based Fuzzing. *Independent Security Evaluators*, *4*. https://api.semanticscholar.org/CorpusID:199412673 (cit. on pp. 27, 34).

Mirza, M., & Osindero, S. (2014). Conditional Generative Adversarial Nets. https://doi.org/10.48550/arXiv.1411.1784 (cit. on p. 43).

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill. https://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/mlbook.html (cit. on p. 35).

MITRE. (1999, September). The CVE Program. https://www.cve.org/ (cit. on p. 92).

Mohan, G. B., Kumar, R. P., Krishh, P. V., Keerthinathan, A., Lavanya, G., Meghana, M. K. U., Sulthana, S., & Doss, S. (2024). An analysis of large language models:

their impact and potential applications. *Knowledge and Information Systems*, *66*, 5047–5070. https://doi.org/10.1007/s10115-024-02120-8 (cit. on p. 42).

Moir, R. (2009). *Defining Malware: FAQ*. https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10) (cit. on p. 18).

Molleri, J. S., Petersen, K., & Mendes, E. (2016). Survey Guidelines in Software Engineering: An Annotated Review. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1145/2961111.2962619 (cit. on pp. 8, 9).

Moore, H. D. (2004). Metasploit Framework Documentation. https://docs.metasploit.com (cit. on p. 20).

Moroz, M., Serebryany, K., & Tsar, S. (2016, August). *Guided in-process fuzzing of Chrome components*. https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html

Morris, M. R., Sohl-dickstein, J., Fiedel, N., Warkentin, T., Dafoe, A., Faust, A., Farabet, C., & Legg, S. (2024). Levels of AGI for Operationalizing Progress on the Path to AGI. https://doi.org/10.48550/arXiv.2311.02462 (cit. on p. 3).

Moukahal, L. J., Zulkernine, M., & Soukup, M. (2021). Vulnerability-Oriented Fuzz Testing for Connected Autonomous Vehicle Systems. *IEEE Transactions on Reliability*, *70*(4), 1422–1437. https://doi.org/10.1109/TR.2021.3112538

MSRC. (2022a). *Windows GDI Information Disclosure Vulnerability*. Retrieved May 25, 2024, from https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-21904 (cit. on p. 93).

MSRC. (2022b). *Windows GDI+ Information Disclosure Vulnerability*. Retrieved May 25, 2024, from https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-21915 (cit. on p. 95).

MSRC. (2022c). *Windows Graphics Component Information Disclosure Vulnerability*. Retrieved May 25, 2024, from https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-26934 (cit. on p. 96).

MSRC. (2022d). *Windows Graphics Component Information Disclosure Vulnerability*. Retrieved May 25, 2024, from https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-29112 (cit. on p. 97).

Mtetwa, N. (2020, February). *Week 9 - Primer on Software Security Testing*. Retrieved June 22, 2025, from https://edshare.gcu.ac.uk/4804/2/index.html

Mugeshwaran, K. (2023). Unmasking Malware: An In-Depth Exploration Of Evasive Techniques And Innovative Countermeasures In Sandboxed Environments. *IJSART*, *9*. https://ijsart.com/unmasking-malware-an-in-depth-exploration-of-evasive-techniques-and-innovative-countermeasures-in-sandboxed-environments-73415 (cit. on p. 18).

Nagy, S., & Hicks, M. (2019). Full-Speed Fuzzing: Reducing Fuzzing Overhead Through Coverage-Guided Tracing. *2019 IEEE Symposium on Security and Privacy (SP)*, 787–802. https://doi.org/10.1109/SP.2019.00069

Nascimento, N. (2024). *Safeguard Analyzer: A Fuzz Testing Tool* [Master's thesis, Bosch Car Multimedia Portugal, S.A.]. https://www.proquest.com/openview/7e250bd35d05bf5ca31b7e (cit. on p. 31).

Nasrabadi, M. Z., Parsa, S., & Kalaee, A. (2018). Neural Fuzzing: A Neural Approach to Generate Test Data for File Format Fuzzing. *arXiv preprint arXiv:1812.09961*. https://doi.org/10.48550/arXiv.1812.09961

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., & Mian, A. (2024). A Comprehensive Overview of Large Language Models. https://doi.org/10.48550/arXiv.2307.06435 (cit. on p. 42).

Nichols, N., Raugas, M., Jasper, R., & Hilliard, N. (2017, November). Faster Fuzzing: Reinitialization with Deep Neural Models. https://doi.org/10.48550/arXiv.1711.02807 (cit. on p. 50).

Nunez, A., Islam, N. T., Jha, S. K., & Najafirad, P. (2024). AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing. https://doi.org/10.48550/arXiv.2409.10737 (cit. on p. 33).

Nurse, J. R., Adamos, K., Grammatopoulos, A., & Di Franco, F. (2021). Addressing the EU Cybersecurity Skills Shortage and Gap Through Higher Education. *European Union Agency for Cybersecurity (ENISA) Report*. https://doi.org/10.2824/033355 (cit. on p. 15).

Oikonomoy, A. (2023). *Vulnerability discovery in Android System through fuzzing* [Master's thesis, Aristotle University of Thessaloniki]. https://ikee.lib.auth.gr/record/346304/files/GRI-2023-38524.pdf

Osisanwo, F., Akinsola, J., Awodele, O., Hinmikaiye, J., Olakanmi, O., Akinjobi, J., et al. (2017). Supervised Machine Learning Algorithms: Classification and Comparison. *International Journal of Computer Trends and Technology (IJCTT)*, *48*(3), 128–138. https://doi.org/10.14445/22312803/IJCTT-V48P126 (cit. on p. 36).

Ostrand, T. J., & Weyuker, E. J. (2002). The Distribution of Faults in a Large Industrial Software System. *SIGSOFT Softw. Eng. Notes*, *27*(4), 55–64. https://doi.org/10.1145/566171.566181 (cit. on p. 53).

Ozkaya, E., & Aslaner, M. (2019). *Hands-On Cybersecurity for Finance: Identify Vulnerabilities and Secure your Financial Services from Security Breaches*. Packt. https://www.packtpub.com/product/hands-on-cybersecurity-for-finance/9781788836296

Padhye, R., Lemieux, C., Sen, K., Papadakis, M., & Le Traon, Y. (2019). Semantic Fuzzing with Zest. *Proceedings of the 28th ACM SIGSOFT International Symposium on*

Software Testing and Analysis. https://doi.org/10.1145/3293882.3330576 (cit. on p. 34).

Padovan, S. (2024, January). *AI-powered Ransomware to Stay Hidden* [Master's thesis, University of Zurich]. https://doi.org/10.5167/uzh-262744 (cit. on p. 19).

Pan, J., & Fung, C. (2008). Artificial intelligence in malware - Cop or culprit? *Ninth Postgraduate Electrical Engineering and Computing Symposium, (PEECS2008)*. https://researchportal.murdoch.edu.au/esploro/outputs/conferencePaper/Artificial-intelligence-in-malware---Cop/991005543807607891 (cit. on p. 18).

Pandit, B. (2023). *Popular Dimensionality Reduction Techniques Every Data Scientist Should Learn*. https://dataaspirant.com/dimensionality-reduction-techniques/ (cit. on p. 43).

Pang, H., Jian, J., Zhuang, Y., Ye, Y., & Li, Z. (2021). SpotFuzz: Fuzzing Based on Program Hot-Spots. *Electronics*, *10*(24). https://doi.org/10.3390/electronics10243142 (cit. on p. 67).

Pantoulas, E. (2022). *Description, Analysis and Implementation of a Web Application Firewall (WAF). Creation of Attack Scenarios and Threats Prevention* [Master's thesis, University of Piraeus]. https://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/14250/Pantoulas_MTE1924.pdf

Panwar, A., Narendra, M., Arya, A., Raj, R., & Kumar, A. (2025). Integrated portable ECG monitoring system with CNN classification for early arrhythmia detection. *Frontiers in Digital Health*, *Volume 7 - 2025*. https://doi.org/10.3389/fdgth.2025.1535335 (cit. on p. 59).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*(85), 2825–2830. https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html (cit. on p. 75).

Pena, J., Lozano, J., Larranaga, P., & Inza, I. (2001). Dimensionality reduction in unsupervised learning of conditional Gaussian networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *23*(6), 590–603. https://doi.org/10.1109/34.927460 (cit. on p. 40).

Pham, V.-T., Böhme, M., Santosa, A. E., Căciulescu, A. R., & Roychoudhury, A. (2021). Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, *47*(9), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681 (cit. on p. 64).

Pitafi, S., Anwar, T., & Sharif, Z. (2023). A Taxonomy of Machine Learning Clustering Algorithms, Challenges, and Future Realms. *Applied Sciences*, *13*(6). https://doi.org/10.3390/app13063529 (cit. on p. 40).

Polemi, N., & Praça, I. (2023). Multilayer Framework for Good Cybersecurity Practices for AI. https://doi.org/10.2824/588830

Polyakov, A., & Brusentsev, V. (2005). *Graphics Programming with GDI+ and DirectX*. A-List Publishing. https://books.google.hu/books?id=MJ_VAwAAQBAJ (cit. on p. 61).

Pramanik, A., & Tayade, A. (2017). Study and Comparison of General Purpose Fuzzers. *University of Wisconsin-Madison*. https://theultramarine19.github.io/data/736.pdf (cit. on p. 27).

Pratt, M. (2023, September). *Emerging Cyber Threats in 2023 from AI to Quantum to Data Poisoning*. Retrieved June 22, 2025, from https://www.csoonline.com/article/651125/emerging-cyber-threats-in-2023-from-ai-to-quantum-to-data-poisoning.html (cit. on pp. 17, 18).

Prechelt, L. (2012). Early Stopping – But When? In *Neural Networks: Tricks of the Trade: Second Edition* (pp. 53–67). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_5 (cit. on p. 37).

Priyambudi, Z. S., & Nugroho, Y. S. (2024). Which algorithm is better? An implementation of normalization to predict student performance. *AIP Conference Proceedings*, *2926*(1), 020110. https://doi.org/10.1063/5.0182879 (cit. on p. 79).

