DEPARTMENT OF
COMPUTER SCIENCE

RICARDO JORGE MATOS BESSA

BSc in Computer Science

REDSHELL: A GENERATIVE AI-BASED APPROACH TO ETHICAL HACKING

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
November, 2025

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

REDSHELL: A GENERATIVE AI-BASED APPROACH TO ETHICAL HACKING

RICARDO JORGE MATOS BESSA

BSc in Computer Science

Adviser: Rui Nuno Lopes Claro
Security Consultant, Layer8

Co-advisers: João Manuel dos Santos Lourenço
Associate Professor, NOVA University Lisbon

João Henrique Correia de Almeida Lourenço Trindade
RnD Unit Manager, Layer8

Examination Committee

Chair: Ricardo João Rodrigues Gonçalves
Assistant Professor, FCT-NOVA

Rapporteur: Miguel Filipe Leitão Pardal
Assistant Professor, IST-UL

Member: Rui Nuno Lopes Claro
Security Consultant, Layer8

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
November, 2025

RedShell: A Generative AI-Based Approach to Ethical Hacking

Copyright © Ricardo Jorge Matos Bessa, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.2.2) [1].

To my parents, João and Manuela, and my sister, Margarida,
for their unconditional support and belief in me.

Acknowledgements

This thesis would not have been possible without the dedication and continuous support of many extraordinary people, to whom I wish to express my sincere gratitude. I would like to begin by thanking my advisors Rui Claro, João Trindade, and João Lourenço for their remarkable contributions to the work presented in this thesis.

My journey with Layer8 began two years ago when João Trindade offered me the unique opportunity to join the R&D team for a curricular internship, where I discovered my passion for distributed systems and cybersecurity. At that time, I also briefly met Rui, never imagining that he would later become my master's thesis advisor and someone I would greatly enjoy working with. I am truly grateful to both Rui and João Trindade for their trust, for granting me the freedom to explore my own ideas, and for supporting me in accomplishing a research work that we could all be proud of. I cherish the time I spent at Layer8, alongside Daniel Gonçalves and all the members of the R&D team.

I extend my heartfelt gratitude to Professor João Lourenço. Although I had not known Professor João previously, I now consider myself fortunate to have crossed paths with him. His dedication, availability, and insightful guidance were crucial in the production of this thesis. I have been deeply inspired by his commitment and his willingness to help others. To all my advisors, I am equally thankful for their guidance in achieving an important milestone in my academic journey, the publication of my first scientific article.

I would also like to acknowledge the Department of Informatics of NOVA University of Lisbon and the NOVA LINCS research centre, both of which have played a crucial role in my academic development. In particular, I am grateful to Professor João Leitão and the colleagues from the Computer Systems Group for their warm welcome and continuous guidance over the past months. I also acknowledge the support of UID/04516/NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) with the financial support of FCT.IP, which allowed me to present my research findings at INForum 2025.

Finally, I wish to thank my colleagues and friends for their positive influence and belief in me. To my family, I am deeply grateful for their patience and support. My parents, João Bessa and Maria Manuela Bessa, and my sister, Margarida Bessa, have always stood by me, and this work would not have been possible without them.

Abstract

The application of Machine Learning techniques in code generation is now a common practice for most developers. Tools such as ChatGPT from OpenAI leverage the natural language processing capabilities of Large Language Models to generate machine code from natural language descriptions. In the cybersecurity field, red teams can also take advantage of generative models to build malicious code generators, providing more automation to pentest audits. However, the application of Large Language Models in malicious code generation remains challenging due to the lack of data to train and evaluate offensive code generators. In this work, we propose RedShell, a tool that allows ethical hackers to generate malicious PowerShell code. We also introduce a ground truth dataset, combining publicly available code samples to fine-tune models in malicious PowerShell generation. Our experiments demonstrate the strong capabilities of RedShell in generating syntactically valid PowerShell, with over 90% of the generated samples successfully parsed without errors. Furthermore, our specialized model was able to produce samples that were semantically consistent with reference snippets, achieving a competitive performance on standard output similarity metrics such as edit distance and METEOR, with their similarity scores exceeding 50% and 40%, respectively. We also conducted a functional evaluation of the snippets generated by our tool, emphasizing their strong effectiveness in a wide range of offensive cybersecurity operations. This work sheds light on the state-of-the-art research in the field of Generative AI applied to pentesting and also serves as a steppingstone for future advancements, highlighting the potential benefits these models hold within such controlled environments.

Keywords: Cybersecurity, Ethical Hacking, Pentesting, Large Language Models

Resumo

A aplicação de técnicas de Aprendizagem Automática na geração de código é atualmente uma prática comum entre programadores. Ferramentas como o ChatGPT da OpenAI tiram partido das capacidades de processamento de linguagem natural dos Modelos de Linguagem de Larga Escala para gerar código a partir de descrições em linguagem natural. No domínio da cibersegurança, as *red teams* podem também beneficiar de modelos generativos para construir geradores de código malicioso, oferecendo a possibilidade de realizar testes de intrusão de forma mais automática. No entanto, a aplicação de Modelos de Linguagem de Larga Escala na geração de código malicioso continua a ser um desafio, devido à escassez de dados para treinar e avaliar geradores de código ofensivo. Neste trabalho, propomos uma nova ferramenta chamada RedShell para auxiliar hackers éticos na geração de código PowerShell malicioso. Introduzimos também um conjunto de dados de referência, combinando amostras de código disponíveis publicamente, para especializar modelos na geração de PowerShell malicioso. As nossas experiências demonstram as fortes capacidades exibidas pelo RedShell na geração de PowerShell sintacticamente válido, com mais de 90% de amostras geradas sem erros de *parsing*. Adicionalmente, o nosso modelo especializado foi capaz de produzir amostras semanticamente consistentes com os dados de referência, alcançando um desempenho competitivo em métricas de distância standard, como a distância de edição e METEOR, com pontuações de similaridade superiores a 50% e 40%, respetivamente. Foi também realizada uma avaliação funcional das amostras de código geradas pela nossa ferramenta, assinalando a sua notável eficácia em diversas operações de cibersegurança ofensiva. Este trabalho realça o estado-da-arte da investigação na área da Inteligência Artificial Generativa aplicada a testes de intrusão, servindo também como referência para trabalhos futuros, ao destacar os potenciais benefícios que estes modelos podem oferecer em ambientes controlados.

Palavras-chave: Cibersegurança, Hacking Ético, Testes de Intrusão, Modelos de Linguagem de Larga Escala

Contents

List of Figures ix

List of Tables x

Glossary xi

Acronyms xvi

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 2

1.3 Thesis Structure 4

2 Background and Related Work 5

2.1 Traditional Pentesting 5

2.1.1 Phases 5

2.1.2 Scenarios 6

2.1.3 Strategies 7

2.1.4 The MITRE ATT&CK Framework 8

2.1.5 Exploitation Techniques 10

2.1.6 Red Teaming Tools 11

2.2 Large Language Models 13

2.2.1 Pattern Recognition 13

2.2.2 Machine Learning 14

2.2.3 Neural Networks 15

2.2.4 Deep Learning 16

2.2.5 Natural Language Processing 17

2.2.6 The Transformer Architecture 18

2.2.7 First Large Language Model 19

2.2.8 Code Generators 20

2.3 Cybersecurity and Generative AI . . . 21
2.3.1 Defensive Cybersecurity . . . 21
2.3.2 Offensive Cybersecurity . . . 22
2.3.3 Ethical Concerns . . . 23
2.3.4 AI-based Solutions for Pentesting . . . 24
2.3.5 State of the Art in LLM-Based Malicious Code Generation . . . 25
2.4 Summary . . . 29

3 The Ground Truth Dataset **30**
3.1 Reference Dataset . . . 30
3.1.1 MITRE ATT&CK Coverage . . . 30
3.1.2 Limitations . . . 31
3.2 Extended Dataset . . . 32
3.2.1 MITRE ATT&CK Coverage . . . 32
3.2.2 Improvements . . . 34
3.3 Summary . . . 35

4 RedShell Design **36**
4.1 Overview . . . 36
4.2 Models . . . 37
4.3 Fine-Tuning . . . 38
4.3.1 LoRA Settings . . . 38
4.3.2 Training Settings . . . 39
4.3.3 Dataset Partitions . . . 40
4.3.4 Computational Performance . . . 41
4.4 Summary . . . 42

5 Evaluation **43**
5.1 Syntactic Assessment . . . 43
5.1.1 Evaluation Metrics . . . 43
5.1.2 Experimental Results . . . 44
5.1.3 Fine-Tuning Impact . . . 46
5.2 Semantic Assessment . . . 47
5.2.1 Evaluation Metrics . . . 47
5.2.2 Experimental Results . . . 48
5.2.3 Comparison with State-of-the-Art Models . . . 50
5.2.4 Comparison with Proprietary Models . . . 51
5.2.5 Fine-Tuning Impact . . . 53
5.2.6 Extended Ground Truth Dataset Impact . . . 54
5.3 Functional Assessment . . . 55
5.3.1 Controlled Environment . . . 55
5.3.2 Pentesting Methodology . . . 58

5.3.3 Evaluation Metrics 61
5.3.4 Experimental Results 63
5.3.5 Comparison with a Proprietary Model 64
5.3.6 Fine-Tuning Impact 66
5.4 Summary 68

6 Conclusions and Future Work **69**
6.1 Conclusions 69
6.2 Future Work 70

Bibliography **71**

List of Figures

2.1 Pentesting scenarios as testing boxes. 6
2.2 Perceptron mathematical representation. 15
2.3 Transformer model architecture. 18
2.4 BERT model architecture. 19

3.1 Reference ground truth dataset coverage of MITRE ATT&CK offensive tactics. 31
3.2 Ground truth dataset coverage of MITRE ATT&CK offensive tactics. 32

4.1 RedShell design overview. 36
4.2 LoRA settings for the fine-tuning processes. 39
4.3 Training settings for the fine-tuning processes. 40
4.4 Peak reserved VRAM while fine-tuning models with the reference dataset. . 41

5.1 Syntactic evaluation of models fine-tuned with the reference dataset. 44
5.2 Syntax report of Qwen2.5-Coder fine-tuned with the reference dataset. . . . 45
5.3 Syntactic evaluation of the base and fine-tuned versions of Qwen2.5-Coder. 46
5.4 Semantic evaluation of models fine-tuned with the reference dataset. 48
5.5 ED score distribution of models fine-tuned with the reference dataset. . . . 48
5.6 ED score distribution of Qwen2.5-Coder fine-tuned with the reference dataset. 49
5.7 Semantic evaluation of Qwen2.5-Coder and reference models. 51
5.8 ED score distribution of Qwen2.5-Coder and reference models. 51
5.9 Semantic evaluation of Qwen2.5-Coder and closed models. 52
5.10 ED score distribution of Qwen2.5-Coder and closed models. 52
5.11 Semantic evaluation of the base and fine-tuned versions of Qwen2.5-Coder. 53
5.12 ED score distribution of the base and fine-tuned versions of Qwen2.5-Coder. 53
5.13 Semantic evaluation of Qwen2.5-Coder fine-tuned with different datasets. . 54
5.14 ED score distribution of Qwen2.5-Coder fine-tuned with different datasets. 54
5.15 Controlled environment setup for the functional evaluation. 56
5.16 Offensive pipeline for the functional evaluation. 59

List of Tables

2.1 MITRE ATT&CK framework overview. 9
2.2 Pentesting tools overview. 12
2.3 Evolution of LLMs in code generation. 20
2.4 Recent contributions to the research on AI-based pentesting approaches. . . 24
2.5 Malicious code generators overview. 26

3.1 Data sources of the new collected PowerShell samples. 33
3.2 Snippets from the extended ground truth dataset. 35

4.1 Training time of models fine-tuned with the reference dataset. 41

5.1 Illustrative syntactic flaws in the snippets generated by Qwen2.5-Coder. . . 46
5.2 Illustrative snippets generated by RedShell with high semantic correctness. 50
5.3 Flags and tactics employed in the functional evaluation. 62
5.4 Functional evaluation of RedShell. 63
5.5 Functional evaluation of ChatGPT-3.5 and RedShell. 64
5.6 Functional evaluation of the non-effective code from ChatGPT-3.5 and RedShell. 65
5.7 Functional evaluation of the the base version of Qwen2.5-Coder and RedShell. 66
5.8 Functional evaluation of the non-effective code from the base version of Qwen2.5-Coder and RedShell. 67

Glossary

Antivirus A program designed to detect and remove viruses and other kinds of malicious software from computers. *(pp. 11, 12, 57, 60, 61, 63, 64, 66)*

Blue Team A team responsible for monitoring and protecting a system or network from cyber threats. *(p. 21)*

Buffer Overflow A software flaw that occurs when the volume of data in a buffer exceeds its capacity. *(p. 10)*

Client-Side Attacks Attacks that explore vulnerabilities in client-side applications such as web browsers. *(p. 7)*

Closed Model A model whose internal architecture, training data, and code are not publicly available. *(pp. 37, 38, 52, 64)*

Convolution A mechanism used by convolutional neural networks, leveraging three-dimensional data for image classification and object recognition tasks. *(p. 18)*

Credential Theft The illicit acquisition of usernames and passwords, often with the intent to gain unauthorized access to computer systems. *(pp. 8, 11)*

Cybersecurity The practice of protecting systems and networks from digital attacks. *(pp. 1, 2, 5, 8, 10, 12, 21, 23, 24, 26, 28, 30, 33, 69, 70)*

Data Classification — The practice of classifying data into different categories. *(pp. 14, 15)*

Deep Learning — The ability of models to learn by decomposing complex concepts into simple concepts. *(pp. 4, 13, 16, 17)*

Defensive Cybersecurity — The practice of protecting systems, networks, and data from attacks by implementing security measures, monitoring threats, and responding to incidents. *(p. 21)*

Epoch — A complete period of training a model with a training set. *(pp. 13, 41)*

Ethical Hacking — An authorized attempt to gain access to a computer system, application, or data by emulating real attacks. *(pp. 23, 26, 30, 58)*

Exfiltration — An unauthorized data transfer. *(pp. 8, 11)*

Exploit — Software that takes advantage of a bug or vulnerability to cause unintended or unanticipated behavior to occur on computer software or hardware. *(pp. 10, 12, 22, 25–27, 29)*

Extranet — A private network that enterprises provide to trusted third parties. *(p. 7)*

Fine-Tuning — A learning process that provides specific knowledge to a pre-trained model. *(pp. 1, 13, 19, 20, 22, 23, 25–28, 36, 38–42, 46, 47, 53, 66, 67, 69)*

Firewall — A network security device that monitors traffic to or from a network, blocking traffic based on a defined set of security rules. *(pp. 7, 9, 11, 26, 28, 34)*

Generative AI AI models that produce original responses as output. *(pp. 2, 4, 5, 13, 17, 21, 23)*

Intrusion Detector Device or software application that monitors a network for malicious activity or policy violations. *(pp. 11, 21)*

Jailbreaking A Chain-of-Thought prompting approach where the instructions to models are modified to go beyond their ethical limits. *(p. 23)*

Lateral Movement A process to move deeper into a network in search of sensitive data and other high-value assets. *(pp. 8, 11)*

Malware A malicious software developed by cyber criminals to steal data and damage or destroy computers. *(pp. 11, 23, 34)*

Network Enumeration A process to identify the hosts in a given network. *(p. 7)*

Neural Network An adaptive model composed of multiple layers of logistic regression models. *(pp. 13, 15–17)*

Obfuscation An encoding process of malicious code to avoid detection. *(pp. 8, 11)*

Offensive Cybersecurity The practice of proactively testing, exploiting, or simulating attacks on systems and networks to identify vulnerabilities and strengthen defenses. *(pp. 1–3, 5, 22, 29, 64, 68–70)*

Over-Fitting An accuracy problem of predictive models that fit perfectly the training data but provide less accurate predictions of new data. *(p. 15)*

Pattern Recognition The automatic discovery of regularities in data through the use of computer algorithms. *(pp. 5, 13–16)*

Pentesting Simulated cyber attack used to identify and exploit vulnerabilities in a computer system, network, or application by employing the same tools and techniques as a malicious attacker. *(pp. 1–6, 10–12, 21, 22, 24, 25, 28–36, 38, 40, 43, 55–59, 61–64, 68, 69)*

Perceptron The base unit of a neural network, composed of a simpler adaptive model. *(pp. 15, 16)*

Phishing An attack that attempts to trick a user into giving up sensitive information by assuming a trustful identity or source. *(pp. 9, 11, 21, 23)*

Pivoting A process to gain access to a system that provides access to an entire network. *(pp. 8, 11)*