Pylant, A. C. (2020). *Initiating a Collaborative Cybersecurity Governance Framework at the State Level* [Doctoral dissertation, West Chester University Doctoral Projects]. https://digitalcommons.wcupa.edu/all_doctoral/59

Qian, R., Zhang, Q., Fang, C., & Guo, L. (2022). Investigating Coverage Guided Fuzzing with Mutation Testing. *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 272–281. https://doi.org/10.1145/3545258.3545285 (cit. on p. 33).

Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. https://doi.org/10.48550/arXiv.1511.06434 (cit. on p. 43).

Rajpal, M., Blum, W., & Singh, R. (2017). Not all bytes are equal: Neural byte sieve for fuzzing. https://doi.org/10.48550/arXiv.1711.04596 (cit. on pp. 25, 26).

Ramgulam, A., Ertekin, T., & Flemings, P. B. (2007). Utilization of artificial neural networks in the optimization of history matching. *SPE Latin American and Caribbean Petroleum Engineering Conference*, 799–813. https://doi.org/10.2523/107468-ms

Rasamoelina, A. D., Adjailia, F., & Sinčák, P. (2020). A Review of Activation Function for Artificial Neural Network. *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 281–286. https://doi.org/10.1109/SAMI48414.2020.9108717 (cit. on p. 80).

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). VUzzer: Application-aware Evolutionary Fuzzing. *NDSS*, *17*, 1–14. https://doi.org/10.14722/ndss.2017.23404 (cit. on p. 24).

Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., & Brumley, D. (2014). Optimizing Seed Selection for Fuzzing. *23rd USENIX Security Sympo-*

sium (USENIX Security 14), 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert (cit. on pp. 46, 47).

Rebstock, R. (2025, April). *Library: Northwestern AI in Education: Introduction to AI*. https://library.ctstate.edu/c.php?g=1451539&p=10789520 (cit. on p. 2).

Rehberger, J. (2020). *Participating in the Microsoft Machine Learning Security Evasion Competition – Bypassing malware models by signing binaries*. https://embracethered.com/blog/posts/2020/microsoft-machine-learning-security-evasion-competition/ (cit. on p. 19).

Reichelt, D. G., Bulej, L., Jung, R., & van Hoorn, A. (2024). Overhead Comparison of Instrumentation Frameworks. *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 249–256. https://doi.org/10.1145/3629527.3652269 (cit. on p. 65).

Reinders, J. (2013, September). *Processor Tracing*. https://web.archive.org/web/20130922062253/https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing (cit. on p. 66).

Ricardo Rodriguez-Torrealba, A. G.-C., Eva Garcia-Lopez. (2025). Joint Generation of Distractors for Multiple-Choice Questions: A Text-to-Text Approach. *Computers, Materials & Continua*, *83*(2), 1683–1705. https://doi.org/10.32604/cmc.2025.062004

Ríos, J. C. C., Kopec-Harding, K., Eraslan, S., Page, C., Haines, R., Jay, C., & Embury, S. M. (2019). A Methodology for Using GitLab for Software Engineering Learning Analytics. https://doi.org/10.48550/arXiv.1903.06772 (cit. on p. 73).

Roccia, T. (2019). *Evolution of Malware Sandbox Evasion Tactics – A Retrospective Study*. Retrieved June 22, 2025, from https://www.mcafee.com/blogs/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/ (cit. on p. 18).

Rödel, A., & Bücheler, K. (2025, February). *Introducing HyperHook: A harnessing framework for Nyx*. https://neodyme.io/en/blog/hyperhook/

Roesch, M., et al. (1999). Snort: Lightweight Intrusion Detection for Networks. *LISA*, *99*(1), 229–238. https://www.usenix.org/legacy/publications/library/proceedings/lisa99/roesch.html (cit. on p. 20).

Röning, J., Laakso, M., Takanen, A., & Kaksonen, R. (2002). PROTOS - Systematic Approach to Eliminate Software Vulnerabilities. *Invited presentation at Microsoft Research*. http://www.gbppr.net/2600/protos/sota/MSR2002-protos/index.html (cit. on p. 34).

Ross, R. S., Pillitteri, V. Y., & Dempsey, K. L. (2022). Assessing Enhanced Security Requirements for Controlled Unclassified Information. https://doi.org/10.6028/NIST.SP.800-172A (cit. on p. 5).

Ruby, U., Yendapalli, V., et al. (2020). Binary cross entropy with deep learning technique for image classification. *Int. J. Adv. Trends Comput. Sci. Eng*, *9*(10). https://doi.org/10.30534/ijatcse/2020/175942020 (cit. on p. 82).

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 757–762. https://doi.org/10.1109/icmla.2018.00120

Saavedra, G. J., Rodhouse, K. N., Dunlavy, D. M., & Kegelmeyer, P. W. (2019, October). A Review of Machine Learning Applications in Fuzzing. https://doi.org/10.48550/arXiv.1906.11133 (cit. on pp. 8, 45).

Sanders, J., & McMullen, R. (2022, November 12). *pyemf3* (Version 3.3). https://github.com/jeremysanders/pyemf3 (cit. on p. 73).

Sari, Y., Arifin, Y. F., Novitasari, & Faisal, M. R. (2023). The Effect of Batch Size and Epoch on Performance of ShuffleNet-CNN Architecture for Vegetation Density Classification. *Proceedings of the 7th International Conference on Sustainable Information Engineering and Technology*, 39–46. https://doi.org/10.1145/3568231.3568239 (cit. on p. 82).

Sarker, I. H. (2021). Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Computer Science*, *2*(3), 160. https://doi.org/10.1007/s42979-021-00592-x (cit. on pp. 35, 36).

Schaad, A., & Binder, D. (2022). Deep-learning-based Vulnerability Detection in Binary Executables. *International Symposium on Foundations and Practice of Security*, 453–460. https://doi.org/10.48550/arXiv.2212.01254

Schneier, B. (2018). Artificial Intelligence and the Attack/Defense Balance. *IEEE Security & Privacy*, *16*(2), 96–96. https://doi.org/10.1109/msp.2018.1870857 (cit. on p. 1).

Selján, G. (2021a). *Out-of-bounds Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_BITBLT Record*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/out-of-bounds-read-information-disclosure-vulnerability-in-microsoft-windows-gdi-emr_bitblt-record/ (cit. on p. 96).

Selján, G. (2021b). *Out-of-bounds Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_CREATEDIBPATTERNBRUSHPT Record*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/out-of-bounds-read-information-disclosure-vulnerability-in-microsoft-windows-gdi-emr_createdibpatternbrushpt-record/ (cit. on p. 95).

Selján, G. (2021c). *Out-of-bounds Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_SETDIBITSTODEVICE Record*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/out-of-bounds-read-information-disclosure-

vulnerability-in-microsoft-windows-gdi-emr%5C_setdibitstodevice-record/ (cit. on p. 92).

Selján, G. (2021d). *Out-of-bounds Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_STRETCHDIBITS Record*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/out-of-bounds-read-information-disclosure-vulnerability-in-microsoft-windows-gdi-emr_stretchdibits-record/ (cit. on p. 93).

Selján, G. (2022a). *Arbitrary Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_STARTDOC Record*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/arbitrary-read-information-disclosure-vulnerability-in-microsoft-windows-gdi-emr_startdoc-record/ (cit. on p. 98).

Selján, G. (2022b). *Out-of-bounds Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_SETPIXELV Record*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/out-of-bounds-read-information-disclosure-vulnerability-in-microsoft-windows-gdi-emr_setpixelv-record/ (cit. on p. 98).

Selján, G. (2022c). *Out-of-bounds Read Information Disclosure Vulnerability in Microsoft Windows GDI+ EMR_STRETCHDIBITS Record (Again)*. Retrieved June 19, 2025, from https://www.seljan.hu/posts/out-of-bounds-read-information-disclosure-vulnerability-in-microsoft-windows-gdi-emr_stretchdibits-record-again/ (cit. on p. 99).

Selján, G. (2023). Assessing Offensive Cyber Capabilities. *Academic and Applied Research in Military and Public Management Science*, *22*. https://doi.org/10.32565/aarms.2023.3.1

She, D., Krishna, R., Yan, L., Jana, S., & Ray, B. (2020). MTFuzz: fuzzing with a multi-task neural network. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 737–749. https://doi.org/10.1145/3368089.3409723

She, D., Pei, K., Epstein, D., Yang, J., Ray, B., & Jana, S. (2019). NEUZZ: Efficient Fuzzing with Neural Program Smoothing. *2019 IEEE Symposium on Security and Privacy (SP)*, 803–817. https://doi.org/10.1109/SP.2019.00052 (cit. on p. 22).

Shi, J., Wang, Z., Feng, Z., Lan, Y., Qin, S., You, W., Zou, W., Payer, M., & Zhang, C. (2023). AIFORE: Smart Fuzzing Based on Automatic Input Format Reverse Engineering. *32nd USENIX Security Symposium (USENIX Security 23)*, 4967–4984. https://www.usenix.org/conference/usenixsecurity23/presentation/shi-ji (cit. on p. 32).

Shin, Y., & Williams, L. (2013). Can Traditional Fault Prediction Models be Used for Vulnerability Prediction? *Empirical Software Engineering*, *18*, 25–59. https://doi.org/10.1007/s10664-011-9190-8 (cit. on p. 53).

Shivamurthaiah, M., Kumar, P., Vinay, S., & Podaralla, R. (2023). *Intelligent Computing: An Introduction to Artificial Intelligence Book*. Shineeks Publishers. https:

//shineeks.com/product/intelligent-computing-an-introduction-to-artificial-intelligence-book/

Shoaib, M., & Umar, M. S. (2023). An investigation in detection and mitigation of smishing using machine learning techniques. *Social Network Analysis and Mining*, *13*, 135. https://doi.org/10.1007/s13278-023-01142-4 (cit. on p. 17).

Simmons, A., & Chappell, S. (1988). Artificial Intelligence-Definition and Practice. *IEEE Journal of Oceanic Engineering*, *13*(2), 14–42. https://doi.org/10.1109/48.551 (cit. on p. 2).

Slaney, R. (2024, March). *One-day, n-day, and zero-day vulnerabilities explained*. Retrieved June 22, 2025, from https://fieldeffect.com/blog/1-day-0-day-vulnerabilities-explained

Smith, B., & Shum, H. (2018). The Future Computed. *Artificial Intelligence and Its Role in Society*. https://blogs.microsoft.com/blog/2018/01/17/future-computed-artificial-intelligence-role-society/ (cit. on p. 3).

Sofia, V. (2024). *Analysis and Innovation in Time Series Classification Algorithms and Methods: Towards a Superior Convolutional Algorithm with Feature Selection* [Master's thesis, Aristotle University of Thessaloniki]. https://ikee.lib.auth.gr/record/356506/files/GRI-2024-44168.pdf (cit. on p. 38).

Sparks, S., Embleton, S., Cunningham, R., & Zou, C. (2007). Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 477–486. https://doi.org/10.1109/ACSAC.2007.27 (cit. on p. 30).

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, *15*(1), 1929–1958 (cit. on p. 37).

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *23rd Annual Network and Distributed System Security Symposium, NDSS 2016*. https://doi.org/10.14722/ndss.2016.23368 (cit. on p. 30).

Stewart, L., Bach, F., Berthet, Q., & Vert, J.-P. (2023). Regression as Classification: Influence of Task Formulation on Neural Network Features. https://doi.org/10.48550/arXiv.2211.05641 (cit. on p. 39).

Suciu, O., Nelson, C., Lyu, Z., Bao, T., & Dumitras, T. (2021). Expected Exploitability: Predicting the Development of Functional Vulnerability Exploits. *Proceedings of the 31st USENIX Security Symposium, Security 2022*. http://arxiv.org/abs/2102.07869 (cit. on p. 106).

Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional. http://www.fuzzing.org/ (cit. on p. 26).

Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction*. MIT press Cambridge. https://www.cambridge.org/core/journals/robotica/article/robot-learning-edited-by-jonathan-h-connell-and-sridhar-mahadevan-kluwer-boston-19931997-xii240-pp-isbn-0792393651-hardback-21800-guilders-12000-8995/737FD21CA908246DF17779E9C20B6DF6 (cit. on p. 44).

SweetScape. (2021). *Introduction to Templates and Scripts*. https://www.sweetscape.com/010editor/manual/IntroTempScripts.htm (cit. on p. 61).

Świechowski, M. (2022). Deep Learning and Artificial General Intelligence: Still a Long Way to Go. *arXiv preprint*. https://doi.org/10.48550/arXiv.2203.14963 (cit. on p. 3).

Swiecki, R., & Gröbert, F. (2016). *Honggfuzz*. https://honggfuzz.dev/ (cit. on p. 32).

Takanen, A., Demott, J. D., Miller, C., & Kettunen, A. (2008). *Fuzzing for software security testing and quality assurance*. Artech House. https://us.artechhouse.com/Fuzzing-for-Software-Security-Testing-and-Quality-Assurance-Second-Edition-P1930.aspx (cit. on p. 29).

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., & Zhai, C. (2014). Bug characteristics in open source software. *Empirical Software Engineering*, *19*. https://doi.org/10.1007/s10664-013-9258-8 (cit. on p. 18).

Teegavarapu, S., Summers, J., & Mocko, G. (2008). Case Study Method for Design Research: A Justification. *Proceedings of the ASME Design Engineering Technical Conference*, *4*. https://doi.org/10.1115/DETC2008-49980 (cit. on p. 11).

Tubberville, J., & Vest, J. (2020). *Red Team Development and Operations: A Practical Guide*. Independently Published. https://redteam.guide/ (cit. on p. 16).

Turchin, A. (2015). *A Map: AGI Failures Modes and Levels*. https://www.lesswrong.com/posts/hMQ5iFiHkChqgrHiH/a-map-agi-failures-modes-and-levels (cit. on p. 6).

Vaishnavi, V., Kuechler, B., & Petter, S. (2004). Design Science Research in Information Systems. *DESRIST*. http://www.desrist.org/design-research-in-information-systems/ (cit. on p. 10).

Valencia, L. J. (2024, May). Artificial Intelligence as the New Hacker: Developing Agents for Offensive Security. https://doi.org/10.48550/arXiv.2406.07561 (cit. on p. 5).

Van der Maaten, L., & Hinton, G. (2008). Visualizing Data Using t-SNE. *Journal of machine learning research*, *9*(86), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html (cit. on p. 43).

Van Rossum, G. (2024, April 2). *Python* (Version 3.11.9). https://www.python.org/ (cit. on p. 74).

van Rooij, O., Charalambous, M. A., Kaizer, D., Papaevripides, M., & Athanasopoulos, E. (2021). webFuzz: Grey-Box Fuzzing for Web Applications. *Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security,*

Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I, 152–172. https://doi.org/10.1007/978-3-030-88418-5_8 (cit. on p. 31).

Versloot, C. (2021, January). *Binary Crossentropy Loss with PyTorch, Ignite and Lightnin*. https://machinecurve.com/index.php/2021/01/20/binary-crossentropy-loss-with-pytorch-ignite-and-lightning (cit. on p. 59).

Vikram, V., Laybourn, I., Li, A., Nair, N., OBrien, K., Sanna, R., & Padhye, R. (2023). Guiding Greybox Fuzzing with Mutation Testing. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 929–941. https://doi.org/10.1145/3597926.3598107 (cit. on p. 31).

Vysala, A., & Gomes, J. (2020). Evaluating and Validating Cluster Results. https://doi.org/10.48550/arXiv.2007.08034 (cit. on p. 40).

Wang, D., Zhang, Z., Zhang, H., Qian, Z., Krishnamurthy, S. V., & Abu-Ghazaleh, N. (2021). SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. *30th USENIX Security Symposium (USENIX Security 21)*, 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

Wang, H., Zariphopoulou, T., & Zhou, X. (2019). Exploration versus exploitation in reinforcement learning: a stochastic control approach. https://doi.org/10.48550/arXiv.1812.01552 (cit. on p. 44).

Wang, J., Chen, B., Wei, L., & Liu, Y. (2017). Skyfire: Data-Driven Seed Generation for Fuzzing. *2017 IEEE Symposium on Security and Privacy (SP)*, 579–594. https://doi.org/10.1109/SP.2017.23 (cit. on pp. 34, 48).

Wang, M., & Robinson, M. (2023). Vulnerability Detection Through an Adversarial Fuzzing Algorithm. *arXiv preprint arXiv:2307.11917*. https://doi.org/10.48550/arXiv.2307.11917 (cit. on pp. 106, 107).

Wang, Y., Jia, P., Liu, L., Huang, C., & Liu, Z. (2020). A Systematic Review of Fuzzing based on Machine Learning Techniques. *PLOS ONE*, *15*(8), 1–37. https://doi.org/10.1371/journal.pone.0237749 (cit. on pp. 21–23, 53).

Wang, Y., Wu, Z., Wei, Q., & Wang, Q. (2019). NeuFuzz: Efficient fuzzing with deep neural network. *IEEE Access*, *7*, 36340–36352. https://doi.org/10.1109/access.2019.2903291 (cit. on p. 23).

Wassermann, S., Meyer, M., Goutal, S., & Riquet, D. (2023, September). Targeted Attacks: Redefining Spear Phishing and Business Email Compromise. https://doi.org/10.48550/arXiv.2309.14166 (cit. on p. 18).

Watson, D. S. (2023). On the Philosophy of Unsupervised Learning. *Philosophy and Technology*, *36*. https://doi.org/10.1007/s13347-023-00635-6

Webb, G. I. (2010). Overfitting. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of Machine Learning* (pp. 744–744). Springer. https://doi.org/10.1007/978-0-387-30164-8_623 (cit. on p. 37).

WelTec, W.
bibinitperiod. (n.d.). *What is a software bug?* Retrieved June 22, 2025, from https://moodle.whitireiaweltec.ac.nz/local/pages/?id=14 (cit. on p. 18).

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer. https://doi.org/10.1007/978-3-662-43839-8 (cit. on pp. 9, 10, 12).

Williams, A., & Nicolett, M. (2005). Improve IT Security with Vulnerability Management. *Technical report*, (G00127481). https://www.gartner.com/doc/480703 (cit. on p. 20).

Winkler, J. P., Grönberg, J., & Vogelsang, A. (2019). Optimizing for Recall in Automatic Requirements Classification: An Empirical Study. *2019 IEEE 27th International Requirements Engineering Conference (RE)*, 40–50. https://doi.org/10.1109/RE.2019.00016 (cit. on p. 83).

Woo, M., Cha, S. K., Gottlieb, S., & Brumley, D. (2013). Scheduling Black-box Mutational Fuzzing. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 511–522. https://doi.org/10.1145/2508859.2516736 (cit. on p. 29).

Xie, Z., Cui, Z., Zhang, J., Liu, X., & Zheng, L. (2020). CSEFuzz: Fuzz Testing Based on Symbolic Execution. *IEEE Access*, *8*, 187564–187574. https://doi.org/10.1109/ACCESS.2020.3030798 (cit. on pp. 30, 46).

Yamin, M. M., Katt, B., & Gkioulos, V. (2020). Cyber Ranges and Security Testbeds: Scenarios, Functions, Tools and Architecture. *Computers & Security*, *88*, 101636. https://doi.org/10.1016/j.cose.2019.101636 (cit. on p. 16).

Yan, G., Lu, J., Shu, Z., & Kucuk, Y. (2017). ExploitMeter: Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability. *Proceedings - 2017 IEEE Symposium on Privacy-Aware Computing, PAC 2017*, *2017-January*. https://doi.org/10.1109/PAC.2017.10 (cit. on p. 106).