Pre-Training The initial learning phase of a model on a large and diverse dataset. *(pp. 13, 20, 22, 25, 26, 28)*

Prompt Engineering The methodology on how to design a prompt. *(pp. 22, 23)*

Reconnaissance A process to collect information about a target system. *(p. 5)*

Recurrence A mechanism used by recurrent neural networks, where outputs from previous steps are fed back as inputs to process sequential or temporal data. *(p. 18)*

Red Team A team responsible for testing the cybersecurity effectiveness of a system through simulated cyber attacks. *(pp. 1, 3, 5, 8, 12, 21, 22, 35)*

Regression A data classification task with an output that consists of one or more continuous variables. *(pp. 14, 15)*

Reverse Psychology The psychological manipulation of prompts to cross the ethical boundaries of a model. *(pp. 23, 24)*

Scanning	The use of scanners to detect flaws in a target system. *(p. 5)*
Shellcode	A small piece of code used as the payload in the exploitation of a software vulnerability. *(pp. 10, 11, 26–28)*
Signatures	A collection of attack patterns. *(p. 11)*
Social-Engineering	A group of malicious activities accomplished through human interactions. *(pp. 7, 11, 23, 29)*
Spearphishing	A phishing campaign targeting a specific organization or a group of individuals. *(pp. 8, 9, 11)*
Transformer	A model architecture that replaces convolution and recurrence of neural networks with an attention mechanism. *(pp. 1, 2, 13, 18, 19, 21, 26, 27)*
Use-After-Free	A security flaw of programs with poor heap memory management. *(p. 10)*
Vulnerability Assessment	A process to identify core vulnerabilities on a system. *(pp. 5, 70)*
Zero-Day Exploit	A vulnerability unpatched by the software publishers. *(p. 10)*

Acronyms

AI Artificial Intelligence *(pp. 1–5, 10, 13, 16, 21–24, 29, 37, 69, 70)*

BERT Bidirectional Encoder Representations from Transformers *(pp. 19–21)*

CNN Convolutional Neural Network *(pp. 17, 18)*

CTF Capture The Flag *(pp. 25, 58)*

ED Edit Distance *(pp. 47–54, 61)*

GAN Generative Adversarial Network *(pp. 26, 28)*

IT Information Technology *(p. 7)*

LLM Large Language Model *(pp. 1–5, 10, 13, 19–29, 31, 36–39, 41–44, 47–53, 57, 62, 65–67, 69, 70)*

LoRA Low Rank Adaption Method *(pp. 38, 39, 41, 42, 51)*

ML Machine Learning *(pp. 13–16)*

NLP Natural Language Processing *(pp. 17, 19)*

NMT Neural Machine Translation *(pp. 17, 18)*

PEFT Parameter-Efficient Fine-Tuning *(p. 38)*

RAG Retrieval-Augmented Generation *(p. 23)*

RNN Recurrent Neural Network *(pp. 17, 18)*

XSS Cross Site Scripting *(pp. 10, 28)*

1

Introduction

In recent years, Large Language Models (LLMs) sparked a revolution in natural language processing through advanced transformers [2]. Generative models such as ChatGPT [3] have demonstrated strong capabilities while performing generic tasks [4]. In addition, fine-tuning techniques have been employed to adapt LLMs to address more specific challenges, with Generative Artificial Intelligence (AI) becoming pervasive in the workflow of many software engineering tasks [5].

In the offensive cybersecurity field, ethical hackers can also take advantage of LLMs to develop malicious code generators, providing more automation to pentest audits. Penetration testing, often referred to as pentesting, is a crucial activity for red teams, which are responsible for testing the cybersecurity effectiveness of a system through simulated cyber attacks [6]. By detecting potential security lapses in a target system, pentesters are able to prevent attacks that could cause harm to that system and its users.

Ethical hackers typically use malicious code to identify and explore security vulnerabilities, assessing what real attackers could achieve after a successful intrusion. However, regardless of the exploitation technique being employed, writing offensive code requires time and technical knowledge from the pentesters. While the desired automation could be provided by LLMs, this strategy also remains challenging due to the lack of data to train and evaluate offensive code generators.

1.1 Motivation

The work presented in this thesis was developed in the context of a research project conducted within the R&D team from Layer8, a Portuguese company focused on information security, privacy, and compliance management services and solutions. The project proposal was motivated by the study of potential applications of state-of-the-art AI-based techniques in the computer security field. In particular, the R&D team aimed to employ small and local fine-tuned LLMs to enhance the online security of applications in cyberspace while providing more automation to traditional cybersecurity workflows. All the experiments were conducted on the local hardware infrastructure provided by Layer8.

In more detail, the work presented in this thesis was motivated by the following:

- Addressing the lack of offensive data to support both the training and evaluation of LLMs in malicious PowerShell code generation.
- Explore state-of-the-art AI-based approaches for malicious PowerShell code generation through advanced transformers and generative models, and start discussions around ethical AI and malicious AI usage.
- Develop tools and frameworks to prevent misuse in real-world applications, acknowledging the potential benefits that LLMs hold within controlled environments such as pentesting, and the significant impact of Generative AI in the computer security field.
- Build novel solutions to assist pentesters in conducting a wide range of offensive cybersecurity operations with more automation while following reference pentesting techniques.
- Assess the generation capabilities and measure the output correctness of different LLM-based solutions for exploiting security vulnerabilities in Microsoft Windows devices through the execution of malicious PowerShell snippets.

1.2 Contributions

In this thesis, we propose RedShell, an AI-based malicious PowerShell generator designed to provide more automation to pentesting activities targeting Microsoft Windows. We also introduce a malicious PowerShell ground truth dataset, combining snippets from various cybersecurity frameworks to train and evaluate LLMs in offensive PowerShell generation.

The evaluation focuses on gauging the quality of the generated snippets by examining their syntactic and semantic correctness, as well as performing a comparative analysis of the generation capabilities of different LLMs. Experiments show that our tool was able to generate syntactically valid PowerShell code, with over 90% of the generated samples successfully parsed without errors. Additionally, RedShell produced samples closely aligned with reference snippets, achieving high scores in common distance metrics such as edit distance and METEOR, with their similarity scores exceeding 50% and 40%, respectively.

We also conducted a functional evaluation of the snippets generated by our tool by performing the malicious code execution in a controlled environment that simulated a realistic pentesting scenario. Following the code execution, the analysis of the malicious effects reproduced within our testing environment demonstrated that RedShell can be employed by pentesters to effectively conduct a wide range of offensive cybersecurity operations targeting Microsoft Windows vulnerabilities.

The four main contributions of this thesis can be summarized as follows:

1. An extended version of a malicious PowerShell ground truth dataset from the literature. Our dataset extension incorporates additional offensive PowerShell snippets annotated with their respective descriptions in the English language, improving the size and quality of the training data in terms of the dataset coverage of the offensive techniques typically employed by security professionals in pentesting scenarios.

2. A novel pentesting tool, named RedShell, that supports the execution of a wide range of offensive cybersecurity operations with more automation. By integrating novel AI-based approaches in traditional pentesting workflows, we were able to build a powerful malicious code generator to assist red teams while performing various offensive activities targeting Microsoft Windows vulnerabilities.

3. An experimental evaluation of the syntactic and semantic correctness of the malicious PowerShell snippets generated by RedShell, employing standard metrics and methodical approaches to validate the generation capabilities of our tool. We also compared the semantic performance achieved by RedShell with the state-of-the-art solutions for malicious PowerShell generation through LLMs, including popular proprietary models such as ChatGPT [3]. Notably, our tool exhibited strong generation capabilities, outperforming the semantic scores achieved by reference alternatives, effectively establishing a new state-of-the-art solution for malicious PowerShell generation in pentesting scenarios.

4. An experimental evaluation of the functional capabilities offered by RedShell in a realistic pentesting scenario. To support the execution and evaluation of the malicious PowerShell snippets, we built a controlled environment providing the required isolation to safely conduct the experiments. We also designed an offensive pipeline simulating a realistic pentest to assess both the effectiveness and the correctness of the generated samples across a wide range of offensive activities. Notably, our specialized model demonstrated strong capabilities in generating PowerShell code that reproduced the intended malicious effects within our testing environment while following reference pentesting techniques.

Additionally, part of the work developed in this thesis led to the publication of a scientific article [7] at the Portuguese Informatics Conference **INForum 2025**, under the track *Segurança de Sistemas de Computadores e Comunicações* (SSCC). This publication was further distinguished with the **Best Student Paper Award** within the SSCC track. The presentation of our research findings at INForum 2025 was supported by UID/04516/NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) with the financial support of FCT.IP. We also acknowledge Layer8 for their valuable contribution and support during both the development and presentation of this research.

Publication:

- R. Bessa et al. "RedShell: A Generative AI-Based Approach to Ethical Hacking". In: *Proceedings of the 16th Simpósio de Informática*. 2025. URL: https://www.hlt.inesc-id.pt/~eugenio.ribeiro/inforum2025/papers/INForum_2025_paper_5.pdf

1.3 Thesis Structure

The remainder of this thesis is organized as follows:

- Chapter 2 presents an overview of related work regarding pentesting activities and Generative AI. We start by providing some fundamentals on the procedures and techniques typically employed by security professionals in pentesting scenarios. Then we introduce a background on AI-based techniques, following the evolution of Deep Learning from the simpler predictive models to the creation of the first LLM. Finally, we describe how Generative AI and LLMs can be employed to provide more automation to traditional pentesting workflows.

- Chapter 3 introduces an extended version of a malicious PowerShell dataset from the literature to support the training and evaluation of malicious PowerShell generators.

- Chapter 4 details the design of RedShell, our novel tool that assists pentesters targeting Microsoft Windows vulnerabilities by leveraging LLMs fine-tuned in malicious PowerShell generation.

- Chapter 5 presents an experimental evaluation validating RedShell's generation capabilities by assessing the syntactic, semantic, and functional correctness of the samples generated by our tool. We also performed a comparison between the capabilities offered by RedShell and the state-of-the-art alternatives for malicious PowerShell generation through LLMs, including popular proprietary solutions such as ChatGPT [3].

- Chapter 6 concludes the thesis and provides directions for future work.

2

Background and Related Work

In this Chapter, we first define the core concepts within the offensive cybersecurity field (Section 2.1), with a focus on the traditional pentesting workflows. We then present the mechanisms underlying Generative AI models (Section 2.2), tracing their origins back to the early pattern recognition techniques, which laid the foundation for modern AI. Finally, we review related work on the application of LLMs in cybersecurity (Section 2.3), particularly to automate the malicious code generation in pentesting scenarios.

2.1 Traditional Pentesting

Penetration testing, often referred to as pentesting, is a crucial activity for red teams [8]. While conducting a pentest audit, security professionals follow an offensive methodology that starts from analyzing a target network or system, and culminates in the emulation of real adversary behaviors. Through pentesting, red teams can assess the functional aspects of a system and detect potential vulnerabilities to intrusion attacks [6]. By evaluating the security countermeasures of the target system and detecting potential security lapses, pentesters are able to prevent attacks that could cause harm to that system and its users.

Weidman [9] highlights the differences between pentesting and vulnerability assessment. In a vulnerability assessment, security professionals identify core vulnerabilities in a system that could be exploited by malicious actors. Pentesting extends vulnerability assessment by further performing the exploitation of the detected flaws to assess what attackers might gain after a successful attack.

2.1.1 Phases

Pentesting can be conducted in three main phases, as described by Bianou and Batogna [10]:

- **Pre-exploitation**. Pentesters collect data to detect potential vulnerabilities in a target system, using methods such as reconnaissance, scanning, and vulnerability assessment, and tools such as Nmap [11], Nessus [12], Burp Suite [13], and Metasploit [14]. According to Weidman [9], pre-exploitation might also include a pre-engagement

phase to define the goals and the scope of the audit, and a preparation phase to develop an action plan based on the defined threat model.

- **Exploitation.** Pentesters try to obtain unauthorized access to the target system by taking advantage of the previously detected vulnerabilities.

- **Post-exploitation.** After the successful intrusion, security professionals try to maintain access to the system and to cover the tracks of the attack while gathering as much information as possible about the target infrastructure.

2.1.2 Scenarios

Pentesting scenarios can be represented as testing boxes, as illustrated in Figure 2.1. The three different testing scenarios are described by Bianou and Batogna in [10]:

- **Black-Box.** A scenario where pentesters are in a position without any detailed information about the target. The audit goal is to explore the target system and to collect information about its internal procedures.

- **Gray-Box.** A scenario that provides pentesters some limited information about the target, allowing the simulation of the behavior of an attacker that gains partial access to the target network.

- **White-Box.** A scenario that offers pentesters all the details about the potential victim, including information about IP addresses, network design, and the OS used by the target. The provided information allows security professionals to inspect in detail all the different components of the system, which might lead pentesters to detect specific security problems, including logical flaws in programs and errors in the network design.

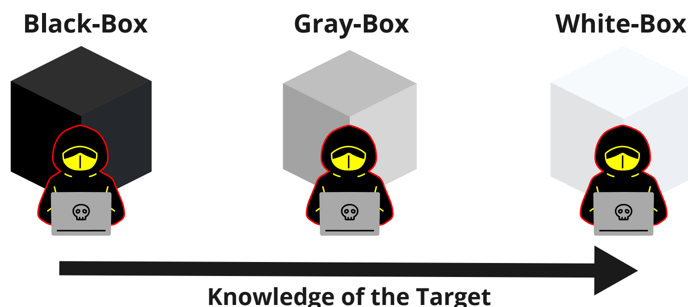

Figure 2.1: Pentesting scenarios as testing boxes.

2.1.3 Strategies

During a pentest audit, security professionals follow different strategies [6, 9, 10], based on the context and scope of the audit. In particular, pentests can be classified as:

External Pentest. An offensive operation conducted from the outside of the system boundary. Ethical hackers employ different techniques outside the security perimeter of the target, including social-engineering and client-side attacks, to gain access to its internal network. The origin of external intrusions is usually the Internet or Extranet, and their typical operational flow is described by De Jimenez in [15]. In particular, the attacker first tries to gather information from public sources and network enumeration, and then performs an attack on the organization's externally visible services, including network devices, DNS servers, email servers, firewalls, and web servers.

Internal Pentest. A testing approach that simulates an attack from the inside perspective of the target, conducted by an attacker that has some privileged access to the system. This captures the scenario of an agent that could be anyone inside the target organization, including an unsatisfied employee, an authorized user/customer, or a malicious supplier. The intruder can also be an external attacker that stole some security credential or gained access to the system through a free wireless network. Some internal audits can also assess the wireless security and the physical security controls at the target location. The advantage of internal pentests is that they allow security professionals to understand what damage an attacker can cause after accessing the internal network of the target.

Blind Pentest. A testing alternative that tries to capture the behavior of a real attacker with no prior knowledge of the target infrastructure or a very limited one. A blind strategy allows the simulation of an attack conducted from a real environment, where the malicious agent is trying to collect information about the target's weak points. Through blind testing, pentesters can find sensitive information about the target that may be available in the public domain.

Double Blind Pentest. A pentest strategy that extends the blind testing approach by having only a few people inside the target system informed about the planned intrusion. The Information Technology (IT) staff responsible for the security of the target company might not be aware of the audit, which allows pentesters to assess the quality of the security response to the intrusion, once it was detected.

Targeted Pentest. To assess more technical aspects of the system such as the design of the network, pentesters can alternatively follow a targeted approach, where the pentest is conducted in cooperation with the IT security team of the target company.

2.1.4 The MITRE ATT&CK Framework

Red teams perform a vast set of offensive activities, including credential theft, lateral movement, pivoting, obfuscation, and exfiltration, as mentioned by Liguori et al. in [16]. By leveraging these techniques, pentesters are able to emulate adversary behaviors and identify vulnerabilities within a target system. The knowledge required to effectively conduct red teaming activities is captured by public datasets such as MITRE ATT&CK [17].

MITRE ATT&CK is a globally-accessible cybersecurity framework that offers comprehensive and detailed documentation on the tactics, techniques, and procedures used by malicious actors to perform real attacks. Malicious activities are classified into 14 tactical groups that represent different stages of an attack. Built on real-world observations, MITRE ATT&CK provides a valuable knowledge base for security professionals to develop threat models in both the private sector and the cybersecurity product and service community [18]. The framework covers the attacks that can be performed against different operating systems such as Windows, Linux, and macOS, including detailed information about the most common tools and methodologies to carry out those attacks. Table 2.1 presents some examples of techniques and sub-techniques described by the framework for each tactical group. Selected examples are only a small fraction of the extensive collection provided by MITRE ATT&CK, which covers over 235 offensive techniques [10]. Following the framework documentation, a typical pentest audit may include the following steps:

1. **Reconnaissance & Initial Access.** Pentesters actively or passively gather public information about the infrastructure and the staff/personnel of the target organization, and then initiate the attack by trying to gain access to the internal network of the company. Unauthorized access can be obtained through the exploitation of public-facing web servers or by deploying spearphishing campaigns, which specifically target individuals or groups within the organization.