Yi, G., Yang, X., Huang, P., & Wang, Y. (2021). A Coverage-Guided Fuzzing Framework based on Genetic Algorithm for Neural Networks. *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, 352–358. https://doi.org/10.1109/DSA52907.2021.00054 (cit. on p. 24).

Yilmaz, A. (2023). *What is Machine Learning?* https://medium.com/@DataOpsEngineer/what-is-machine-learning-5151f1dfa10d (cit. on p. 35).

Ying, X. (2019). An Overview of Overfitting and its Solutions. *Journal of Physics: Conference Series*, *1168*(2), 022022. https://doi.org/10.1088/1742-6596/1168/2/022022 (cit. on p. 37).

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., & Rossow, C. (2016). SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In F.

Monrose, M. Dacier, G. Blanc, & J. Garcia-Alfaro (Eds.), *Research in Attacks, Intrusions, and Defenses* (pp. 165–187). Springer International Publishing. https://doi.org/10.1007/978-3-319-45719-2_8 (cit. on p. 19).

You, W., Liu, X., Ma, S., Perry, D., Zhang, X., & Liang, B. (2019). SLF: Fuzzing without Valid Seed Inputs. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 712–723. https://doi.org/10.1109/ICSE.2019.00080 (cit. on p. 34).

You, W., Zong, P., Chen, K., Wang, X., Liao, X., Bian, P., & Liang, B. (2017). SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2139–2154. https://doi.org/10.1145/3133956.3134085 (cit. on p. 31).

Yudkowsky, E. (2008, July). Artificial Intelligence as a positive and negative factor in global risk. In *Global Catastrophic Risks*. Oxford University Press. https://doi.org/10.1093/oso/9780198570509.003.0021 (cit. on pp. 3, 6).

Zakeri Nasrabadi, M., Parsa, S., & Kalaee, A. (2021). Format-Aware Learn&Fuzz: Deep Test Data Generation for Efficient Fuzzing. *Neural Computing and Applications*, *33*, 1497–1513. https://doi.org/10.1007/s00521-020-05039-7 (cit. on p. 50).

Zalewski, M. (2014). *Technical Whitepaper for afl-fuzz*. Retrieved June 22, 2025, from https://lcamtuf.coredump.cx/afl/technical%5C_details.txt (cit. on p. 67).

Zalewski, M. (2017, November 4). *American Fuzzy Lop (AFL) fuzzer* (Version 2.52b). https://lcamtuf.coredump.cx/afl/ (cit. on p. 67).

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., & Holler, C. (2024). The Fuzzing Book. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/html/00_Table_of_Contents.html (cit. on p. 27).

Zhang, Y., & Du, Z. (2023). Optimizing WinAFL for Image Parsing Engine Vulnerability Discovery in PDF Readers. *Frontiers in Computing and Intelligent Systems*, *3*(1), 78–81. https://doi.org/10.54097/fcis.v3i1.6028 (cit. on p. 68).

Zhang, Z. (2018). Improved Adam Optimizer for Deep Neural Networks. *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 1–2. https://doi.org/10.1109/IWQoS.2018.8624183 (cit. on pp. 75, 82).

Zhao, Q., & Wong, W.-F. (2005, January). Ameliorating the Overhead of Dynamic Optimization. https://dspace.mit.edu/handle/1721.1/7411 (cit. on p. 66).

Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., & Sun, L. (2019). FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. *28th USENIX Security Symposium (USENIX Security 19)*, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng (cit. on p. 67).

Zhong, Y., Wei, Y., Liang, Y., Liu, X., Ji, R., & Cang, Y. (2024). A comparative study of generative adversarial networks for image recognition algorithms based on deep

learning and traditional methods. https://doi.org/10.48550/arXiv.2408.03568 (cit. on p. 42).

Zhu, Q., & Liang, L. (2019). Research on Security Vulnerabilities Based on Artificial Intelligence. *Intelligent Computing Theories and Application*, 377–387. https://doi.org/10.1007/978-3-030-26763-6_37 (cit. on p. 21).

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., & He, Q. (2020). A Comprehensive Survey on Transfer Learning. https://doi.org/10.48550/arXiv.1911.02685 (cit. on p. 36).

Zong, P., Lv, T., Wang, D., Deng, Z., Liang, R., & Chen, K. (2020). FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-Box Fuzzing Through Deep Learning. *Proceedings of the 29th USENIX Conference on Security Symposium*. https://www.usenix.org/conference/usenixsecurity20/presentation/zong (cit. on p. 53).

Appendices

A. Assessment results

#	Cycles	Speed	Progress	Stability	Coverage	Crashes	Bucket
1	0	124.47	608/12208 (4.98%)	69.08%	38.79%	3	6
2	26	120.31	4128/12229 (33.76%)	68.51%	38.81%	2	0
3	25	112.53	11569/12238 (94.53%)	69.13%	38.81%	3	6
4	27	117.48	1575/12247 (12.86%)	68.81%	38.81%	2	3
5	37	107.45	10941/12239 (89.39%)	68.16%	38.81%	1	0
6	40	110.05	8422/12213 (68.96%)	67.68%	38.81%	3	4
7	39	109.19	11528/12247 (94.13%)	68.26%	38.81%	2	3
8	39	110.16	11168/12232 (91.30%)	68.84%	38.81%	2	1
9	39	108.52	10270/12241 (83.90%)	68.80%	38.81%	3	0
10	40	106.12	11812/12264 (96.31%)	68.22%	38.81%	4	3
11	54	93.17	8967/12253 (73.18%)	68.69%	38.81%	1	4
12	40	110.73	12210/12265 (99.55%)	68.43%	38.81%	3	6
13	39	106.85	6136/12280 (49.97%)	61.89%	38.81%	2	2
14	43	111.43	10754/12280 (87.57%)	68.52%	38.81%	2	1
15	39	107.52	11414/12242 (93.24%)	68.56%	38.81%	4	1
16	62	95.96	1271/12322 (10.31%)	68.84%	38.81%	3	1
17	58	94.81	10442/12243 (85.29%)	66.19%	38.81%	3	4
18	57	94.17	890/12283 (7.25%)	68.99%	38.81%	1	0
19	53	89.35	6250/12231 (51.10%)	68.14%	38.81%	2	3
20	56	95.45	10184/12258 (83.08%)	68.51%	38.81%	0	0

Table 9: Per fuzzer statistics for the *randomization* strategy

#	Cycles	Speed	Progress	Stability	Coverage	Crashes	Bucket
1	0	155.48	714/12856 (5.55%)	65.98%	39.09%	3	9
2	36	155.73	1399/12857 (10.88%)	65.57%	39.11%	3	0
3	35	158.64	2823/12899 (21.89%)	64.96%	39.11%	5	3
4	37	164.80	1974/12927 (15.27%)	65.29%	39.11%	3	1
5	35	158.37	3123/12909 (24.19%)	66.01%	39.11%	4	2
6	50	142.66	2088/12947 (16.13%)	65.01%	39.11%	2	1
7	54	145.17	4888/12978 (37.66%)	66.13%	39.11%	1	0
8	55	146.20	3970/12941 (30.68%)	65.25%	39.11%	3	2
9	52	143.46	4256/12986 (32.77%)	64.03%	39.11%	4	0
10	50	142.52	5119/12944 (39.55%)	63.75%	39.11%	2	0
11	53	145.51	4630/12981 (35.67%)	64.88%	39.11%	2	3
12	56	148.59	5092/12914 (39.43%)	66.41%	39.11%	4	0
13	56	150.33	5819/12953 (44.92%)	65.52%	39.11%	3	0
14	57	145.92	5645/12989 (43.46%)	65.98%	39.11%	1	1
15	55	145.76	5991/12919 (46.37%)	65.05%	39.11%	5	0
16	54	144.83	5785/12957 (44.65%)	66.21%	39.11%	2	0
17	76	129.23	8745/12947 (67.54%)	66.16%	39.11%	2	1
18	75	129.50	8937/12949 (69.02%)	66.41%	39.11%	3	0
19	78	129.85	4156/12950 (32.09%)	64.60%	39.10%	2	0
20	70	127.86	7461/12901 (57.83%)	66.44%	39.11%	5	3

Table 10: Per fuzzer statistics for the *minimization* strategy

#	Cycles	Speed	Progress	Stability	Coverage	Crashes	Bucket
1	0	185.81	1066/11641 (9.16%)	67.05%	39.03%	6	24
2	32	132.13	1143/11812 (9.68%)	65.89%	39.04%	6	224
3	31	136.72	11301/11890 (95.05%)	62.85%	39.04%	5	205
4	31	141.01	1536/11864 (12.95%)	66.11%	39.04%	4	226
5	30	143.67	1978/11839 (16.71%)	62.08%	39.04%	6	226
6	48	126.48	1753/11852 (14.79%)	64.78%	39.04%	7	217
7	46	123.21	3208/11884 (26.99%)	58.59%	39.04%	5	210
8	49	125.49	2795/11896 (23.50%)	64.78%	39.04%	7	214
9	48	126.21	2607/11853 (21.99%)	65.75%	39.04%	5	226
10	45	128.34	2082/11841 (17.58%)	66.09%	39.04%	7	228
11	46	116.14	2016/11838 (17.03%)	64.71%	39.04%	5	206
12	46	123.90	2425/11848 (20.47%)	65.75%	39.04%	6	223
13	70	104.69	4795/11830 (40.53%)	64.23%	39.04%	5	212
14	46	118.61	2212/11820 (18.71%)	62.60%	39.04%	5	192
15	50	122.24	2347/11805 (19.88%)	65.74%	39.04%	5	227
16	72	104.15	3117/11850 (26.30%)	64.79%	39.04%	5	200
17	72	104.34	3317/11795 (28.12%)	63.79%	39.04%	5	201
18	68	107.23	4201/11866 (35.40%)	64.81%	39.04%	5	211
19	64	105.43	4899/11829 (41.42%)	66.04%	39.04%	4	206
20	68	99.87	3639/11848 (30.71%)	66.68%	39.04%	5	199