2. **Execution & Persistence.** Pentesters take advantage of remote access tools to run malicious code while trying to establish persistence in the target system.

3. **Privilege Escalation & Defense Evasion.** Pentesters typically employ privilege escalation techniques to acquire higher-level permissions, such as root level. Throughout the attack, pentesters may also try to obfuscate their actions and steal legitimate credentials to avoid being detected.

4. **Discovery & Lateral Movement.** To proceed with the intrusion, security professionals employ discovery techniques to identify additional opportunities for exploitation. In particular, pentesters may map other systems within the target network and attempt lateral movement.

5. **Collection & Exfiltration.** Ethical hackers can then try to collect, manipulate, and exfiltrate sensitive data or even escalate the attack to the point where they try to gain control over the compromised systems.

Table 2.1: MITRE ATT&CK framework overview.

Tactical Group	**Offensive Techniques**	**Offensive Sub-Techniques**
Reconnaissance	Gather Victim Information, Active Scanning	Vulnerability Scanning
Resource Development	Acquire/Compromise Infrastructure	DNS, Domains
Initial Access	Content Injection, Phishing	Spearphishing Link/Attachment
Execution	Command and Scripting Interpreter	PowerShell, Python
Persistence	Account Creation/Manipulation	SSH Authorized Keys, Cloud Account
Privilege Escalation	Access Token Manipulation	Token Impersonation/Theft
Defense Evasion	Impair Defenses	Disable/Modify System Firewall
Credential Access	Adversary-in-the-Middle, Brute Force	DHCP Spoofing, Password Cracking
Discovery	Account/Permission Groups	Email Account, Domain Groups
Lateral Movement	Remote Services	SSH, Cloud Services
Collection	Data from Information Repositories	Confluence, SharePoint
Command and Control	Application Layer Protocol	DNS, Web/Mail Protocols
Exfiltration	Over Web Service, Automated Exfiltration	Traffic Duplication, Text Storage
Impact	Data Manipulation	Stored Data

Cybersecurity tools can also leverage the MITRE ATT&CK framework to build solutions upon its comprehensive knowledge base. For instance, in the context of software engineering, making appropriate decisions regarding software requirements and design specifications is fundamental for producing secure and reliable software artifacts. To support the efficient and accurate design of such artifacts, Lasky et al. [19] proposed an AI-based approach that maps software requirements to the MITRE ATT&CK database.

Similarly, Ajmal et al. [20] employed MITRE ATT&CK documentation as a knowledge base to develop a sophisticated attacker emulation framework for pentesting, introducing three algorithms capable of bypassing common industry security mechanisms.

Although these approaches address distinct challenges, they share an automation-oriented methodology built upon the MITRE ATT&CK framework. This highlights the potential benefits of using this dataset as a reliable information source for training AI models, particularly LLMs, to accurately perform complex tasks, providing more automation to traditional workflows.

2.1.5 Exploitation Techniques

Exploits play a crucial role in the pentesting process. Yang et al. [21] define exploit as software that takes advantage of bugs or vulnerabilities to cause unintended or unanticipated behavior to occur in computer software or hardware. Pentesters perform the exploitation of vulnerable systems and programs using techniques such as:

Shellcode. Pentesters may release and execute in the target system malicious payloads such as shellcodes. For instance, Assembly shellcodes can be used to target programs with security lapses in memory allocation, input validation, or error handling. By manipulating stack and heap memory of programs, ethical hackers are able to perform the low-level exploitation of flaws that may exist at the hardware, kernel, or software level. Binary exploitation techniques, including buffer overflows and the exploration of use-after-free vulnerabilities, are typically performed with low-level programming languages such as C and Assembly.

Web injections. Web applications typically rely on SQL-based databases as a backend to store data. By manipulating the queries that users send to the database, pentesters are able to read or modify stored data, which can be achieved through SQL injections [22]. Alternatively, ethical hackers often perform Cross Site Scripting (XSS) attacks by injecting malicious scripts to be executed in the user's browser.

Fuzzing. A technique for detecting software vulnerabilities by generating random test cases that repeatedly target a program, monitoring for exceptions [23].

Zero-day exploits. Pentesters examine newly released programs to identify vulnerabilities unpatched by the software publishers.

Social-engineering. Even when a target system is secured with the latest patches, users often remain the weakest link [9]. Sensitive data can be compromised through social-engineering campaigns, which commonly take the form of phishing attacks. In such attacks, adversaries impersonate trusted identities to deceive users into revealing confidential information. Phishing emails, in particular, may lure victims into visiting malicious websites or downloading infected attachments. These attacks can be conducted at scale, through mass phishing campaigns targeting multiple organizations, or in a more focused manner via spearphishing, which targets specific individuals or groups within a single company.

Obfuscation. Security programs such as intrusion detectors, antivirus, and firewalls typically rely on signatures to detect recurring attack patterns and block malicious payloads. To evade such defenses, attackers often employ obfuscation techniques that alter the appearance of malicious code while preserving its functionality. A common approach involves applying encoding transformations to the shellcode, accompanied by a lightweight decoder that restores the payload upon execution on the target system. This strategy allows adversaries to bypass signature-based detection and deliver the intended malicious behavior once the obfuscated payload is decoded [24].

PowerShell. As Microsoft Windows stands out as one of the most targeted operating systems, the PowerShell scripting language also became a powerful exploitation tool for pentesters [16]. PowerShell is included by default in modern versions of Windows, and its flexibility allows ethical hackers to perform a wide variety of offensive activities, including credential theft, lateral movement, pivoting, obfuscation, and exfiltration. By executing PowerShell commands, pentesters are able to perform malicious activities without deploying complex malware files that would be easier to detect. Phishing campaigns can also take advantage of malicious attachments containing both embedded code to launch a shell and obfuscated PowerShell commands that trigger the spawn of additional processes or change system configurations. Pentesters can execute PowerShell scripts to extract sensitive data and to download and execute malicious code from remote sources, using cmdlets and .NET features of PowerShell. Additionally, ethical hackers can execute PowerShell commands to interact with red teaming tools such as Mimikatz [25] and Cobalt Strike [26], and invoke Azure PowerShell modules to perform attacks against cloud services.

2.1.6 Red Teaming Tools

Red Teaming tools allow security professionals to perform offensive activities with more automation [9]. Table 2.2 summarizes some of the most commonly used pentesting tools.

Table 2.2: Pentesting tools overview.

Tool	Description
Metasploit [14]	Open-source framework that allows red teams to develop and execute exploits to assess system vulnerabilities.
Atomic Red Team [27]	Open-source framework that maps the offensive techniques from MITRE ATT&CK into a library of red teaming tests.
Kali-Linux [28]	Debian-based Linux distribution with built-in pentesting utilities, eliminating the need for manual tool configuration.
ParrotOS [29]	Debian-based Linux distribution focusing on security, privacy, and cybersecurity, providing pre-installed pentesting utilities.
Port Scanners	Hping [30], NMap [11], Super Scan [31], and Netcat [32] perform port scanning operations.
Web Scanners	OWASP ZAP [33] and Burp Suite [13] provide pentesting tools targeting online applications.
Traffic Analyzers	TCPdump [34] and Wireshark [35] allow pentesters to monitor, capture, and analyze network traffic.
Data Collectors	Netcraft [36], Whois Lookups [37], and Nslookup [38] assist pentesters in gathering public information of target systems.
Binary Debuggers	GDB [39], objdump [40], and strace [41] perform binary analysis activities.
Password Crackers	Brutus Sec.Tools [42], John the Ripper [43], and Hydra [44] allow pentesters to brute-force passwords.
Cryptographic Tools	Cryptographic algorithms and utilities are provided by tools such as OpenSSL [45], CyberChef [46], and Cryptii [47].
Veil [48]	Payload generator to bypass common antivirus solutions.
Ettercap [49]	Tool that performs man-in-the-middle attacks, including ARP and DNS spoofing, and SSL attacks.
Arjun [50]	Tool that identifies hidden URL parameters.
Nessus [12]	Tool to detect security flaws in systems and applications through scanning and vulnerability assessment.

2.2 Large Language Models

Deep Learning was proposed in the 1980s as a way for computer algorithms to perceive complex concepts from simple representations [51]. However, the computer resources required to train Deep Learning models were not available at that time, compromising the full understanding of what these new models could achieve.

Today, transformer architectures have facilitated building higher-capacity models, and pre-training has made it possible to effectively employ these models for a wide variety of tasks [2]. The advances in computer infrastructure allowed data scientists to build larger neural networks trained on vast amounts of generic corpora. The produced models, often referred to as LLMs, have shown their ability to perform generic tasks with high accuracy [4]. In addition, fine-tuning techniques have been employed to adapt LLMs to address more specific tasks.

The generative capabilities of LLMs allowed the creation of original outputs from input descriptions, sparking a revolution in the Generative AI field. To fully understand the impact of these models in Generative AI, it is crucial to trace their origins back to the early pattern recognition techniques, which laid the foundation for modern AI.

2.2.1 Pattern Recognition

Pattern recognition consists of the automatic discovery of regularities in data through the use of computer algorithms. Detected similarities can be used to take actions and perform tasks with more automation, including classifying data into different categories. Throughout the years, many strategies were proposed to deal with the problem of searching for patterns in data. However, the best results were achieved by adopting a **Machine Learning (ML)** approach, according to Bishop and Nasrabadi [52].

In ML, a large set of n data points $\{x_1, \ldots, x_n\}$, known as the training set, is used to tune the parameters of an adaptive model. The categories of those data points are known in advance, typically by individual inspection and manual labeling. The training set can be used to build a ML algorithm, which can be expressed as a function $y(x)$ that takes a new data input x and generates an output vector y. The precise form of the function $y(x)$ is determined during the training phase, also known as the learning phase, on the basis of the training data.

An **epoch** is a complete period of training the model with the training set. Multiple epochs allow the model to revisit the same data and get more accurate learning. A set of n samples extracted from the training set compose a batch and its size corresponds to a hyperparameter that determines the number of samples to work through before updating the internal model parameters. Before completing one epoch of training, the model receives weight updates for each batch that is part of the train set.

The trained model can be used to predict the label of new x inputs, which are part of the test set. The capability of the model to produce correct predictions is known as

generalization. This is a crucial property to achieve pattern recognition because the training set typically has a very limited variability in terms of known input vectors. Through generalization, trained models will be able to make correct classifications of input vectors that differ from the ones previously learned during the training phase.

Before executing the training phase, pattern recognition algorithms usually take advantage of a pre-processing phase, where the input variables are pre-processed to transform them into some new space of variables. The goal of this transformation is to reduce the variability of the input vectors, to find a space of variables where the pattern recognition problem will be easier to solve. A less variable space of inputs makes it easier for ML algorithms to distinguish between the different classes of inputs.

The transformation of variables to improve pattern recognition, also referred to as feature extraction, can be employed to speed up computations that process very complex data. The selection of the most useful features for pattern recognition allows the removal of features that are less relevant for the problem, which leads to a dimensionality reduction. However, the pre-processing should be handled carefully because if important information is discarded, the overall accuracy of the pattern recognition solution can be compromised.

2.2.2 Machine Learning

ML algorithms can follow different approaches to achieve the learning goal. Applications in which the training data comprises examples of the input vectors along with their corresponding target vectors are known as **supervised learning** problems [52]. It is the case of the data classification problem, where the goal is to assign each input vector to one of a finite number of discrete categories. If the output consists of one or more continuous variables, then the task is called a regression. For instance, a regression problem occurs while observing a real-valued input variable x to predict the value of a real-valued target variable t. In particular, the Polynomial Curve Fitting problem is a specific type of a regression problem, which can be solved by building a linear model based on a polynomial function of the form

$$y(x, w) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

The values of the coefficients w will be determined by fitting the polynomial to the training data. This can be done by minimizing an error function that measures the misfit between the function $y(x, w)$, for any given value of w, and the data points of the training set. One simple choice of error function, which is widely used, is given by the sum of the squares of the errors between the predictions $y(x_n, w)$ for each data point x_n and the corresponding target t_n. The error function can be expressed as

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2$$

While building the model, if the selected polynomial fits perfectly each point of the training data, then it might be possible to observe that the fitted curve oscillates wildly and gives a very poor representation of the intended function. This will affect the accuracy of the model predictions for new inputs, reflecting a behavior known as **over-fitting**.

Unsupervised learning is a different learning approach that occurs when the training data consists of a set of input vectors x without any corresponding target values. The goal in such a problem may be to perform clustering to discover similar groups within the input data, density estimation to determine the distribution of data in the input space, or data projection from a high-dimensional space down to two or three dimensions for the purpose of visualization.

Additionally, a **reinforcement learning** problem occurs when the learning goal is to find the right actions to take to maximize a reward. In this problem, also known as the credit assignment problem, the ML algorithm is not aware of the optimal outputs like in supervised learning, and those might be discovered by performing an exploration process of trial and error.

2.2.3 Neural Networks

Models composed of linear combinations of fixed basis functions can be employed for **data classification** and **regression** tasks since they have useful analytical and computational properties [52]. However, their practical applicability is limited by the curse of dimensionality, which is the problem of performing pattern recognition in spaces with many dimensions. The most successful model in solving this challenge is the feed-forward neural network, also known as the multilayer perceptron, which is composed of multiple layers of logistic regression models. The base unit of a neural network is a neuron or **perceptron**, which consists of a set of inputs x, weights w, the weighted sum of inputs z, and an activation function $f(z)$ that produces the output, as represented in Figure 2.2.

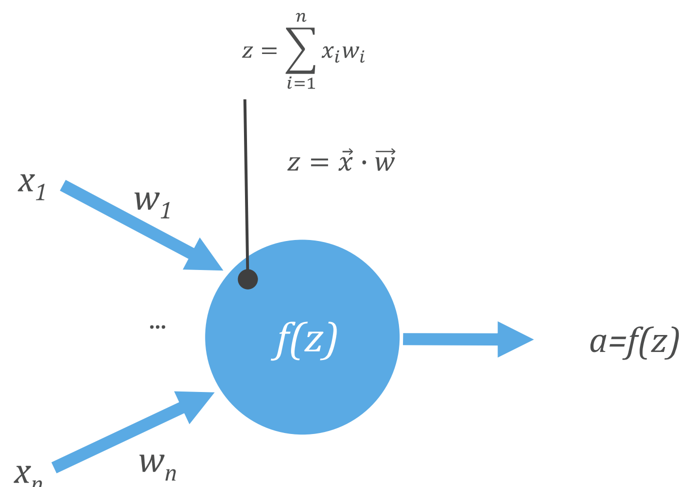

Figure 2.2: Perceptron mathematical representation.

During the training phase, if the neuron produces an output that does not match the expected one, the weights receive a small update. The update and the new weight can be computed by the equations

$$\Delta w_{ij} = \alpha \cdot (t_j - a_j) \cdot x_i$$

$$w_{ij}^{\text{new}} = w_{ij}^{\text{old}} + \Delta w_{ij}$$

While a single neuron can only solve simple linear problems, a network of perceptrons combining simple activation functions is able to solve more complex problems composed of linear subproblems. The multilayer perceptron can be defined as a network of connected neurons that will act as a single perceptron, hiding the complex operations computed in the middle layers, also known as hidden layers. Only the input and output layers are observed during that process, which creates the illusion of a single perceptron solving the problem.

Neural networks need to compute a gradient, which is a derivative of the error function. The gradient will slowly descend into a local minimum to minimize that error function. The error function value is computed at the output layer, where the produced output is available to be compared with the expected one. If there is a mismatch, there is the need to adjust all the weights, which can be achieved by executing the backpropagation algorithm, where the network is iterated backwards while computing the error in each neuron and adjusting the weights.

To solve even more complex problems, such as non-linear problems, the network needs neurons with more complex activation functions, including Sigmoid, tanh, ReLu, and Softmax. Large networks with billions of neurons combining complex activation functions are the basis to achieve Deep Learning.

2.2.4 Deep Learning

One of the key challenges in ML is to represent the informal knowledge that models need to be able to solve complex problems such as recognizing spoken words or faces in images. Informal knowledge is intuitive for humans, but it needs to be formally represented to allow computers to learn from it. However, human operators usually struggle to provide formal representations for informal concepts.

According to Goodfellow, Bengio, and Courville [51], the solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. By allowing models to acquire knowledge from experience, human operators no longer need to formally specify all the required informal knowledge. Through representation learning, models can find the optimal features to extract from raw data by performing their own pattern recognition. Representations produced by models often result in much better performance when compared to the formalizations provided by human hands, allowing AI systems to rapidly adapt to new tasks with minimal human intervention [51].

Deep Learning takes advantage of representation learning by leveraging neural networks that allow computers to build complex concepts out of simpler concepts. Through Deep Learning, it is possible to produce optimal representations of complex data, improving the capability of models to learn how to perform complex tasks. Deep Learning also allows models to learn multistep computer programs, where each layer of the neural network represents a state achieved after executing a set of instructions in parallel. Networks with greater depth can also execute more instructions in sequence, which offers the power to create complex systems that are able to refer back to the results of earlier instructions during the general computation. Following the advances on Deep Learning, neural networks were employed to automate a wide range of complex tasks, sparking a revolution in fields such as Natural Language Processing (NLP) and Neural Machine Translation (NMT).

2.2.5 Natural Language Processing

NLP aims to convert human language into a formal representation that is easy for computers to manipulate [53]. Taking advantage of NLP techniques, Deep Learning models are able to communicate with humans by processing prompts in natural language and producing original responses in some target language, which is the basis to achieve **Generative AI**. NLP is composed of different sub-tasks, including finding semantically related words, performing semantic role labeling and conducting NMT.

NMT consists of the machine translation of a sequence of tokens x in a given source language, into a sequence of tokens y in a target language [54]. The translation is performed for every individual token by choosing the prediction with the highest probability from a set of translation probabilities $P(y|x)$. Typically the models that perform these predictions are auto-regressive since they compute y based on x and also consider the previous translated tokens. The final translation probability will be the product of each token's translation probability.

The first NMT models were built using an **encoder-decoder** architecture composed of two different types of neural networks, the encoder and the decoder. The encoder network transforms x into a fixed-length vector or matrix representation, and the decoder network takes that as input and performs the auto-regressive translation. Typically, Convolutional Neural Networks (CNNs) were employed to perform the input processing in parallel, while the auto-regressive translation was commonly performed by Recurrent Neural Networks (RNNs).

The decoder was trained to predict the next word $y_{t'}$ given the context vector c and all the previously predicted words $\{y_1, \cdots, y_{t'-1}\}$. Assuming $y = (y_1, \cdots, y_{T_y})$, the probability over the translation y is defined by decomposing the joint probability into the ordered conditionals

$$p(y) = \prod_{t=1}^{T} p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c)$$

However, the decoder performance was limited by the recursive behavior of RNNs, creating a gap for the proposal of a new architecture known as transformer.

2.2.6 The Transformer Architecture

The **transformer** architecture was introduced by Vaswani et al. in [2] as a way to mitigate the performance limitations of traditional CNNs and RNNs. The proposed model architecture replaces recurrence and convolutions by connecting the encoder and decoder networks through an attention mechanism, as illustrated in Figure 2.3. Experiments on NMT tasks showed transformer-based models to be superior in quality while being more parallelizable and requiring significantly less time to train when compared to traditional models composed of CNNs and RNNs.

The attention mechanism employed by this architecture consists of a function that maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

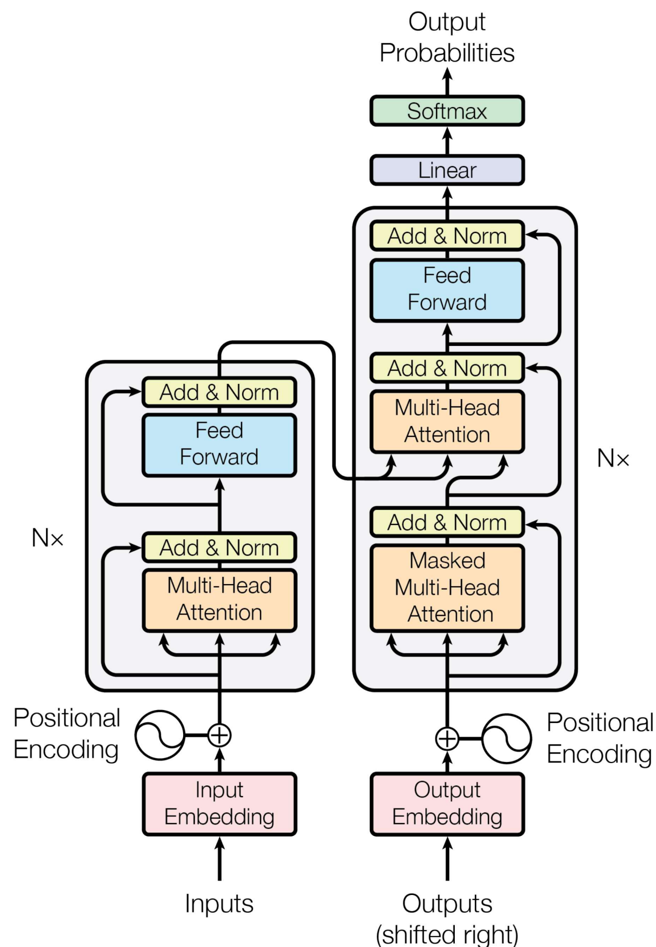

Figure 2.3: The transformer model architecture (from [2]).

Transformers take advantage of stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, as illustrated in the left and right halves of Figure 2.3, respectively. Self-attention, sometimes called intra-attention, is a particular type of an attention mechanism that relates different positions of a single sequence in order to compute a representation of that sequence. Instead of performing a single attention function, transformers take advantage of multi-head attention to linearly project the queries, keys, and values h times and employ the projected versions to execute the attention function in parallel. This allows the model to jointly attend to information from different representation subspaces at different positions. Transformers became the dominant architecture to build LLMs.

2.2.7 First Large Language Model

In October 2018, Google researchers introduced Bidirectional Encoder Representations from Transformers (BERT) [55], which is an early example of a LLM. Due to its unique architecture, BERT achieved a significant improvement while performing NLP tasks such as question answering and language inference. This model can be used to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. This allows the pre-trained BERT model to be fine-tuned with just one additional output layer.

Before BERT, strategies for applying pre-trained language representations to downstream tasks were limited to two major alternatives, feature-based and fine-tuning. The feature-based approach uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach introduces minimal task-specific parameters, and it is trained on the downstream tasks by fine-tuning all the pre-trained parameters. Both approaches are limited by the fact they use only unidirectional language models to learn general language representations.

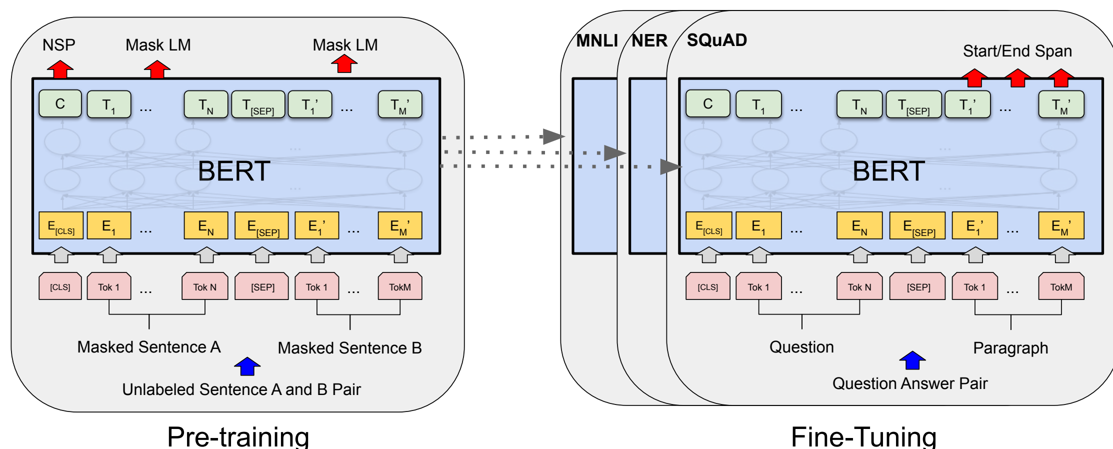

Figure 2.4: The BERT model architecture (from [55]).

BERT approach differs from those solutions by improving the fine-tuning process with context from both directions. This is achieved by employing a Masked Language Model pre-training objective, which randomly masks some of the tokens from the input, training the model to predict the original vocabulary ID of the masked word based only on its bidirectional context. Figure 2.4 shows the general architecture of BERT. Pre-training and fine-tuning follow the same process and differ only in the output layers. The same pre-trained model parameters are used to initialize models for different downstream tasks. During fine-tuning, all parameters are fine-tuned. $[CLS]$ is a special symbol added in front of every input, and $[SEP]$ is a special separator token used to separate questions and answers, for example.

2.2.8 Code Generators

Table 2.3 presents the recent evolution of LLMs in machine code generation.

Table 2.3: Evolution of LLMs in code generation.

Model	**Institution**	**Year**	**Machine Code**
CodeBERT [56]	Microsoft	2020	Python, Java, JavaScript, C, C++
CodeGPT [57]	Microsoft	2021	Python, JavaScript, TypeScript
GPT-Neo [58]	EleutherAI	2021	Python, C++, Java
Codex [59]	OpenAI	2021	Python, JavaScript, SQL, HTML
CodeParrot [60]	Hugging Face	2022	Python
PaLM-Coder [61]	Google	2022	Python, JavaScript, Java
BLOOM [62]	BigScience	2022	Python, JavaScript, C++
ChatGPT [3]	OpenAI	2022	Python, SQL, JavaScript, HTML
GPT-4 [63]	OpenAI	2023	Python, SQL, JavaScript, Rust
StarCoder [64]	Hugging Face	2023	Python, C, C++, JavaScript, Java
WizardCoder [65]	Microsoft	2023	Python, Java, SQL
Code Llama [66]	Meta	2023	Python, JavaScript, Java
Claude 3 [67]	Anthropic	2024	Python, JavaScript, SQL
Llama3 [68]	Meta	2024	Python, C++, JavaScript
StarCoder2-Instruct [69]	Hugging Face	2024	Python, JavaScript, Java, HTML
Codestral [70]	Mistral AI	2024	Python, C++, JavaScript

After BERT, various LLMs have been proposed to perform different generation tasks. For code generation in particular, Feng et al. introduced CodeBERT [56], a bimodal pre-trained model supporting the understanding of both natural and programming languages. CodeBERT is able to perform the translation of natural language prompts into machine code, being highly accurate in tasks such as performing code search and generating code documentation. CodeBERT was trained on data extracted from GitHub code repositories with 6 different programming languages. Table 2.3 offers only a broad view on the generation capabilities of LLMs since models such as Codestral [70] are already fluent in more than 80 programming languages.

2.3 Cybersecurity and Generative AI

LLMs can be employed to provide more automation to both defensive and offensive approaches within the cybersecurity field [71]. From a defensive point of view, transformer models can be leveraged to perform complex classification tasks such as detecting malicious activities. LLMs can also be employed for other purposes than just defensive strategies. Following an offensive approach, Generative AI can emulate adversarial behaviors by generating attacks and offensive code snippets that can be used by red teams in controlled environments such as pentesting.

2.3.1 Defensive Cybersecurity

From a **defensive cybersecurity** perspective, the goal is to achieve some level of automation in the detection and mitigation of security risks. Detection can be achieved by leveraging security detectors such as intrusion detectors and by performing the stream processing of security logs. After some threat is identified, an alert can be produced, and the detected risk can be mitigated either by human hand or by automatically executing some security protocol.

To achieve the desired automation, it is important to consider the capability of LLMs to consume and analyze large amounts of data, which can be very useful in the stream processing field. It is possible to take advantage of the ability of LLMs in classification tasks to identify phishing emails and malicious files.

LLMs can be used to improve defensive cybersecurity approaches, from identifying potential risks to providing countermeasures and response actions to the identified threats [71]. For example, Eze and Shamir [72] take advantage of AI-based tools to identify AI-generated phishing emails through automatic text analysis. Vulnerability detection can be achieved through models such as VulBERTa [73].

Blue teams can take advantage of LLMs to identify, detect, mitigate, and respond to different threats. LLMs are able to deliver accurate and predictive risk assessments by processing large amounts of data and by detecting risk elements that can be easily overlooked by human analysts. Protection can also be achieved by employing LLMs as web

content filters and URL scanners. Automated vulnerability fixing with LLMs diminishes the risk of cyber attacks and also provides complementary information on the detected flaws. This might include suggestions about the origin of the threat and the best strategies to perform its mitigation.

LLMs can also support the creation of Honey-Pots based on ChatBots to detect and evade intrusion attacks. For instance, Sladić et al. introduce shellLM [74] as a LLM-based solution to generate Linux Shell Honey-Pots. The produced Shell emulates a real Shell to lure attackers by conducting their interaction through a LLM.

2.3.2 Offensive Cybersecurity

Most studies focus on leveraging LLMs for defensive approaches, leaving a gap to explore their potential application in offensive cybersecurity [16]. Following an offensive approach, red teams can take advantage of malicious code generators to produce valid offensive code snippets for pentesting scenarios. Pentesters need to produce exploit code to explore vulnerabilities in a target system or network. However, the process of writing valid malicious code is slow and complex, demanding high expertise and effort from security professionals to explore the intended vulnerabilities. To improve that process, AI-based code generators can be employed to produce offensive code snippets from descriptions in natural language.

While LLMs usually perform well in generic programming tasks such as building web applications and solving coding challenges, their application in more particular scenarios such as malicious code generation is more challenging. Without the knowledge required to perform specific tasks, models might suffer from hallucination and produce less accurate results. Hallucination occurs when LLMs produce incorrect outputs that appear to be plausible, as explained by Fan et al. in [5]. In the context of malicious code generation, the execution of the generated code snippets might be compromised by unexpected bugs. Additionally, the output code might not match the input description, which prevents the snippet from producing the desired malicious behavior.

Pre-training and **fine-tuning** techniques can be employed to train models to perform more specific tasks while avoiding hallucination. For example, Codex [59] was fine-tuned to perform code completion with more accuracy. The positive impact of fine-tuning is also highlighted by Gunasekar et al. in [75] by showing that LLMs with lower parameter counts trained only with a textbook-quality code are able to achieve a comparable performance to larger models while performing generation tasks. Following the same approach, Liguori et al. trained LLMs to perform the accurate generation of malicious code by employing pre-training and fine-tuning techniques [16, 24].

Prompt Engineering techniques can also be employed to improve the performance of generative models in specific tasks by providing additional context to the input prompts. Li et al. introduce in [76] a novel prompting technique to improve the performance of LLMs in code generation tasks. However, while evaluating the code generation capabilities of

GPT-4 [63], Shin et al. [77] show that fine-tuning techniques produce better results when compared to Prompt Engineering.

Furthermore, the performance of generation tasks can be improved through **Retrieval-Augmented Generation (RAG)**. Lewis et al. [78] explain that this technique enriches pre-trained, parametric-memory generation models with a non-parametric memory through a general-purpose fine-tuning. Models take advantage of the provided knowledge base to find accurate information to improve the context of the input description. By enhancing generative models with retrieval-based strategies, it is possible to improve the quality and relevance of the generated outputs.

2.3.3 Ethical Concerns

LLMs offer a powerful way to enhance both defensive and offensive cybersecurity solutions by allowing them to achieve more automation while still being highly accurate. However, there are some ethical concerns about the possible criminal usage of these models. According to Alotaibi, Seher, and Mohammad [79], it is possible to prompt engineer OpenAI ChatGPT [3] to bypass its ethical limits and generate offensive code. Furthermore, ethical hacking frameworks such as CSRT [80] take advantage of the multilingual understanding of LLMs to create prompts that are able to bypass ethical restrictions. These tools can also be employed by malicious actors to perform attacks with more automation. With LLMs, there is the potential risk to lower the effort and the degree of expertise required for real attackers to generate malicious code.

Advances on Generative AI will affect the Computer Security field due to the impact of AI-based code generators [76]. A new threat arises from this novel paradigm, since attackers can now easily use LLMs to produce malicious code, which brings more diversity and agility to their attacks. Today anyone can easily become an attacker by taking advantage of AI-based code generators, available through public services such as GitHub Copilot [81] and Amazon CodeWhisperer [82]. These generators leverage the same technology behind the well-known OpenAI ChatGPT [3] to convert natural language prompts into entire methods and functions.

LLMs can be employed to automate malicious activities such as conducting phishing campaigns and generating malware and offensive payload code. For example, models such as WormGPT [83] and FraudGPT [84] can be employed to perform social-engineering attacks, while automated hacking can be achieved by XXXGPT [85] and WolfGPT [85].

The ethical safeguards of generative models such as OpenAI ChatGPT [3] and Google Gemini [86] can be manipulated by attackers to achieve malicious purposes [87]. By employing techniques such as jailbreaking, reverse psychology and injections, malicious actors are able to cross the ethical boundaries of Generative AI, transforming well-intended solutions into tools with malicious and disinformation purposes.