Table 11: Per fuzzer statistics for the *prediction* strategy

#	Cycles	Speed	Progress	Stability	Coverage	Crashes	Bucket
1	0	149.79	615/11925 (5.16%)	69.36%	40.35%	2	9
2	28	93.36	4612/12110 (38,08%)	70.14%	40.37%	3	130
3	28	91.56	4740/12144 (39.03%)	64.34%	40.37%	3	126
4	27	93.23	7363/12148 (60.61%)	74.50%	40.36%	4	125
5	28	91.87	4685/12184 (38.45%)	67.62%	40.36%	4	135
6	29	92.96	4800/12164 (39.46%)	71.42%	40.36%	2	142
7	32	97.81	5368/12081 (44.43%)	65.54%	40.36%	3	131
8	31	97.09	9774/12151 (80.44%)	72.45%	40.36%	2	158
9	27	87.50	11461/12111 (94.63%)	67.48%	40.36%	3	136
10	30	89.28	402/12128 (3.31%)	71.28%	40.36%	4	132
11	46	81.12	7812/12128 (64.41%)	70.47%	40.36%	5	140
12	46	80.98	309/12109 (2.55%)	73.78%	40.36%	2	111
13	47	79.38	10397/12053 (86.26%)	65.82%	40.36%	2	136
14	48	80.79	11102/12129 (91.53%)	70.49%	40.36%	3	148
15	42	77.13	638/12111 (5.27%)	65.34%	40.36%	4	130
16	45	79.75	10564/12122 (87.15%)	75.85%	40.36%	2	153
17	48	82.25	7923/12151 (65.20%)	68.05%	40.36%	5	129
18	49	82.56	5525/12144 (45.50%)	67.77%	40.36%	3	132
19	47	78.86	530/12027 (4.41%)	68.28%	40.36%	3	108
20	45	80.83	9926/12072 (82.22%)	69.52%	40.36%	3	142

Table 12: Per fuzzer statistics for the *combined* strategy

B. Unique crash samples

File (SHA256)	Size	BugId	Library	Function
00AE644CE5C9	608	f33.cf7	GdiPlus.dll	DpRegion::Set
011E5AC5E789	220	2b1.cf1	Gdi32Full.dll	CreateFontIndirectWImpl
0330DE6EF25F	806	358.587	GdiPlus.dll	MRRESIZEPALETTE::bPlay
05B0FED97564	608	db3.f44	GdiPlus.dll	Div64_Asm
097C40E7A079	550	e5f.b0d	GdiPlus.dll	EmfEnumState::ModifyRecordColor
09A8E14E95E9	248	9ce.d2f	GDI32.dll	CreateDIBitmap
0CC3CBE5B224	446	3f4.c6a	GdiPlus.dll	GdipGetWinMetaFileBitsEx
1455438460E2	144	d05.7fb	GdiPlus.dll	EmfEnumState::CreatePen
19917F1A9771	248	6b3.d8d	Gdi32Full.dll	StretchDIBitsImpl
1FC0FB7EFAE0	136	0e5.7fb	GdiPlus.dll	EmfEnumState::ExtCreatePen
21013B2BF3D4	756	5ab.a57	GdiPlus.dll	CopyGindices
2544E3E4EE24	56020	a89.bf8	GDI32.dll	ConvertDxArray
29DE93CD3EFD	526	fe8.b17	GdiPlus.dll	MfEnumState::ModifyDib
2D3E1B99717E	159000	c31.d2f	GdiPlus.dll	DoRotatedStretchBlt
431E261EEC30	612	9e2.372	GdiPlus.dll	FullTextImager::Render
46F782B08565	904	552.f99	GdiPlus.dll	DoExtTextOut
4979D4379004	236	bba.7fb	GdiPlus.dll	EmfEnumState::ModifyRecordColor
51D49EC5E723	313	289.e06	GDI32.dll	pbmiConvertInfo
522AD0595FDE	780	cbf.7fb	GdiPlus.dll	EmfEnumState::CreateDibPatternBrushPt
54FF54F83BDB	280	1ce.77d	GdiPlus.dll	DrawImagePointsEPR::Play
5BA20DD2301B	472	9f7.7fb	GdiPlus.dll	EmfEnumState::SelectObject
64F96E2C43A4	232	999.7fb	GdiPlus.dll	EmfEnumState::SetROP2
671F4A6C7173	248	7c8.7c8	msvcrt.dll	_VEC_memcpy
6E39F2B013BE	3700	3a8.13d	GdiPlus.dll	DoGdiCommentMultiFormats
77526891CE54	555	f33.8ed	GdiPlus.dll	DpRegion::Set
79E5F5357EDA	1270	334.995	GdiPlus.dll	ToCOLORREF
8437F6E0B646	248	72d.895	GDI32.dll	MRSETDIBITSTODEVICE::bPlay
878209CA7A27	555	3ce.9b5	GdiPlus.dll	GpStringFormat::SetData
893201F60329	204	3cd.699	GdiPlus.dll	bHandleFrameRgn
8A18FBD266E6	248	72d.895	GdiPlus.dll	bEmitWin16StretchBlt
8DDB06D2AA0E	314	2b1.699	GdiPlus.dll	bHandlePaintRgn
99161BA9CBC0	769	6fe.4dc	GdiPlus.dll	EpAliasedFiller::FillEdgesAlternate
9F8741698275	500	8bb.46b	GdiPlus.dll	DoSetDIBitsToDevice
A0F5E53D7009	139	eb3.7fb	GdiPlus.dll	EmfEnumState::Rectangle
A31613A730D8	34662	be9.1bc	GdiPlus.dll	GpPen::GetMaximumWidth
B497A1311CF8	248	72d.9ce	GDI32.dll	CreateDIBitmap
BB7A4559A8E4	524	895.d2f	GdiPlus.dll	bEmitWin16StretchBlt
C3E2D0DCB379	1000	77e.699	GdiPlus.dll	bHandleExtCreateFont
C54C96B5171B	194	5c1.419	GdiPlus.dll	DoCreateMonoBrush
CDD925ECE5EB	372		Gdi32Full.dll	StartDocWImpl
D08258709EEE	500	a56.798	GDI32.dll	MRCREATECOLORSPACEW::bCheckRecord
D3C1377A4C15	160	009.699	GdiPlus.dll	bHandleInvertRgn
E00B5FAFBD7E	578	8ed.260	GdiPlus.dll	DpRegion::And
E112D9DC0123	137	efe.7fb	GdiPlus.dll	EmfEnumState::ExtCreateFontIndirect
F30797A5B8BB	626	932.8d1	GdiPlus.dll	ValidateBitmapInfo
F5F958FEF5E9	599	853.699	GdiPlus.dll	bHandleFillRgn
F7AFF247E121	252	699.0e9	GdiPlus.dll	bParseWin32Metafile
FD0F262AA4D7	854	673.fce	Gdi32Full.dll	MRBDIB::vInit
A390F3B1FEDF	400	520.506	Gdi32Full.dll	NamedEscape
44168A76232A	608	2df.af7	dwrite.dll	GlyphDataElement<...>::GetExistingGlyphs

Table 13: Summary of unique crash samples identified during the research

C. Stack backtraces

```
 # ChildEBP RetAddr
00 009df3b8 71165417 gdiplus!EmfEnumState::ModifyRecordColor+0x22
01 009df3dc 71127c5d gdiplus!EmfEnumState::ProcessRecord+0x61157
02 009df400 7114812c gdiplus!GdipPlayMetafileRecordCallback+0xdd
03 009df42c 76373945 gdiplus!EnumEmfDownLevel+0x6c
04 009df500 7636cb4c gdi32full!bInternalPlayEMF+0x855
05 009df514 76f7450b gdi32full!EnumEnhMetaFile+0x2c
06 009df534 711085a2 GDI32!EnumEnhMetaFileStub+0x2b
07 009df588 71105aac gdiplus!MetafilePlayer::EnumerateEmfRecords+0xc8
08 009df640 7110a07c gdiplus!GpGraphics::EnumEmf+0x464
09 009df7b0 71117461 gdiplus!GpMetafile::EnumerateForPlayback+0x651
0a 009df908 7112867f gdiplus!GpGraphics::DrawImage+0x541
0b 009df970 711976a8 gdiplus!GpGraphics::DrawImage+0x61
0c 009df9d0 711988bf gdiplus!GpMetafile::GetBitmap+0x1c0
0d 009df9e4 7117c8a5 gdiplus!GpMetafile::GetThumbnail+0x2f
0e 009dfa0c 005d128b gdiplus!GdipGetImageThumbnail+0x65
```

Listing 22: Stack trace of `EMR_SETPIXELV` (CVE-2022-34728)

```
 # ChildEBP RetAddr
00 012ff0f0 6eb1592c msvcrt!memcpy+0x5a
01 012ff124 6eac784e gdiplus!MfEnumState::ModifyDib+0x6e
02 012ff164 6eb13bfb gdiplus!EmfEnumState::CreateModifiedDib+0x61920
03 012ff198 6eac7608 gdiplus!EmfEnumState::CreateDibPatternBrushPt+0x126
04 012ff1ac 6ea889dd gdiplus!EmfEnumState::ProcessRecord+0x624c8
05 012ff1d0 6eaa8ecc gdiplus!GdipPlayMetafileRecordCallback+0xdd
06 012ff1fc 75c54b25 gdiplus!EnumEmfDownLevel+0x6c
07 012ff2d0 75c4dd2c gdi32full!bInternalPlayEMF+0x855
08 012ff2e4 76b6462b gdi32full!EnumEnhMetaFile+0x2c
09 012ff304 6ea69302 GDI32!EnumEnhMetaFileStub+0x2b
0a 012ff358 6ea6680c gdiplus!MetafilePlayer::EnumerateEmfRecords+0xc8
0b 012ff410 6ea6addc gdiplus!GpGraphics::EnumEmf+0x464
0c 012ff580 6ea781e1 gdiplus!GpMetafile::EnumerateForPlayback+0x651
0d 012ff6d8 6ea893ff gdiplus!GpGraphics::DrawImage+0x541
0e 012ff744 6eaf9898 gdiplus!GpGraphics::DrawImage+0x61
0f 012ff7a4 6eafaaaf gdiplus!GpMetafile::GetBitmap+0x1c0
10 012ff7b8 6eadea95 gdiplus!GpMetafile::GetThumbnail+0x2f
11 012ff7e0 002b128b gdiplus!GdipGetImageThumbnail+0x65
```