While performing jailbreaking, attackers take advantage of a Chain-of-Thought prompting approach where the instructions to the model are provided several times, step by step,

using synonyms and psychological tricks to go beyond the imposed ethical limits. An offensive prompt can also be masked with a curious tone by applying reverse psychology methods. Additionally, malicious actors can perform injections by creating specially crafted prompts or sequences to allow generators to produce outputs that may not align with their ethical or operational guidelines [87].

Although the ethical concerns about LLMs should be taken seriously, attackers will inevitably take any opportunity to use AI-based code generators and cybersecurity professionals should also strive to use the same offensive methods to better prevent and mitigate malicious activities [76].

2.3.4 AI-based Solutions for Pentesting

In recent years, several AI-based tools were proposed to provide more automation to complex pentest audits. Agent-based architectures combining multiple LLMs became an effective way to build frameworks covering all the phases of a pentest. From collecting information about the target to detecting and exploring vulnerabilities, these tools are able to provide complete automation to pentesting. Furthermore, different benchmarks and evaluation metrics were proposed to evaluate the performance of AI-based solutions for pentesting, including the validation of malicious code generators. Table 2.4 presents some of the latest contributions for achieving automated pentesting.

Table 2.4: Recent contributions to the research on AI-based pentesting approaches.

Year	Reference	Contribution
2023	PentestGPT [88]	Deng et al. introduce a LLM-empowered automated pentesting framework.
2023	[89]	Liguori et al. present a study on output similarity metrics to evaluate the performance of malicious code generators.
2024	PENTEST-AI [10]	Bianou and Batogna propose a framework for automated pentesting using LLM Agents and MITRE ATT&CK [17].
2024	RedCode [90]	Guo et al. present a study on the capabilities of LLMs in generating and executing malicious code, providing a controlled environment for the experiments.

PentestGPT [88] and PENTEST-AI [10] are examples of frameworks that take advantage of LLM Agents to provide full automation to pentest audits. Both frameworks are composed of a set of modules that perform specific tasks within the pentesting process while working together to conduct a complete and effective attack.

In particular, PENTEST-AI is composed of three main components: the Perception, the Brain, and the Action Agents. The Perception Agent converts multimodal information, such as text and video, into a representation that can be processed by the Brain Agent,

which performs activities such as thinking, decision-making, and operations with storage, including the knowledge provided by MITRE ATT&CK [17]. The processed data is leveraged by the Action Agent to support the execution of tools that produce some action in the environment.

During the execution of a pentest audit, PENTEST-AI starts by performing the reconnaissance of the target. The scanning results will allow the framework to choose the best tools to explore the detected vulnerabilities, following the offensive tactics described by MITRE ATT&CK. Both PentestGPT and PENTEST-AI store relevant data in a key-value store. While PENTEST-AI takes advantage of the stored data to report the pentest results, PentestGPT also stores intermediate data to avoid context loss between modules.

To evaluate the performance of the framework, PentestGPT provides a benchmark with data extracted from common pentesting tools. In particular, the benchmark is composed of tasks and Capture The Flag (CTF) challenges with different difficulty levels, emulating real-world scenarios. The benchmark allows security professionals to track the progress of pentest audits, producing a score based on the pentest phases that were accomplished with success. While popular LLMs achieved good performances in specific sub-tasks within pentesting, they failed to execute complete pentest audits. In contrast, the results show that PentestGPT outperformed other popular solutions in tackling real-world pentesting scenarios.

The semantic evaluation of malicious code generators can be achieved by computing similarity metrics between the generated code and the expected one, as described by Liguori et al. in [89]. Secure and isolated environments are also required to assess the efficacy of the generated code since its execution is potentially harmful.

Guo et al. in [90] introduce a platform to evaluate LLM Agents capability to produce and execute malicious code, providing a controlled environment in a Docker sandbox to run the experiments. Results show that Agents are more inclined to reject executing unsafe operations on the OS, but are less likely to refuse executing code with technical bugs, which represents a significant security risk. Furthermore, unsafe operations described in natural language result in lower rejection rates compared to those presented in code format. The experiments also show that Agents with stronger coding capabilities are more likely to generate sophisticated and effective harmful software.

2.3.5 State of the Art in LLM-Based Malicious Code Generation

The development of software exploits is a challenging and time-consuming task. Pentesters take advantage of complex offensive techniques to perform the exploitation of security vulnerabilities in the target system. To automate the generation of exploits for pentesting scenarios, security professionals can take advantage of malicious code generators.

LLMs have already proven capabilities in performing code generation tasks with high accuracy [91]. Additionally, techniques such as pre-training and fine-tuning can be employed to train models to effectively assist pentesters in producing exploits. While

pre-training and fine-tuning techniques might be an effective approach to provide the required knowledge for models to perform malicious code generation, their application is also challenging due to the lack of offensive data to train and evaluate generators.

Recent studies focus on gathering data to build malicious code datasets. Code snippets are usually extracted from public sources such as forums and cybersecurity frameworks, and their labels are manually assigned by security professionals. The code labels consist of natural language descriptions explaining the intent of the snippets. The produced datasets are a useful contribution to enable the training and evaluation of LLMs in malicious code generation. Table 2.5 presents some of the latest contributions for achieving malicious code generation in automated ethical hacking scenarios.

Table 2.5: Malicious code generators overview.

Year	Reference	Malicious Code	Technique	Obfuscation
2021	EVIL [24]	Python, Assembly	Fine-tuning	✓
2022	Shellcode_IA32 [92]	Assembly	Fine-tuning	✗
2022	DualSC [93]	Assembly	Shallow Transformer	✗
2023	ExploitGen [21]	Python, Assembly	Template Augmented	✓
2023	[94]	HTML, JavaScript, SQL	GANs	✓
2024	violent-python [95]	Python	Fine-tuning	✗
2024	[16]	PowerShell	Pre-training, Fine-tuning	✓

Shellcode_IA32 [92] and DualSC [93] are both solutions to perform the generation of Assembly shellcode to target Linux IA-32 systems with the 32-bit x86 Intel architecture. However, these solutions follow different approaches to reach the same goal. In particular, **Shellcode_IA32**:

- Provides a dataset with 3200 samples of instructions in the Assembly language.
- The data was collected from public sources containing a varied set of shellcode attacks.
- Supported exploits go from simple instructions such as running a system shell through the command `/bin/sh`, to more sophisticated attacks, including exfiltrating passwords from `/etc/passwd`, flushing firewall rules from IPtables, and conducting fork-based denial-of-service attacks.

- The collected data was used to train LLMs in malicious Assembly generation through fine-tuning techniques.
- Results show that the fine-tuned LLMs are able to generate real shellcodes, given their natural language description.
- The experimental evaluation also highlights the syntax and semantic correctness of the produced snippets.

As an alternative, **DualSC**:

- Introduces a transformer-based architecture supporting the execution of two different tasks, the generation of Assembly shellcodes and the summarization of the produced snippets for educational purposes.
- Addresses the lack of training data by building a simplified version of a transformer model, the Shallow transformer, and by creating a novel normalization method known as Adjust QKNorm.

EVIL [24] and ExploitGen [21] also support the generation of Assembly shellcode. Additionally, they are able to produce Python code to obfuscate the generated shellcodes, allowing them to avoid security detectors. In more detail, **EVIL**:

- Leverages a LLM-based approach for automatic exploit generation in Assembly, from natural language descriptions.
- Selected models were fine-tuned in a dataset containing both real exploits and their respective encoders/decoders, described in the English language.
- A pre-processing and a post-processing phases were executed to maximize the accuracy of the output.
- The pre-processing phase removes irrelevant words from the initial prompt to clarify its intentions.
- The post-processing phase improves the quality and the readability of the output code by performing a cleanup procedure.
- The validation of the solution was conducted by computing a set of similarity metrics between the generated snippets and the expected ones.
- In addition, a human evaluation was conducted on the syntax and semantic correctness of the generated code. The detected semantic errors were perceived as simple to correct since they were, in most cases, caused by wrong labels or variable names, and the omission of parentheses around expressions.

In contrast, **ExploitGen**:

- Offers a solution for a Template-Augmented shellcode generation approach.
- A Template Parser is used to generate optimal prompts based on the defined templates, improving the accuracy of the output.
- The conducted evaluation highlights the syntax and semantic correctness of the produced code. However, the parsing process introduces an additional overhead in the solution.

Today, the Windows OS also stands out as one of the most targeted systems, and the PowerShell language has become a crucial exploitation tool for pentesters. Liguori et al. in [16] present a solution for generating **malicious PowerShell**:

- Pre-training and fine-tuning techniques are employed to train the selected LLMs in malicious PowerShell generation.
- Pre-training was conducted with a dataset containing around 90,000 raw PowerShell samples extracted from public GitHub repositories.
- Fine-tuning was conducted with a dataset containing around 1000 manually labeled samples, covering the most part of the attacks described by MITRE ATT&CK. The snippets were manually labeled, combining descriptions from attacker emulation tools and cybersecurity frameworks.
- The experimental evaluation of the solution was useful to assess the impact of the pre-training and fine-tuning techniques on the quality of the produced PowerShell.
- Results show the significant improvements introduced by the fine-tuning process.
- Additionally, the experiments show that, in some cases, pre-training does not have a significant impact on the output correctness, especially in models with longer Fine-Tune phases. With the increase of the fine-tuning period, it was observed that the benefits of pre-training diminish or even become counterproductive.
- The code's effectiveness was evaluated in a controlled environment by observing the events produced by its execution.

With the Internet of Things, wireless sensor networks and web applications also became popular targets. Chowdhary, Jha, and Zhao in [94] propose a framework to assess the security of **web applications**:

- The framework takes advantage of GANs to generate XSS attacks and SQL injections, capable of bypassing web application firewalls.
- The solution scales well on a large-scale web application platform, saving the significant effort invested in manual pentesting.

Malicious Python code is often used by attackers due to its simplicity and the flexibility provided by the available libraries. Natella et al. in [95] present **violent-python**:

- A malicious Python dataset to study the capabilities of LLMs in generating Python-based exploits.
- The dataset provides labeled code for pentesting, forensic and network traffic analysis, and social-engineering attacks.

2.4 Summary

In this Chapter, we presented the fundamentals on both traditional pentesting and generative AI-based models. Then we studied how security professionals can use LLMs to enhance the automation of the traditional offensive cybersecurity workflows, particuraly with the generation of malicious code in pentesting scenarios.

3

The Ground Truth Dataset

In this Chapter, we introduce a malicious PowerShell ground truth dataset to support the training and evaluation of RedShell. The dataset combines PowerShell code snippets extracted from various pentesting tools and frameworks, and their corresponding descriptions in the English language. To build the ground truth dataset, we first selected a high-quality malicious PowerShell corpus from the literature, which we describe in more detail in Section 3.1. Additionally, we introduce in Section 3.2 our extended version of the dataset, which incorporates additional code samples, thus improving the overall size and quality of the training data.

3.1 Reference Dataset

We selected as a reference dataset the offensive PowerShell collection provided by Liguori et al. in [16]. The dataset is composed of 1,127 samples of offensive PowerShell commands, capturing different real-world scenarios where pentesters may take advantage of PowerShell to perform cybersecurity operations targeting Microsoft Windows devices.

The snippets comprising the dataset were collected from online security-related resources, including community-driven cybersecurity wikis and blogs about ethical hacking such as HackTricks [96], Red Team Recipe [97] and Infosec Matter [98]. Additionally, snippets were also extracted from various cybersecurity frameworks such as Atomic Red Team [27], Stockpile [99] and Empire [100].

3.1.1 MITRE ATT&CK Coverage

The reference dataset covers 13 out of 14 offensive tactics described by the MITRE ATT&CK [17] framework, only leaving uncovered the *Resource Development* tactic. Figure 3.1 presents the number of samples available in the dataset for each offensive tactic, according to the classification provided by Liguori et al. in [16].

Furthermore, each PowerShell snippet was annotated with a natural language description extracted from the respective code source, while additional information was provided to the samples that did not come with a predefined label or lacked more clear descriptions.

3.1.2 Limitations

The reference ground truth dataset provides a reliable baseline to create and evaluate malicious PowerShell generators, as demonstrated by Liguori et al. in [16]. However, we identified some limitations and improvement opportunities regarding the contents of the dataset, including:

- **Size Limitation.** A small dataset (only ~1k entries) might not be suitable to effectively fine-tune the most recent and complex LLMs.
- **Offensive Modules.** Some important offensive PowerShell modules were lacking representation in the dataset, including Nishang [101], PowerUpSQL [102], and MicroBurst [103].
- **MITRE ATT&CK Coverage.** The *Resource Development* tactic was not covered by the reference PowerShell collection. Furthermore, the dataset offered a limited number of snippets for relevant pentesting tactics, including *Execution* (54 snippets), *Privilege Escalation* (37 snippets), and *Reconnaissance* (5 snippets), as illustrated in Figure 3.1.

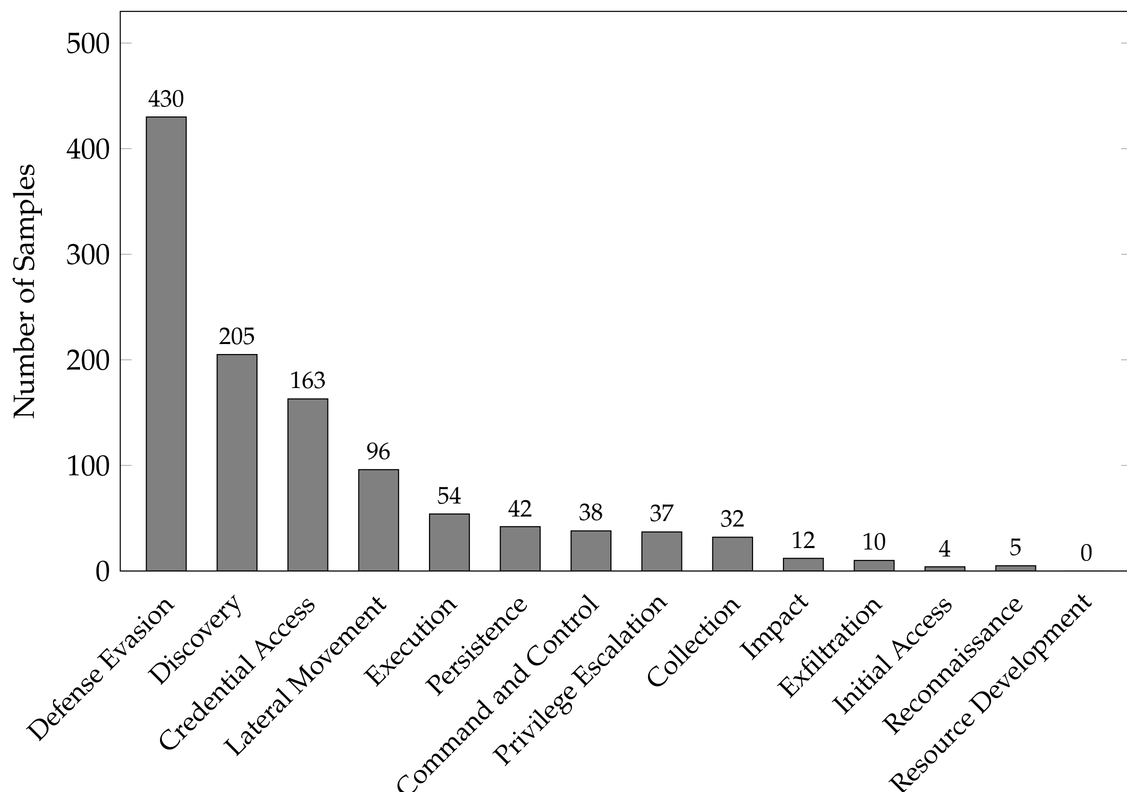

Figure 3.1: Reference ground truth dataset coverage of MITRE ATT&CK offensive tactics.

3.2 Extended Dataset

We built an extended version of the reference ground truth dataset by collecting new malicious samples, thus improving the overall quality and size of the dataset. We introduced 1,135 additional code samples of offensive PowerShell, effectively doubling the size of the original dataset. The new collected snippets and their natural language descriptions were both extracted from the sources enumerated in Table 3.1, providing a broader view on existing offensive PowerShell modules and frameworks.

3.2.1 MITRE ATT&CK Coverage

The extended ground truth dataset not only covers all the 14 offensive tactics from the MITRE ATT&CK framework, but also increments the number of available samples for the techniques typically employed by security professionals in pentesting scenarios. We manually classified the tactical group of each new code snippet based on the MITRE ATT&CK documentation. In particular, the classification process relied on the information provided by each code label and the contextual details extracted from its source, identifying keywords indicative of the adversarial techniques and the security vulnerabilities being targeted by each snippet.