Listing 23: Stack trace of `EMR_CREATEDIBPATTERNBRUSHPT` (CVE-2022-26934)

```
 # ChildEBP RetAddr
00 00daed24 76861859 ucrtbase!memcpy+0x4e
01 00daed44 76861a0b gdi32full!MRBDIB::vInit+0x79
02 00daedf8 76874d24 gdi32full!MF_AnyDIBits+0x16d
03 00daeee4 76a871a3 gdi32full!StretchDIBitsImpl+0x134
04 00daef24 70505ed5 GDI32!StretchDIBits+0x43
05 00daf088 70505646 gdiplus!MfEnumState::OutputDIB+0x886
06 00daf0e8 705051e9 gdiplus!EmfEnumState::StretchDIBits+0x10b
07 00daf0fc 70528a0d gdiplus!EmfEnumState::ProcessRecord+0x79
08 00daf120 70548eec gdiplus!GdipPlayMetafileRecordCallback+0xdd
09 00daf14c 76863945 gdiplus!EnumEmfDownLevel+0x6c
0a 00daf220 7685cb4c gdi32full!bInternalPlayEMF+0x855
0b 00daf234 76a8462b gdi32full!EnumEnhMetaFile+0x2c
0c 00daf254 70509332 GDI32!EnumEnhMetaFileStub+0x2b
0d 00daf2a8 7050683c gdiplus!MetafilePlayer::EnumerateEmfRecords+0xc8
```

```
0e 00daf360 7050ae0c gdiplus!GpGraphics::EnumEmf+0x464
0f 00daf4d0 70518211 gdiplus!GpMetafile::EnumerateForPlayback+0x651
10 00daf628 7052942f gdiplus!GpGraphics::DrawImage+0x541
11 00daf694 70548d36 gdiplus!GpGraphics::DrawImage+0x61
12 00daf6f8 70548c07 gdiplus!GdipDrawImage+0x116
13 00daf718 008b12e6 gdiplus!GdipDrawImageI+0x37
```

Listing 24: Stack trace of `EMR_STRETCHDIBITS` (CVE-2022-21915)

```
 # ChildEBP RetAddr
00 0089f060 6b4220a0 gdiplus!ValidateBitmapInfo+0x17a
01 0089f4ac 6b406390 gdiplus!CopyOnWriteBitmap::CopyOnWriteBitmap+0x88
02 0089f4cc 6b40635b gdiplus!CopyOnWriteBitmap::Create+0x25
03 0089f4dc 6b405bc5 gdiplus!GpBitmap::GpBitmap+0x3a
04 0089f61c 6b4b3960 gdiplus!MfEnumState::OutputDIB+0x5a6
05 0089f6b8 6b4675e4 gdiplus!EmfEnumState::BitBlt+0x2a0
06 0089f6cc 6b4289dd gdiplus!EmfEnumState::ProcessRecord+0x624a4
07 0089f6f0 6b448ecc gdiplus!GdipPlayMetafileRecordCallback+0xdd
08 0089f71c 767f4b25 gdiplus!EnumEmfDownLevel+0x6c
09 0089f7f0 767edd2c gdi32full!bInternalPlayEMF+0x855
0a 0089f804 769c462b gdi32full!EnumEnhMetaFile+0x2c
0b 0089f824 6b409302 GDI32!EnumEnhMetaFileStub+0x2b
0c 0089f878 6b4068e6 gdiplus!MetafilePlayer::EnumerateEmfRecords+0xc8
0d 0089f930 6b40addc gdiplus!GpGraphics::EnumEmf+0x53e
0e 0089faa0 6b4181e1 gdiplus!GpMetafile::EnumerateForPlayback+0x651
0f 0089fbf8 6b4293ff gdiplus!GpGraphics::DrawImage+0x541
10 0089fc64 6b448d16 gdiplus!GpGraphics::DrawImage+0x61
11 0089fcc8 6b448be7 gdiplus!GdipDrawImage+0x116
12 0089fce8 009912e6 gdiplus!GdipDrawImageI+0x37
```

Listing 25: Stack trace of `EMR_BITBLT` (CVE-2022-29112)

```
 # ChildEBP RetAddr
00 009bee08 76a04393 ucrtbase!__ascii_wcsicmp+0x1b
01 009bee44 76a10c65 ucrtbase!__crt_state_management::wrapped_invoke<...>+0x2b
02 009bee50 76349f87 ucrtbase!_o__wcsicmp+0x15
03 009bef88 75e05112 gdi32full!StartDocWImpl+0x31547
04 009bef9c 76377c81 GDI32!StartDocW+0x22
05 009befd4 76326969 gdi32full!MRSTARTDOC::bPlay+0x81
06 009bf060 755a44ad gdi32full!PlayEnhMetaFileRecord+0x59
07 009bf078 755a4309 gdiplus!EmfEnumState::PlayRecord+0x2d
08 009bf094 755c7c5d gdiplus!EmfEnumState::ProcessRecord+0x49
09 009bf0b8 755e812c gdiplus!GdipPlayMetafileRecordCallback+0xdd
0a 009bf0e4 76323945 gdiplus!EnumEmfDownLevel+0x6c
0b 009bf1b8 7631cb4c gdi32full!bInternalPlayEMF+0x855
0c 009bf1cc 75e0450b gdi32full!EnumEnhMetaFile+0x2c
0d 009bf1ec 755a85a2 GDI32!EnumEnhMetaFileStub+0x2b
0e 009bf240 755a5aac gdiplus!MetafilePlayer::EnumerateEmfRecords+0xc8
0f 009bf2f8 755aa07c gdiplus!GpGraphics::EnumEmf+0x464
10 009bf468 755b7461 gdiplus!GpMetafile::EnumerateForPlayback+0x651
11 009bf5c0 755c867f gdiplus!GpGraphics::DrawImage+0x541
12 009bf62c 755e7f76 gdiplus!GpGraphics::DrawImage+0x61
13 009bf690 755e7e47 gdiplus!GdipDrawImage+0x116
14 009bf6b0 00c0136a gdiplus!GdipDrawImageI+0x37
```

Listing 26: Stack trace of `EMR_STARTDOC` (CVE-2022-35837)

D. Fuzzing harness

```
#include "GDI.h"

unsigned char* shm_data;
bool use_shared_memory;
HANDLE map_file;

int setup_shmem(LPCWSTR name) {
  map_file = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, name);
  if (map_file == NULL) return 0;

  shm_data = (unsigned char*)MapViewOfFile(map_file, FILE_MAP_ALL_ACCESS, 0, 0,
    MAX_SAMPLE_SIZE + sizeof(uint32_t));
  if (shm_data == NULL) return 0;

  return 1;
}

int clear_shmem(void) {
  UnmapViewOfFile(shm_data);
  CloseHandle(map_file);
  return 0;
}

extern "C" __declspec(noinline dllexport) int Fuzz(HDC hDC) {
  int argCount;
  LPWSTR* szArgList;
  HENHMETAFILE hEmf = NULL;
  uint32_t sample_size = 0;
  char* sample_bytes = NULL;
  IStream* stream = NULL;
  Gdiplus::Image* image = NULL;
  Gdiplus::Image* thumbnail = NULL;
  Gdiplus::Metafile* metafile = NULL;
  Gdiplus::Graphics* graphics = NULL;

  szArgList = CommandLineToArgvW(GetCommandLine(), &argCount);

  if (!use_shared_memory) {
    SHCreateStreamOnFile(szArgList[2], NULL, &stream);
    image = Gdiplus::Image::FromStream(stream);
    metafile = new Gdiplus::Metafile(stream, hDC);
  } else {
    if (shm_data == NULL) return 1;

    sample_size = *(uint32_t*)(shm_data);
    if (sample_size > MAX_SAMPLE_SIZE) sample_size = MAX_SAMPLE_SIZE;
    sample_bytes = (char*)malloc(sample_size);
    if (sample_bytes == NULL) return 1;
    memcpy(sample_bytes, shm_data + sizeof(uint32_t), sample_size);
    stream = SHCreateMemStream(reinterpret_cast<BYTE*>(sample_bytes), sample_size);
    image = Gdiplus::Image::FromStream(stream);
    metafile = new Gdiplus::Metafile(stream, hDC);
  }

```

```
54   if (image && (Ok == image->GetLastStatus()))
55     thumbnail = image->GetThumbnailImage(100, 100, NULL, NULL);
56     graphics = Graphics::FromImage(image);
57     if (graphics && (Ok == graphics->GetLastStatus()))
58       graphics->DrawImage(image, 0, 0);
59
60   if (metafile)
61     hEmf = metafile->GetHENHMETAFILE();
62     for (size_t x = 1; x <= 8; x++)
63       for (size_t y = 0; y <= 4; y++)
64         Gdiplus::Metafile::EmfToWmfBits(hEmf, 0, NULL, (INT)x, (INT)y);
65
66   if (image) delete image;
67   if (metafile) delete metafile;
68   if (thumbnail) delete thumbnail;
69   if (graphics) delete graphics;
70   if (sample_bytes) free(sample_bytes);
71   if (hEmf) DeleteEnhMetaFile(hEmf);
72   if (stream) stream->Release();
73   LocalFree(szArgList);
74
75   return 0;
76 }
77
78 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
       nCmdShow) {
79   int argCount;
80   LPWSTR* szArgList;
81   ULONG_PTR gdiplusToken;
82   GdiplusStartupInput gdiplusStartupInput;
83
84   GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
85
86   szArgList = CommandLineToArgvW(GetCommandLine(), &argCount);
87   if (szArgList == NULL || argCount < 3) return 1;
88   if (!wcscmp(szArgList[1], L"-s")) use_shared_memory = true;
89   else if (!wcscmp(szArgList[1], L"-f")) use_shared_memory = false;
90   else return 0;
91
92   if (use_shared_memory && !setup_shmem(szArgList[2])) return 1;
93
94   HDC hDC = GetDC(NULL);
95   Fuzz(hDC);
96
97   if (use_shared_memory) clear_shmem();
98   ReleaseDC(NULL, hDC);
99   LocalFree(szArgList);
100  GdiplusShutdown(gdiplusToken);
101
102  return 0;
103 }
```