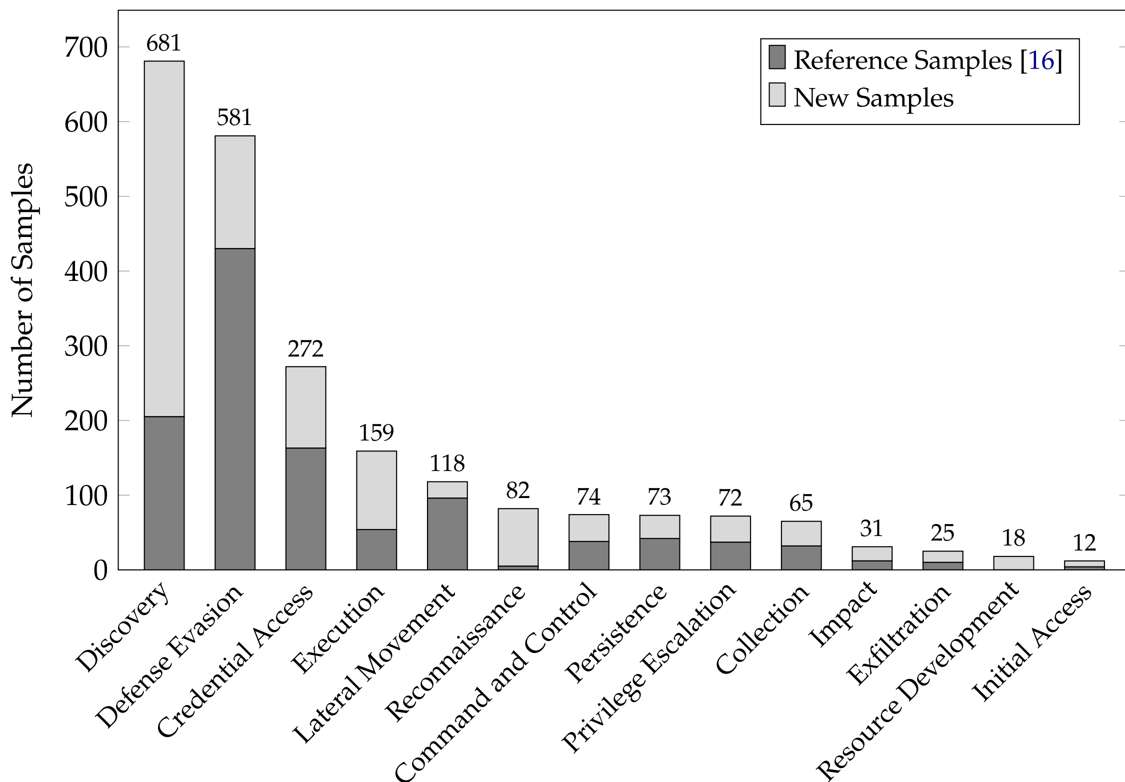

Figure 3.2: Ground truth dataset coverage of MITRE ATT&CK offensive tactics.

Table 3.1: Data sources of the new collected PowerShell samples.

Data Source	Samples	Data Source Description
GitHub [104, 105]	201	Public repositories containing malicious PowerShell scripts and commands.
Medium [106]	153	Online blog and publication platform that includes both cybersecurity-related PowerShell cheatsheets and walkthroughs on CTF tasks from tryhackme [107].
Nishang [101]	131	Framework and collection of PowerShell scripts and payloads to perform various malicious activities.
MicroBurst [103]	116	PowerShell functions and scripts that support Azure Services discovery, weak configuration auditing, and post-exploitation actions through modules such as Az [108], AzureAd [109] and MSOnline [110].
PowerView [111]	102	Enumeration tool to extract network information on Windows domains, using AD [112] hooks and the underlying Win32 API.
Posh-SecMod [113]	98	PowerShell module offering a vast set of functions and utilities for security professionals.
Mimikatz [25]	79	Tool to extract plaintext passwords from memory, perform pass-the-hash attacks, and inject code into remote processes.
PowerUpSQL [102]	77	Functions that support the discovery of SQL Server instances, weak configuration auditing, privilege escalation on scale, and post-exploitation actions.
PoweSploit [114]	48	A collection of PowerShell modules providing pentesting utilities to ethical hackers.
StationX [115]	38	cybersecurity training platform that includes PowerShell cheatsheets for system administrators.
Powermad [116]	26	Functions targeting MachineAccountQuota and DNS.
PowerUp [117]	24	Program to perform quick checks against a Windows machine for any privilege escalation opportunities.
RACE [118]	21	PowerShell module for executing Access Control List attacks against Active Directory services.
PowerCat [119]	21	PowerShell TCP/IP networking tool.

Figure 3.2 presents the number of new collected PowerShell samples for each tactical group, highlighting how both reference and extended datasets contribute to the final ground truth composition. The ground truth dataset effectively covers the offensive tactics typically employed by ethical hackers in pentesting scenarios. In particular, the five MITRE ATT&CK tactics most strongly represented in the dataset are the following:

Discovery. Represents almost half of the number of new collected snippets. The high availability of *Discovery* samples highlights the capabilities offered by PowerShell to perform various discovery activities on Microsoft Windows devices, including network service discovery and permission groups discovery.

Defense Evasion. The dataset offers a wide representation of defensive strategies employed by ethical hackers to avoid detection and conceal traces of their malicious activities. *Defense Evasion* techniques include disabling or modifying security software such as the Windows Defender and the Antimalware Scan Interface (AMSI), to allow the stealthy execution of malicious programs. Pentesters might also create firewall rules to allow incoming malicious traffic into the target network or try to obfuscate/encrypt malicious files to bypass security detectors.

Credential Access. Security professionals take advantage of *Credential Access* techniques to dump credentials and steal account names and passwords. Legitimate credentials can then be used to access sensitive information and services while making malicious activities harder to detect.

Execution. Ethical hackers typically employ *Execution* techniques to run malicious code on a local or remote system. For instance, intruders might take advantage of PowerShell utilities to download and run malicious scripts in-memory, creating a file-less malware that is harder to detect since it does not leave traces on disk.

Reconnaissance. Pentesters usually try to actively or passively gather information about the target to identify potential vulnerabilities. Administrative data such as information regarding the topology of the target network might be collected during the reconnaissance of the potential victim.

3.2.2 Improvements

Table 3.2 presents illustrative dataset entries for the most covered MITRE ATT&CK offensive tactics. When compared to the original dataset, our extended version of ground truth registered the following improvements:

- **Dataset Size.** The final dataset doubles the size of the reference PowerShell collection.
- **Offensive Modules.** The offensive PowerShell modules that were lacking representation, including Nishang [101], PowerUpSQL [102], and MicroBurst [103], are now a relevant part of the dataset, as illustrated in Table 3.1.

- **MITRE ATT&CK Coverage.** The extended dataset significantly improves the coverage of the offensive tactics described by the MITRE ATT&CK framework. In particular, the tactics that registered the largest increases in sample count were *Discovery* (+476 snippets), *Defense Evasion* (+151 snippets) and *Credential Access* (+109 snippets). Furthermore, the most pronounced relative growth in sample count occurred in *Reconnaissance* (over a 15-fold increase) and *Discovery* (over a three-fold increase), when compared to their original representation in the reference dataset. Relevant pentesting tactics such as *Command and Control*, *Persistence*, and *Privilege Escalation* also experienced improvements, with their sample counts nearly doubling.

Table 3.2: Snippets from the extended ground truth dataset.

Tactic	Code Description	Code Snippet
Discovery	*List the members of Domain Admins group.*	`Get-NetGroupMember -GroupName "Admins"; Get-ADGroupMember -Identity "Admins"`
Defense Evasion	*Obfuscate a malicious command by encoding it to Base 64.*	`$ps=[Convert]::ToBase64String ([System.Text.Encoding]:: Unicode.GetBytes("$Cmd"))`
Credential Access	*Use Mimikatz to dump credentials from the Security Account Manager.*	`Invoke-Mimikatz -Command "lsadump::sam"`

3.3 Summary

In this Chapter, we presented the first contribution of this thesis, the extension of a malicious PowerShell dataset from the literature. The reference dataset from [16] was identified as a meaningful contribution to address the lack of data to train and evaluate offensive code generators. Additionally, our extended version significantly improved the dataset in terms of sample number and coverage of the offensive tactics and tools described by the MITRE ATT&CK framework. The final ground truth dataset offers the required knowledge base to allow red teams to develop automated pentesting tools targeting Microsoft Windows.

4

RedShell Design

Without collecting the required amount of snippets, the training and evaluation of malicious code generators would be highly compromised. To address that, in the previous Chapter we introduced our extended version of a malicious PowerShell ground truth dataset from the literature, effectively covering the offensive tactics typically employed by security professionals in pentesting scenarios.

In this Chapter, we discuss the design of RedShell, a novel tool that assists pentesters targeting Microsoft Windows by leveraging specialized LLMs. We present the design overview of our tool in Section 4.1. In particular, we selected three state-of-the-art LLMs as potential RedShell candidates. The selection criteria for the models are described in Section 4.2, while their fine-tuning procedures are discussed in Section 4.3.

4.1 Overview

To develop RedShell, we followed the methodology illustrated in Figure 4.1, where the ground truth dataset (Chapter 3) was employed as a knowledge base to fine-tune three LLMs in malicious PowerShell generation. The snippets produced by the specialized models were then targeted by a syntactic, semantic, and functional evaluations (Chapter 5).

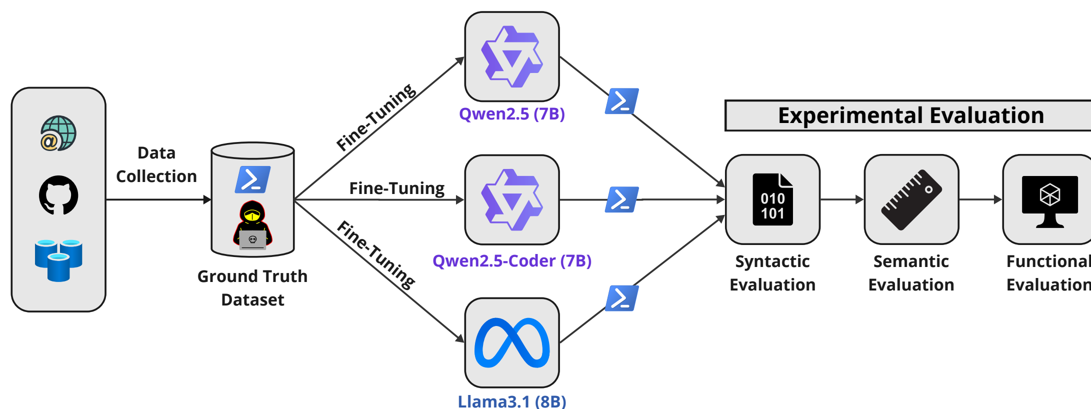

Figure 4.1: RedShell design overview.

4.2 Models

We selected three different LLMs to be fine-tuned in malicious PowerShell generation using the datasets previously described in Chapter 3. The selection process was performed based on the following criteria:

- **State-of-the-art.** We prioritized models representing the state-of-the-art families and architectures of LLMs, also considering their release date and popularity within the AI research community. In particular, we selected models that ranked highly on the HuggingFace Open LLM Leaderboard [120] and were among the most downloaded during the first half of 2025. Only models released between the second half of 2024 and the first half of 2025 were considered.

- **Availability.** Only open-weights models (i.e. LLMs with publicly available weights) were considered.

- **Coding Skills.** We selected models already pre-trained in multiple programming languages to perform general-purpose coding tasks, including code generation, summarization and reasoning.

- **Size.** We chose models with relatively small sizes, offering the possibility of being fine-tuned with less computational resources. In particular, we considered LLMs with a parameter count up to 8 billions.

Based on the previously described criteria, we identified the following LLMs as the best candidates to be fine-tuned in malicious PowerShell generation:

1. **Qwen2.5-7B** [121] belongs to the latest series of Qwen LLMs, which offer significantly more knowledge than the previously released versions while providing improved capabilities in coding and mathematics.

2. **Qwen2.5-Coder-7B-Instruct** [122] belongs to the latest series of code-specific Qwen LLMs, offering strong capabilities in code generation, reasoning and fixing.

3. **Llama3.1-8B** [123] was released on July of 2024 as part of the multilingual collection of LLMs from Meta, outperforming many of the available open-source and closed models on common industry coding benchmarks.

The selected models were downloaded from HuggingFace and fine-tuned locally, providing properties to our solution that overcome the limitations presented by closed models such as ChatGPT from OpenAI. When compared to proprietary LLMs, RedShell candidates offer the following properties:

Privacy. By manipulating our specialized models locally, we avoid to share sensitive data with private companies whose proprietary LLMs are only accessible through an external API.

Specialization. Closed models usually offer strong capabilities while performing generic tasks. In contrast, our specialized models were specifically trained to assist pentesters in malicious PowerShell generation. The same training process could not be applied to proprietary LLMs since their weights are not publicly accessible.

Weak Ethical Boundaries. Closed models are usually protected by ethical boundaries that restrict their generation capabilities. Although these protections can be often bypassed through prompt-engineering techniques, that manipulation requires time and effort from the pentesters. In contrast, the behavior of our specialized models was specifically adapted to support the automation of pentesting activities. While unrestricted LLMs may raise ethical concerns, attackers will inevitably exploit them, making it essential for ethical hackers to do the same to prevent real threats. Additionally, to ensure the responsible use of RedShell, we excluded from its training data all the *Impact* samples that could potentially compromise the integrity of target systems, thus aligning our tool with the non-destructive nature of pentesting.

4.3 Fine-Tuning

The training of the LLMs was conducted through Unsloth [124], a fine-tuning framework that manually patches complex mathematical steps and optimizes GPU kernels and VRAM (Video Random Access Memory) allocation to make training faster without any hardware changes. Unsloth also supports partial fine-tuning processes through the Low Rank Adaption Method (LoRA) [125], which allows the training to adjust only a small number of weights from the selected models, reducing the computational costs of the fine-tune.

4.3.1 LoRA Settings

LoRA is a Parameter-Efficient Fine-Tuning (PEFT) technique that operates on 16-bit transformers by freezing most pre-trained parameters and introducing a small number of trainable weights. During the partial fine-tuning process, only these additional parameters are updated. Once training is complete, the fine-tuned weights can be merged with the original model to produce the specialized version of the LLM.

We adopted standard values for both LoRA and training parameters, following the recommendations provided by the Unsloth documentation. Additionally, to determine the most effective training configuration, we conducted a set of experiments evaluating the performance of models fine-tuned under different settings. We describe the experiments in more detail in Chapter 5.

Figure 4.2 presents the LoRA configuration used to fine-tune the selected LLMs. The model parameter (line 2) initially refers to the original LLM, which was loaded into VRAM via the Unsloth framework. The hyper-parameters r (line 3) and lora_alpha (line 6) define the influence of the adapted weights in the fine-tuned LLM. In particular, the r parameter defines the number of low-rank factors introduced during adaptation, while lora_alpha acts as a scaling factor for the weight updates. In our configuration, we employed expressive values for both to allow a stronger model adaptation. Furthermore, we were able to reduce the computational costs of the training by using LoRA to restrict the fine-tuning to a subset of model layers, as defined by the target_modules parameter (lines 4 and 5).

```
model = FastLanguageModel.get_peft_model(
    model,
    r = 64,
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj",
                      "up_proj", "down_proj"],
    lora_alpha = 64,
    lora_dropout = 0,
    bias = "none",
    use_gradient_checkpointing = "unsloth",
    random_state = 3407,
    use_rslora = False,
    loftq_config = None,
)

```

Figure 4.2: LoRA settings for the fine-tuning processes.

4.3.2 Training Settings

Figure 4.3 presents the training configuration of the fine-tuning processes. The training parameters were also defined based on experimentation (Chapter 5), aiming to create specialized LLMs with strong malicious PowerShell generation capabilities. While model-specific parameters such as the tokenizer (line 3) and the num_train_epochs (line 12) were optimally adapted for each LLM, the remaining parameters (including the LoRA settings) were defined statically for all the selected LLMs, to ensure a fair evaluation between different models. In particular, we set the batch size to 8, representing the number of samples processed simultaneously during each iteration over the training dataset. This value was obtained as the product of the parameters per_device_train_batch_size (line 9) and gradient_accumulation_steps (line 10).

The learning_rate parameter (line 13), which controls the magnitude of updates to the model's weights during training, was set to 2×10^{-4}. To prevent overfitting, we applied a weight_decay of 0.01 (line 17), penalizing excessively large weight updates. For the lr_scheduler_type parameter (line 18), we employed a cosine learning rate scheduler, which modulates the learning_rate variation following a cosine decay pattern throughout the training process.

Finally, we provided the model with a context describing its expected behavior after the fine-tuning process. The context was defined as: "*Act as a malicious PowerShell generator. Generate commands in a single line, separated by semicolons and provide no further explanations.*".

```
trainer = SFTTrainer
    model = model,
    tokenizer = tokenizer,
    train_dataset = train_dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    packing = False,
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_ratio = 0.05,
        num_train_epochs = 20,
        learning_rate = 2e-4,
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported(),
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "cosine",
        seed = 3407,
        output_dir = "outputs",
        report_to = "none",
        logging_strategy="epoch",
        save_strategy="epoch",
        save_total_limit=2,
    ),
)

```

Figure 4.3: Training settings for the fine-tuning processes.

4.3.3 Dataset Partitions

The ground truth dataset was randomly split in two partitions, the training and the test datasets. Since the fine-tuning was conducted with a small and manually curated dataset, we adopted a 90/10 train-test split to maximize the amount of data available for training, optimizing the intended knowledge transfer. The test dataset, while comprising a significantly smaller part of ground truth, still provides a sufficient amount of unseen data to assess the generalization capabilities of the fine-tuned models and detect potential overfitting scenarios.

Since the reference dataset provided only aggregate tactic counts rather than a tactical classification per snippet, the train-test split does not guarantee balanced representation across MITRE ATT&CK tactics. However, this does not represent a threat to the validity of RedShell since the most common pentesting tactics such as *Discovery*, *Defense Evasion* and *Credential Access* dominate the dataset, meaning that random partitioning still ensures a high likelihood of their presence on both partitions.

4.3.4 Computational Performance

The fine-tuning processes were executed on a local Linux machine, employing a single NVIDIA GeForce RTX 4090 GPU and 23.643 GB of VRAM. By taking advantage of the optimizations provided by Unsloth and LoRA, our strategy minimizes the time, energy and computational resources required to fine-tune the selected LLMs.

Table 4.1: Training time of models fine-tuned with the reference dataset.

Model	Training Epochs	Total Training Time (min)
Llama-3.1 (8B)	18	28
Qwen2.5-Coder (7B)	20	30
Qwen2.5 (7B)	28	47

Table 4.1 presents the training times of our specialized models, both in number of epochs and minutes. The number of epochs (i.e. complete model iterations through the training data) was defined based on experimentation (Chapter 5), reflecting the different learning abilities of each LLM.

Figure 4.4 illustrates the peak reserved VRAM for each model fine-tuned with the reference dataset. We can perceive that the observed VRAM peaks are significantly far below our max VRAM limit of 23.643 GB, represented by the red dashed line.

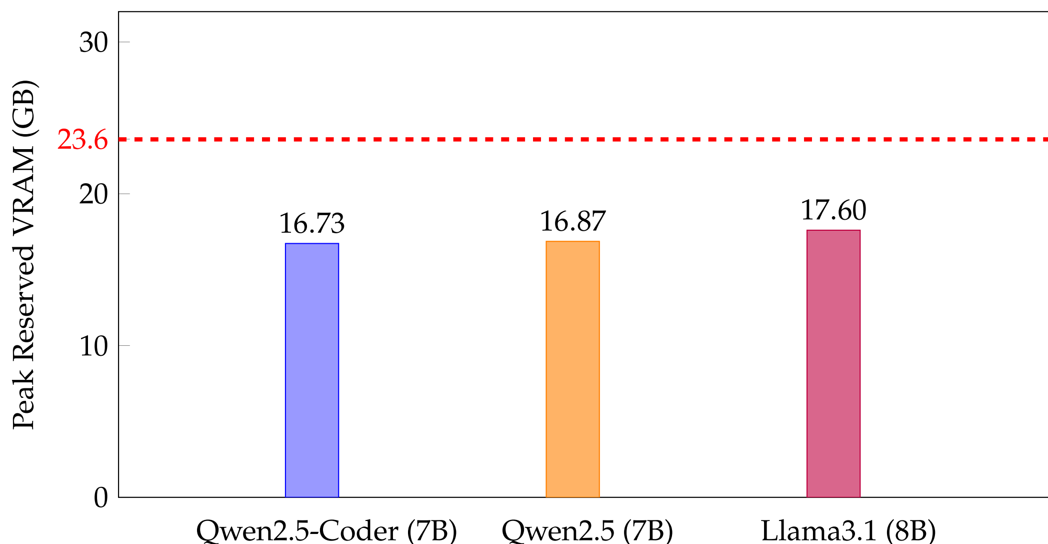

Figure 4.4: Peak reserved VRAM while fine-tuning models with the reference dataset.

4.4 Summary

In this Chapter, we presented the second major contribution of this thesis, the fine-tuning of three state-of-the-art LLMs in malicious PowerShell generation. We introduced these specialized models as potential RedShell candidates, designed to assist pentesters in automating the generation of malicious commands targeting Microsoft Windows vulnerabilities. We discussed the selection criteria for the models and the specific properties that make them suitable for our proposed solution. Additionally, we detailed our fine-tuning methodology using the Unsloth framework, and outlined the strategies employed to reduce the computational costs of the training process, particularly through the application of LoRA techniques.

5 Evaluation

In this Chapter, we present the experimental evaluation that was conducted to assess the effectiveness of employing the fine-tuned LLMs as malicious PowerShell generators in pentesting scenarios. To perform the experiments, we inferred our specialized models using the previously unseen code descriptions from the test dataset. The evaluation aimed to validate the quality of the generated code blocks by examining their syntactic and semantic correctness, as detailed in Sections 5.1 and 5.2, respectively. Additionally, Section 5.3 presents a functional evaluation, in which the generated snippets were executed within a controlled environment designed to simulate a real-world pentesting scenario. This setting allowed us to observe and document the malicious effects exhibited in the simulation environment during code execution.

5.1 Syntactic Assessment

The syntactic correctness of the PowerShell snippets was evaluated based on the number and severity of the syntactic flaws identified by PSScriptAnalyzer [126], a static PowerShell code checker provided by Microsoft.

5.1.1 Evaluation Metrics

PSScriptAnalyzer detected syntactic violations of different severity levels in the outputs of our specialized models, particularly:

- **Parse Errors.** High-severity errors that occur during the parsing of the PowerShell code, preventing the execution of the generated samples.
- **Warnings.** Flaws that may alert for the presence of bad coding practices or unexpected PowerShell patterns.
- **Errors.** High-severity flaws that alert for the violation of semantic and security rules from PowerShell.

The presence of parse errors in the generated samples was a crucial metric to identify the snippets that could not be executed. In contrary, PowerShell warnings and errors typically do not prevent the code from executing. However, these occurrences allowed us to evaluate the quality of the generated samples in terms of the adherence to the best PowerShell practices.

The results from the syntactic evaluation were then used to compute the parse error, warning, and error percentages in the samples generated by our specialized models. Since a single sample could potentially contain multiple parse errors, warnings and errors, simultaneously, our approach was to classify a sample as having a parse error if the code for that sample registered one or more parse errors, regardless of the additional presence of warnings or errors. Samples that did not register parse errors were then classified as containing warnings or errors if their code included at least one warning or one error, respectively. Samples (without parse errors) containing both warnings and errors were classified under both categories.

5.1.2 Experimental Results

The results of the syntactic evaluation of the models fine-tuned with the reference ground truth dataset can be observed in Figure 5.1. According to PSScriptAnalyzer, our specialized models were able to generate valid malicious PowerShell code. In fact, the generated samples registered low parse error and error percentages, highlighting the strong capabilities of the fine-tuned LLMs in generating syntactically correct PowerShell code.

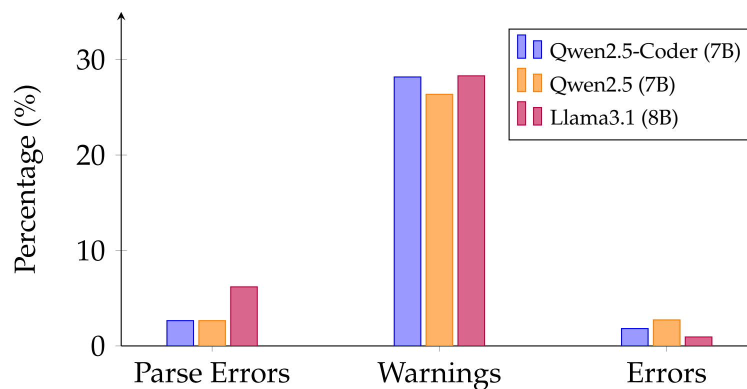

Figure 5.1: Syntactic evaluation of models fine-tuned with the reference dataset.

From the three fine-tuned LLMs, Qwen2.5 and Qwen2.5-Coder were the models that produced samples with lower parse error percentages. Additionally, all the three specialized models produced snippets with significantly high warning percentages. However, since the LLMs were specifically trained to produce malicious code, it was expected that PSScriptAnalyzer would identify warnings in the generated samples regarding safety violations of PowerShell.

Figure 5.2 presents all the syntax flaws identified by PSScriptAnalyzer in the samples generated by Qwen2.5-Coder fine-tuned with the reference ground truth dataset. The most frequently reported warnings were associated with the *PSAvoidUsingInvokeExpression* and *PSAvoidUsingCmdletAliases* rules. The *PSAvoidUsingCmdletAliases* warning is triggered by the use of command aliases, which can introduce potential bugs and reduce the maintainability of the generated code. According to PSScriptAnalyzer, command aliases should be replaced with their full names to eliminate ambiguity and improve code clarity. Furthermore, the *PSAvoidUsingInvokeExpression* warning highlights the use of a PowerShell command that enables the execution of code via string evaluation, which poses security risks by potentially allowing the execution of unsafe or dynamically constructed commands.

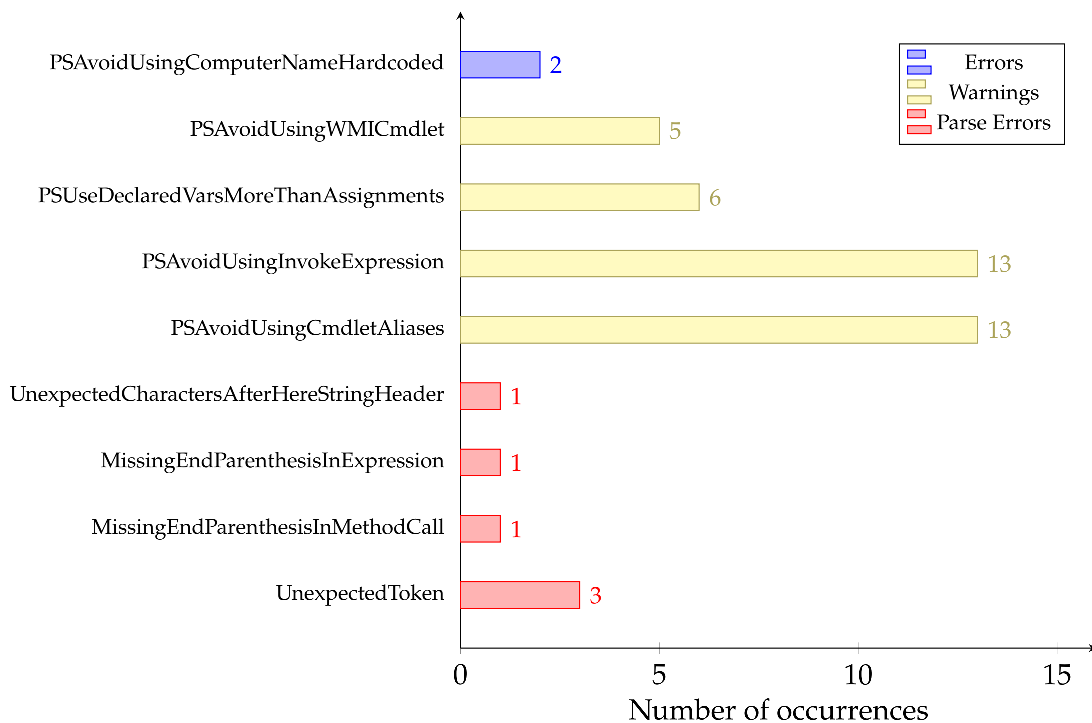

Figure 5.2: Syntax report of Qwen2.5-Coder fine-tuned with the reference dataset.

The *UnexpectedToken* parse error was the most frequent occurrence to prevent the execution of the generated samples. Furthermore, *PSAvoidUsingComputerNameHardcoded* was the most registered PowerShell error. According to PSScriptAnalyzer, hardcoding the value of the `ComputerName` argument violates a security rule from PowerShell since it will potentially expose sensitive information regarding the target host.

It is relevant to note that the *RedirectionNotSupported* parse error was excluded from the syntactic evaluation. This decision was based on the observation that many of the generated samples included the character "<", which was flagged by PSScriptAnalyzer as an unsupported redirection attempt. However, the characters "<" and ">" were frequently used by the generated samples as placeholders or delimiters for arguments without

assigned values, following the formatting patterns from the training dataset. For instance, PSScriptAnalyzer identified a redirection parse error in the snippet "`Get-PSSession -ComputerName <target>`", despite the fact that `<target>` was intended as a placeholder rather than an actual redirection operation.

Table 5.1 presents illustrative examples of syntactic flaws, categorized by severity level, as identified by PSScriptAnalyzer in the samples generated by Qwen2.5-Coder fine-tuned with the reference dataset. Text highlighted in yellow represents the code segments responsible for the identified syntactic occurrences.

Table 5.1: Illustrative syntactic flaws in the snippets generated by Qwen2.5-Coder.

<table>
<tr><th>Severity</th><th>PSScriptAnalyzer Rule</th><th>Generated Snippet</th></tr>
<tr><td>Warning</td><td>PSAvoidUsingInvokeExpression</td><td><code>Invoke-Expression -Command "IEX (New-Object Net.WebClient).DownloadString (’http://hack.site.com/code.ps1’)"</code></td></tr>
<tr><td>Error</td><td>PSAvoidUsingComputerNameHardcoded</td><td><code>Invoke-Command -ComputerName 192.168.50.200 -ScriptBlock {ps_cmd}</code></td></tr>
<tr><td>Parse Error</td><td>UnexpectedToken</td><td><code>$PSBypass=[PSCredential]::new (’username’,’password’) | ConvertTo-SecureString -AsPlainText -Force) ; $PSBypass | Invoke-Expression {Invoke-Mimikatz -DumpCreds}</code></td></tr>
</table>

5.1.3 Fine-Tuning Impact

To assess the fine-tuning impact on the syntactic performance of Qwen2.5-Coder, we compared two different versions of the model, the base version (before being fine-tuned), and the specialized version (after being fine-tuned with the reference ground truth dataset).

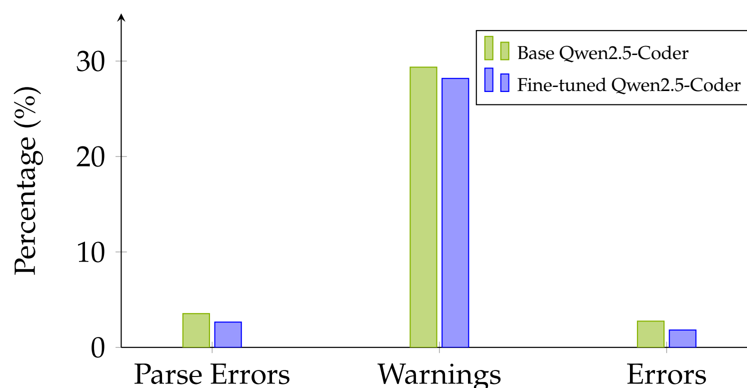

Figure 5.3: Syntactic evaluation of the base and fine-tuned versions of Qwen2.5-Coder.

The percentages presented in Figure 5.3 showcase a slight improvement across all syntactic metrics when the model was fine-tuned with the reference dataset. In fact, the observed decrease in parse errors, warnings, and errors percentages underscores the positive impact of fine-tuning on enhancing Qwen2.5-Coder's ability to generate PowerShell with greater syntactic correctness. Although the improvements were minor across all evaluation metrics, the specialized LLM successfully maintained (and slightly improved) the strong syntactic performance already exhibited by the base model.

5.2 Semantic Assessment

We defined the semantic correctness of the PowerShell code as a distance that statistically measures the similarity between the generated snippets and the corresponding expected snippets in the test dataset.