Listing 27: Main functionality of the test harness used for fuzzing Windows GDI

E. Model prototype

```
import os, heapq, logging, argparse
import numpy as np
import pandas as pd
from base64 import b64encode
from tensorflow.keras import Input
from tensorflow.keras.regularizers import L2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.metrics import AUC, Recall, Precision
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import KFold, train_test_split

parser = argparse.ArgumentParser(prog='SeedClass')
parser.add_argument('-N', '--negative', dest='negative')
parser.add_argument('-P', '--positive', dest='positive')
parser.add_argument('-U', '--unknown', dest='unknown')
parser.add_argument('-E', '--encoding', dest='encoding', choices=['simple', 'product'
    ], default='simple')
parser.add_argument('-V', '--validation', dest='validation', action='store_true')
parser.add_argument('-R', '--regularization', dest='regularization', action='
    store_true')
args = parser.parse_args()

ENCODING       = args.encoding
VALIDATION     = args.validation
REGULARIZATION = args.regularization

FEATURE_RANGE_MAX_SIMPLE = 0.646464646464
FEATURE_RANGE_MAX_PRODUCT = (pow(64, 6)) / pow(10, 11) # 0.68719476736
FEATURE_RANGE_MAX = FEATURE_RANGE_MAX_SIMPLE if ENCODING == 'simple' else
    FEATURE_RANGE_MAX_PRODUCT

# Increase for lower FP and higher precision, yielding fewer files.
# Decrease for lower FN and higher recall, yielding more files.
THRESHOLD      = args.threshold      # Default is 0.50

LABEL_NEGATIVE = 0
LABEL_POSITIVE = 1

CHARMAP        = {
  "A":64, "N":13, "a":26, "n":39, "0":52,
  "B":1,  "O":14, "b":27, "o":40, "1":53,
  "C":2,  "P":15, "c":28, "p":41, "2":54,
  "D":3,  "Q":16, "d":29, "q":42, "3":55,
  "E":4,  "R":17, "e":30, "r":43, "4":56,
  "F":5,  "S":18, "f":31, "s":44, "5":57,
  "G":6,  "T":19, "g":32, "t":45, "6":58,
  "H":7,  "U":20, "h":33, "u":46, "7":59,
  "I":8,  "V":21, "i":34, "v":47, "8":60,
  "J":9,  "W":22, "j":35, "w":48, "9":61,
  "K":10, "X":23, "k":36, "x":49, "+":62,
  "L":11, "Y":24, "l":37, "y":50, "/":63,
  "M":12, "Z":25, "m":38, "z":51, "=":0,
```

```
}

def load_data(data, dir, label=LABEL_NEGATIVE):
  def process(path):
    # Normalize decimal numbers.
    def scale(X):
      return [(i - min(X)) / (max(X) - min(X)) * FEATURE_RANGE_MAX for i in X]

    # Yield successive n-sized chunks from a list.
    def chunks(lst, n):
      return [lst[i:i + n] for i in range(0, len(lst), n)]

    with open(path, 'rb') as f:
      data = f.read()
    # Encode file content via Base64 and map each byte using CHARMAP.
    encoded = [CHARMAP.get(chr(i), 0) for i in b64encode(data)]

    # Group every 6 numbers into a single decimal feature.
    decimal = [
      np.prod(np.array(chunk)[np.array(chunk) != 0]) if ENCODING == 'product'
      else int(''.join(map(str, chunk)))
      for chunk in chunks(encoded, 6)
    ]

    # Pad the feature list to 256 elements.
    while len(decimal) < 256:
      decimal.append(0)
    return decimal

  good, bad = [], []

  logging.info(f'Processing directory {os.path.join(os.getcwd(), dir)} containing
    samples for label {label}')
  for file in [f for f in os.listdir(os.path.join(os.getcwd(), dir)) if f.lower().
    endswith('.emf')]:
    features = process(os.path.join(os.getcwd(), dir, file))
    features.insert(0, label)
    if len(features) == 257: # Ignore too big files
      good.append(len(features))
      data.update({file:features})
    else:
      bad.append(len(features))
      logging.warning(f'Ignoring {file} which has {len(features)} features and is {os.
    path.getsize(os.path.join(os.getcwd(), dir, file))} bytes')

  total = len(good) + len(bad)
  ratio = (len(bad) / total * 100) if total > 0 else 0
  logging.info(f'Good: {len(good)} - Bad: {len(bad)} - Ratio: {ratio:.2f}%')
  return data

def main():
  dd = {}
  dd = load_data(dd, args.negative, LABEL_NEGATIVE)
  dd = load_data(dd, args.positive, LABEL_POSITIVE)

  df = pd.DataFrame.from_dict(dd, orient='index').add_prefix('f')

```

```
X = df.drop('f0', axis=1).values
y = df['f0'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True)

scaler = MinMaxScaler(feature_range=(0, FEATURE_RANGE_MAX))
X_train = pd.DataFrame(scaler.fit_transform(X_train)).values
X_test = pd.DataFrame(scaler.transform(X_test)).values

model = Sequential()
model.add(Input(shape=(256,)))
model.add(Dense(256, activation='relu', kernel_regularizer=L2(0.00001) if
  REGULARIZATION else None))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu', kernel_regularizer=L2(0.00001) if
  REGULARIZATION else None))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(
  optimizer='adam',
  loss='binary_crossentropy',
  metrics=['accuracy',
    Recall(thresholds=THRESHOLD, name='recall'),
    Precision(thresholds=THRESHOLD, name='precision'),
  ]
)

kfold = KFold(n_splits=4, shuffle=True)
for train, test in kfold.split(X_train, y_train):
  model.fit(
    x=X_train[train],
    y=y_train[train],
    validation_data=(X_train[test], y_train[test]),
    callbacks=[
      EarlyStopping(monitor='val_loss', mode='min', restore_best_weights=True),
    ]
  )

scores = model.evaluate(X_test, y_test)
logging.info(f'Loss: {scores[0] * 100:.2f}% - Recall: {scores[2] * 100:.2f}% -
  Precision: {scores[3] * 100:.2f}%')

dd = {}
if args.unknown:
  dd = load_data(dd, args.unknown)

df = pd.DataFrame.from_dict(dd, orient='index').add_prefix('f')

X_pred = df.drop('f0', axis=1).values
X_pred = pd.DataFrame(scaler.transform(X_pred))

preds = (model.predict(X_pred) > THRESHOLD).astype('int32')
indices = []

if np.count_nonzero(preds) == 0:
  consent = input('No positive predictions found. Automatically adjust threshold? (
```

```
    yes/no): ').strip().lower()
160   if consent in ['yes', 'y']:
161     best = []
162     uniques = set()
163
164     preds = model.predict(X_pred)
165
166     flats = [item for sublist in preds for item in sublist]
167     for v in heapq.nlargest(len(flats), flats):
168       if v not in uniques:
169         uniques.add(v)
170         best.append(v)
171       if len(best) == 20:
172         break
173
174     indices = [flats.index(v) for v in best]
175
176     preds = (preds > min(best)).astype('int32')
177
178 lst = list()
179 for idx, l in np.ndenumerate(preds):
180   if l == 1:
181     if indices and idx[0] not in indices:
182       continue
183     lst.append(df.index[idx[0]])
184   if len(lst) == 20:
185     break
186
187 for filename in lst:
188   logging.info(filename)
```

Listing 28: Main functionality of the `SeedClass` prototype

F. Helper scripts

```python
import os, re, shutil, hashlib, argparse

def calculate_hash(path):
  with open(path, 'rb') as file:
    data = file.read()
  return hashlib.sha256(data).hexdigest().upper()

def is_valid_crash(filename):
  return re.match(r'^id_\d{6}_[0-9A-Fa-f]{8}_.+$', filename) is not None

def process_folder(input_path, output_path, extension):
  for root, dirs, files in os.walk(input_path):
    if 'crashes' in dirs:
      crashes = os.path.join(root, 'crashes')
      for crash in os.listdir(crashes):
        if is_valid_crash(crash):
          source = os.path.join(crashes, crash)
          hash = calculate_hash(source)
          destination = os.path.join(output_path, f'{hash}{extension}')
          shutil.copy2(source, destination)

      bucket = os.path.join(crashes, 'bucket')
      if os.path.exists(bucket):
        for crash in os.listdir(bucket):
          if is_valid_crash(crash):
            source = os.path.join(bucket, crash)
            hash = calculate_hash(source)
            destination = os.path.join(output_path, f'{hash}{extension}')
            shutil.copy2(source, destination)

def main():
  parser = argparse.ArgumentParser(prog='SeedCollect')
  parser.add_argument('-i', '--input', required=True)
  parser.add_argument('-o', '--output', required=True)
  parser.add_argument('-e', '--extension', type=str, default='.emf')
  args = parser.parse_args()

  os.makedirs(args.output, exist_ok=True)
  process_folder(args.input, args.output, args.extension)
```

Listing 29: `CrashCollect` - Gather unique crash samples from output directories

```python
import os, re, shutil, argparse, threading, subprocess, concurrent.futures
from pathlib import Path

def run_bugid(file, files, crash_dir, clean_dir, lock):
  cmd = [
    'BugId.cmd', '-q', '--collateral=1', '--isa=x86', 'GDI.exe', '--', '-f', file
  ]

  proc = subprocess.run(cmd, capture_output=True, text=True, encoding='utf-8', errors=
    'replace')
  if 'A bug was detected' in (stdout := proc.stdout or ''):
    if (match := re.search(r'Id @ Location:\s+\S+\s+(\w+\.\w+) @ [^!]+!(\w+\.\w+)!(.+)
    ', stdout)):
      bugid, library, function = match.groups()
      crashes = crash_dir / Path(file).name
      shutil.move(file, crashes)
      with lock:
        if bugid not in files or Path(file).stat().st_size < files[bugid][1]:
          files[bugid] = (crashes, Path(file).stat().st_size, function, library)
  else:
    shutil.move(file, clean_dir / Path(file).name)

def process_files(input_dir, crash_dir, clean_dir):
  files, lock = {}, threading.Lock()
  with concurrent.futures.ThreadPoolExecutor() as executor:
    executor.map(lambda f: run_bugid(input_dir / f, files, crash_dir, clean_dir, lock)
    , os.listdir(input_dir))
  return files

def copy_unique_files(files, unique_dir):
  [shutil.copy(file, unique_dir / Path(file).name) for file, *_ in files.values()]

def main():
  parser = argparse.ArgumentParser(prog='SeedSort')
  parser.add_argument('-i', '--input', required=True, type=Path)
  parser.add_argument('-o', '--output', required=True, type=Path)
  args = parser.parse_args()

  crash_dir, unique_dir, clean_dir = (args.output / sub for sub in ['all', 'unique', '
    none'])
  [d.mkdir(parents=True, exist_ok=True) for d in [crash_dir, unique_dir, clean_dir]]

  unique_bugs = process_files(args.input, crash_dir, clean_dir)
  copy_unique_files(unique_bugs, unique_dir) if unique_bugs else None
```

Listing 30: `CrashSort` - Triage crash samples using BugID

```python
import os, re, shutil, argparse
from collections import defaultdict

def process_log(path, input_dir, output_dir=None, distribution_size=100):
  samples = defaultdict(list)
  bugs = {}
  optimized_counts = defaultdict(int)

  pattern = re.compile(
    r'File (\S+) with (\d+) bytes triggered bug (\S+) in (\S+) within (\S+)\.'
  )

  with open(path, 'r') as file:
    for line in file:
      match = pattern.search(line)
      if match:
        sample, _, id, function, library = match.groups()
        samples[id].append(sample)
        bugs[id] = (function, library)

  os.makedirs(output_dir, exist_ok=True)

  for id, files in samples.items():
    total_count = len(files)
    target_count = min(distribution_size, total_count)
    copied_files = 0

    for file in files[:target_count]:
      src_path = os.path.join(input_dir, file)
      dest_path = os.path.join(output_dir, file)
      if os.path.exists(src_path):
        shutil.copy(src_path, dest_path)
        optimized_counts[id] += 1
        copied_files += 1
        if copied_files >= target_count:
          break

def main():
  parser = argparse.ArgumentParser(prog='SeedDistribute')
  parser.add_argument('-l', '--log', required=True)
  parser.add_argument('-i', '--input', required=True)
  parser.add_argument('-o', '--output')
  parser.add_argument('-s', '--size', type=int, default=100)
  args = parser.parse_args()

  process_log(args.log, args.input, args.output, args.size)
```

Listing 31: `CrashDistribute` - Optimize bug distribution for a balanced dataset

```python
import os, shutil, argparse
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor

def calculate_variance(data1, data2):
  if len(data1) != len(data2): return 100
  return (sum(abs(b1 - b2) for b1, b2 in zip(data1, data2)) / (len(data1) * 255)) *
    100

def group_files_by_size(input_dir):
  size_groups = defaultdict(list)
  for root, _, files in os.walk(input_dir):
    for file in files:
      size_groups[os.path.getsize(path := os.path.join(root, file))].append(path)
  return size_groups

def filter_files_with_variance(group, threshold=30):
  filtered = []
  for file1 in group:
    data1 = open(file1, 'rb').read()
    if all(calculate_variance(data1, open(file2, 'rb').read()) >= threshold for file2
    in filtered):
      filtered.append(file1)
  return filtered

def process_size_group(size, group, variance, output_dir):
  for file in filter_files_with_variance(group, threshold=variance): shutil.copy(file,
     output_dir)

def main():
  parser = argparse.ArgumentParser(prog='SeedFilter')
  parser.add_argument('-i', '--input', required=True)
  parser.add_argument('-o', '--output', required=True)
  parser.add_argument('-v', '--variance', type=float, default=30)
  args = parser.parse_args()

  os.makedirs(args.output, exist_ok=True)

  with ThreadPoolExecutor() as executor:
    for size, files in group_files_by_size(args.input).items():
      executor.submit(process_size_group, size, files, args.variance, args.output)
```

Listing 32: `SeedFilter` - Optimize valid samples based on byte-level variance

```python
import os, random, shutil, argparse

def create_random_groups(input_dir, output_dir, num_files, num_groups):
  files = [f for f in os.listdir(input_dir) if os.path.isfile(os.path.join(input_dir,
    f))]
  count = len(files)

  needed = num_groups * num_files

  if count < needed:
    needed = count
    num_files = min(num_files, count)

  random.shuffle(files)
  selected = random.sample(files, needed)

  os.makedirs(output_dir, exist_ok=True)

  groups = [selected[i:i + num_files] for i in range(0, len(selected), num_files)]

  if num_groups == 1:
    for file in groups[0]:
      shutil.copy(os.path.join(input_dir, file), os.path.join(output_dir, file))
  else:
    for i, group in enumerate(groups, 1):
      dir_path = os.path.join(output_dir, f'group{i}')
      os.makedirs(dir_path, exist_ok=True)
      for file in group:
        shutil.copy(os.path.join(input_dir, file), os.path.join(dir_path, file))

def main():
  parser = argparse.ArgumentParser(prog='SeedGroup')
  parser.add_argument('-i', '--input', required=True)
  parser.add_argument('-o', '--output', required=True)
  parser.add_argument('-s', '--size', type=int, required=True)
  parser.add_argument('-g', '--groups', type=int, required=True)
  args = parser.parse_args()

  create_random_groups(args.input, args.output, args.size, args.groups)
```

Listing 33: `SeedGroup` - Generate random, shuffled sample groups

```python
import pyemf3, random, hashlib, argparse
from pathlib import Path

def calculate_hash(path):
  return hashlib.sha256(Path(path).read_bytes()).hexdigest().upper()

def save_emf(records, output, extension, index):
  pyemf3.EMF(width=random.randint(0, 12), height=random.randint(0, 6), records=records
    ).save(output / f'id_{index:06d}{extension}')

def filter_emf(output, extension):
  for file in sorted(output.glob(f'*{extension}')):
    (file.unlink() if (new_path := output / f'{calculate_hash(file)}{extension}').
    exists() else file.rename(new_path))

def split_emf(input_file, output, extension, pieces):
  emf = pyemf3.EMF(verbose=False).load(input_file)
  records, index = [], 1
  for record in emf.records:
    if record.iType != 1: records.append(record)
    if len(records) == pieces:
      save_emf(records, output, extension, index)
      index, records = index + 1, []
  if records: save_emf(records, output, extension, index)
  filter_emf(output, extension)

def process_emf(file_path, output_base, extension, pieces):
  checksum, file_dir = calculate_hash(file_path), Path(file_path).parent
  output_folder = file_dir / checksum if file_dir == Path(output_base) else Path(
    output_base)
  output_folder.mkdir(exist_ok=True)

  split_emf(file_path, output_folder, extension, pieces)

  total = len(list(output_folder.glob(f'*{extension}')))
  new_name = f'{checksum}{"discarded" if total < 10 else "splitted"}{extension}'
  Path(file_path).rename(file_dir / new_name)

def main():
  parser = argparse.ArgumentParser(prog='SeedSplit')
  parser.add_argument('-i', '--input', type=Path, required=True)
  parser.add_argument('-o', '--output', type=Path, default=Path('output'))
  parser.add_argument('-e', '--extension', type=str, default='.emf')
  parser.add_argument('-p', '--pieces', type=int, default=20)
  args = parser.parse_args()

  files = args.input.glob('*') if args.input.is_dir() else [args.input]
  list(map(lambda f: process_emf(f, args.output, args.extension, args.pieces), files))
```

Listing 34: `SeedSplit` - Divide large samples into smaller pieces

```python
import os, time, shutil, hashlib, logging, argparse, concurrent.futures
from pathlib import Path
from pywinauto.application import Application
from pywinauto.findwindows import ElementNotFoundError

MAX_RETRIES, RETRY_DELAY = 3, 2

def calculate_hash(path):
  return hashlib.sha256(Path(path).read_bytes()).hexdigest().upper()

def is_emf_valid_via_ui(file_path):
  try:
    app = Application(backend='uia').start(f'mspaint "{file_path}"', wait_for_idle=
    False)
    return not app.window(title='Paint cannot read this file.', enabled_only=True).
    exists()
  except ElementNotFoundError:
    return True
  finally:
    if app: app.kill()

def process_file(filename, args):
  src = args.input / filename
  if not is_emf_valid_via_ui(src):
    logging.warning(f'{filename} is NOT valid!')

  dst = args.output / f'{calculate_hash(src)}{args.extension}'
  for _ in range(MAX_RETRIES):
    try:
      if src.exists() and os.access(src, os.R_OK | os.W_OK):
        shutil.move(src, dst)
        logging.info(f'{filename} is VALID.')
      time.sleep(RETRY_DELAY)
    except Exception:
      time.sleep(RETRY_DELAY)

def main():
  parser = argparse.ArgumentParser(prog='SeedValidate')
  parser.add_argument('-i', '--input', type=Path, required=True)
  parser.add_argument('-o', '--output', type=Path, required=True)
  parser.add_argument('-e', '--extension', type=str, default='.emf')
  args = parser.parse_args()

  args.output.mkdir(exist_ok=True)
  filenames = [f for f in args.input.glob(f'*{args.extension}') if f.is_file()]

  with concurrent.futures.ThreadPoolExecutor() as executor:
    list(executor.map(lambda f: process_file(f.name, args), filenames))
```

Listing 35: `SeedValidate` - Verify valid samples by opening them in Paint