5.2.1 Evaluation Metrics

To measure the semantic correctness of the PowerShell samples, we computed the standard output similarity metrics, described by Liguori et al. in [89]:

- **ROUGE-L** Measures the similarity between the reference and generated code samples based on the longest common subsequence metric, producing a score that ranges between 0 (perfect mismatch) and 1 (perfect matching). We employed the *rouge* [127] Python package to compute ROUGE-L.
- **METEOR** Measures the alignment between reference and generated samples by mapping unigrams, producing a score that ranges between 0 (perfect mismatch) and 1 (perfect matching). We computed METEOR by leveraging the *evaluate* [128] Python package from HuggingFace.
- **BLEU** Measures the n-gram intersection between the reference and generated snippets using a score that ranges between 0 (perfect mismatch) and 1 (perfect matching), penalizing the score of the generated samples that are longer than their corresponding references. We computed BLEU for n-grams with $n = 4$, taking advantage of the BLEU implementation provided by Microsoft in CodeXGLUE [129].
- **Edit Distance (ED)** Measures the output distance by computing the minimum number of operations on single characters required to make each generated snippet equal to the reference sample. The ED score ranges between 0 (perfect matching) and a positive integer representing the number of character operations required to achieve a perfect matching. We computed ED through the *pylcs* [130] Python package, normalizing the produced mean score between 0 (perfect mismatch) and 1 (perfect matching).

- **Exact-Match** Measures the mean percentage of generated samples that perfectly match their corresponding reference samples in test dataset.

5.2.2 Experimental Results

Figure 5.4 presents the results of the semantic evaluation performed for our specialized models. The fine-tuned LLMs achieved significantly high scores across different output similarity metrics, highlighting their strong capabilities to generate malicious PowerShell closely aligned with the expected code references. In particular, Qwen2.5-Coder outperformed Qwen2.5 and Llama3.1 in all output similarity metrics. Following that observation, we defined the specialized version of Qwen2.5-Coder as the best candidate to be incorporated in RedShell.

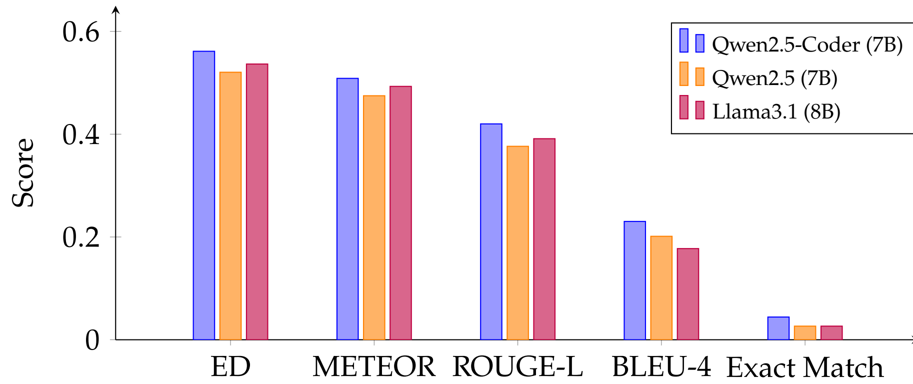

Figure 5.4: Semantic evaluation of models fine-tuned with the reference dataset.

Additionally, it is important to note that the BLEU-4, ROUGE-L, and METEOR scores were computed using the standard corpus-based approach, in which the semantic evaluation compares the aggregated collection of model predictions against the aggregated collection of code references from the test dataset. In contrast, the ED score is a sentence-level distance metric, meaning that the reported ED values correspond to the mean of all distances computed between each generated snippet and its reference.

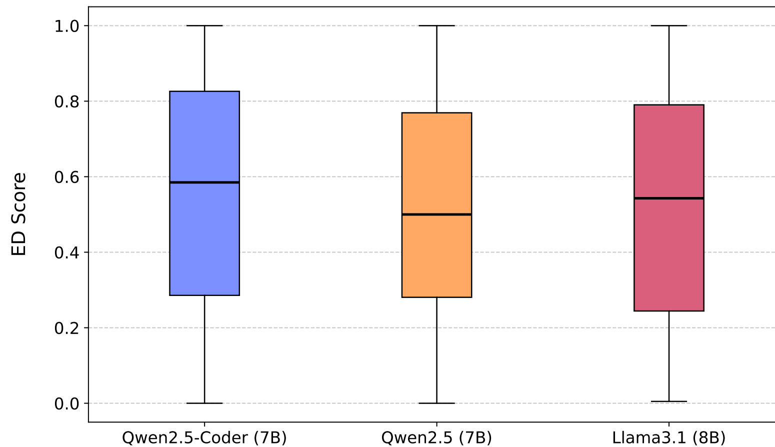

Figure 5.5: ED score distribution of models fine-tuned with the reference dataset.

Because the mean distance value can be influenced by outliers, we provide a more statistical evaluation of the ED scores for samples generated by our specialized models. Figure 5.5 shows the distribution of the computed ED scores for the snippets generated by our selected LLMs fine-tuned with the reference dataset. Reported values emphasize the stronger ED performance of Qwen2.5-Coder when compared to the remaining models. In particular, we can observe that 50% of the samples generated by Qwen2.5-Coder registered an ED score higher than 0.5, highlighting the strong semantic alignment between the produced code and its references.

To examine the distribution of ED scores in more detail, Figure 5.6 presents the frequency of distances computed for samples generated by our specialized version of Qwen2.5-Coder. The reported occurrences indicate that the most frequent ED scores fall within the intervals *[0.2, 0.3[*, *[0.8, 0.9[*, and *[0.9, 1.0]*. The interval *[0.2, 0.3[* suggests that the model struggles to reproduce the reference patterns for a significant subset of snippets, which may be due to: *i)* task-specific complexity; *ii)* insufficient representation of similar examples in the training dataset; or *iii)* the presence of more generic tasks that can be addressed through multiple approaches. Nevertheless, there is also a significant number of computed distances that fall within the *[0.8, 1.0]* range, particularly between 0.8 and 0.9, indicating that there is a high number of generated snippets that were able to achieve a strong semantic alignment with their references.

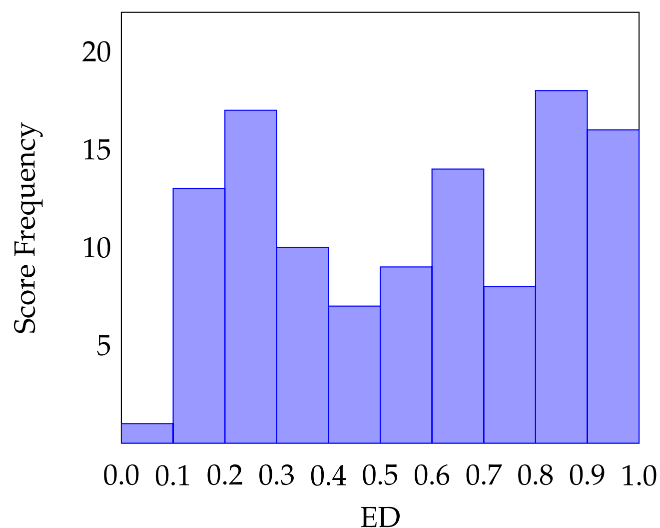

Figure 5.6: ED score distribution of Qwen2.5-Coder fine-tuned with the reference dataset.

Table 5.2 compares some illustrative examples of snippets generated by our tool and the respective ground truth references, visually emphasizing their strong similarity. Text highlighted in yellow denotes unexpected code segments while text highlighted in blue represents missing elements in the generated snippets. The first produced sample perfectly matched its reference, while the second and third snippets registered minor differences. In particular, the second snippet correctly reproduced the reference code except for the additional redirection of the command output to the file `creds.txt`. Finally, in the the third

case, the generated snippet directly invoked a PowerShell command, while the reference sample first created a PowerShell process to support the code execution. Furthermore, the two snippets diverged in their decoding operations, with the generated sample decoding the argument `Cmd` using PowerShell's native Unicode format while the reference code decoded the argument `base64EncodedCmd` considering the UTF-8 string representation.

Table 5.2: Illustrative snippets generated by RedShell with high semantic correctness.

Snippet Description	Reference Snippet	Generated Snippet	ED
List the members of Admins group.	`Get-ADGroupMember -Identity "Admins"`	`Get-ADGroupMember -Identity "Admins"`	1.0
Use Mimikatz to dump plain text passwords from memory.	`Invoke-Mimikatz -Command '"sekurlsa:: logonpasswords"'`	`Invoke-Mimikatz -Command '"sekurlsa:: logonpasswords"' | Out-File -FilePath C:\temp\creds.txt`	0.5506
Execute a malicious command encoded as a Base64 string.	`powershell.exe -c "iex ([System.Text.Encoding] ::Unicode.GetString( [System.Convert]:: FromBase64String ('Cmd'))"`	`powershell.exe -c " iex -Command ([System.Text.Encoding] :: UTF8 .GetString( [System.Convert]:: FromBase64String (' base64Encoded Cmd'))) "`	0.6422

We also report in Table 5.2 the ED scores between reference and generated snippets to statistically highlight their similarity. In fact, all reported scores exceed 0.5, indicating that the generated samples share more than 50% similarity with their code references. These results are aligned with the overall strong similarity performance exhibited by RedShell, particularly in metrics such as ED, for which the specialized version of Qwen2.5-Coder obtained a mean score of 0.5613 (Figure 5.4).

5.2.3 Comparison with State-of-the-Art Models

To further validate the generation capabilities of our specialized models, we conducted a detailed analysis of the state-of-the-art solutions for automatic malicious PowerShell generation through LLMs. We used as reference the specialized versions of CodeGPT, CodeGen and CodeT5+ proposed in [16].

The reference models were fine-tuned by Liguori et al. using the reference ground truth dataset. Additionally, CodeT5+ and CodeGen were also pre-trained with a dataset composed of 89,814 generic PowerShell samples extracted from GitHub. Figure 5.7 presents the comparative analysis of the semantic evaluation results achieved by the reference models (reported in [16]) and our specialized version of Qwen2.5-Coder, also fine-tuned with the reference ground truth dataset.

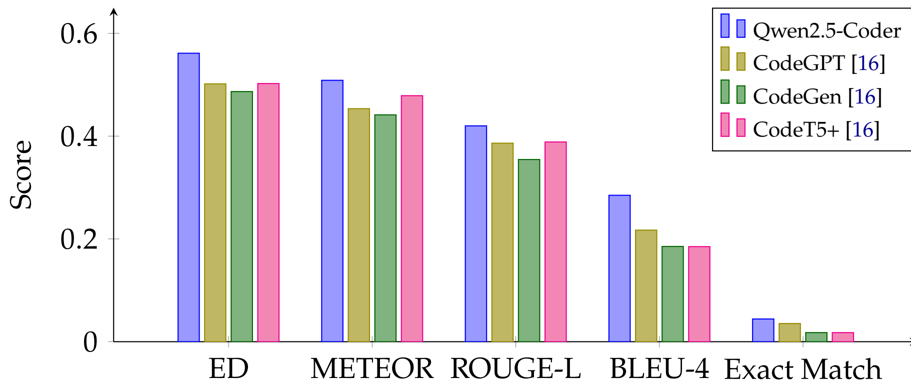

Figure 5.7: Semantic evaluation of Qwen2.5-Coder and reference models.

Qwen2.5-Coder outperformed the reference models across all output similarity metrics. Notably, our specialized LLM was only partially fine-tuned through LoRA and Unsloth, using a single GPU under low VRAM consumption conditions, while the reference models were targeted by a complete fine-tune, with CodeT5+ and CodeGen also benefiting from a specialized pre-train in generic PowerShell.

Figure 5.8 shows the distribution of ED scores on the test dataset for our specialized Qwen2.5-Coder and the reference models. The results indicate that Qwen2.5-Coder achieved higher ED similarity to the reference snippets, with 50% of its samples outperforming those generated by the reference LLMs.

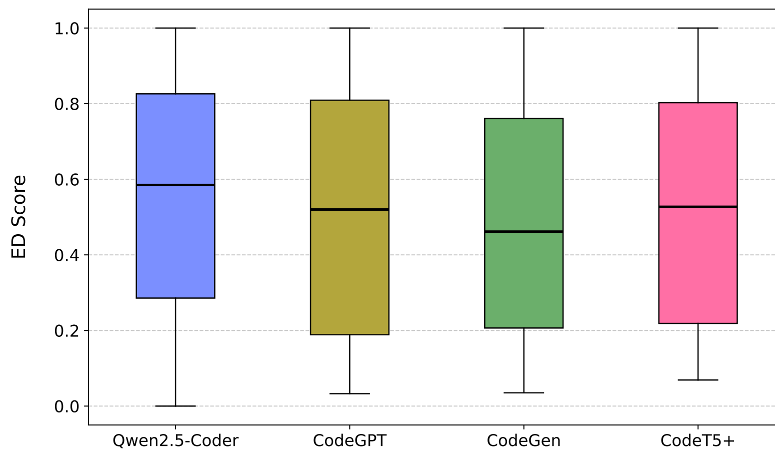

Figure 5.8: ED score distribution of Qwen2.5-Coder and reference models.

5.2.4 Comparison with Proprietary Models

We also compared the semantic performance of Qwen2.5-Coder with some of the most popular proprietary LLMs such as ChatGPT 3.5 [3] and DeepSeekChat-V3 [131], as illustrated in Figure 5.9. While we inferred DeepSeekChat through its public API and evaluated the generated snippets, for the ChatGPT assessment we relied on the scores computed by Liguori et al. in [16] using the same code descriptions.

Additionally, it is important to note that while DeepSeek provides open-source versions of its models (i.e. LLMs with weights and architecture publicly accessible), for our assessment we considered the proprietary web-based version of DeepSeekChat. Achieved results demonstrate that Qwen2.5-Coder is a strong alternative for malicious PowerShell generation when compared to the current state-of-the-art proprietary solutions.

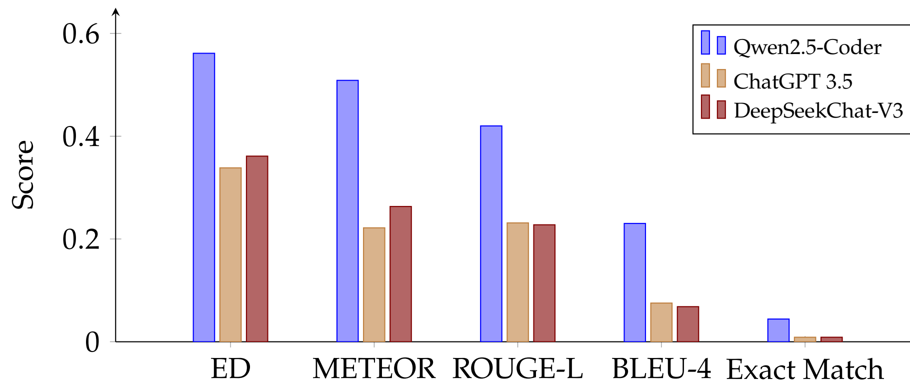

Figure 5.9: Semantic evaluation of Qwen2.5-Coder and closed models.

Figure 5.10 shows the distribution of ED scores on the test dataset for our specialized Qwen2.5-Coder and the closed models ChatGPT 3.5 and DeepSeekChat-V3. Qwen2.5-Coder achieved higher ED similarity to the reference snippets, with 50% of its samples significantly outperforming those generated by the proprietary models.

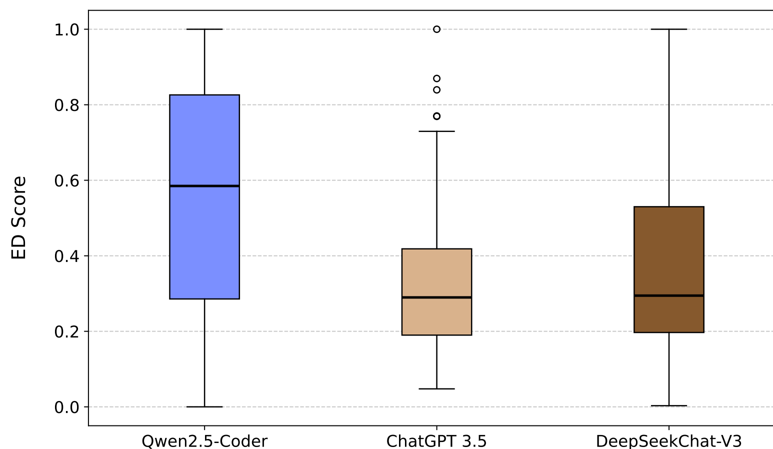

Figure 5.10: ED score distribution of Qwen2.5-Coder and closed models.

While ChatGPT and DeepSeekChat exhibited a less variable distribution of ED scores, the closed models also demonstrated lower similarity to the code references. Additionally, ChatGPT produced a few high-score outliers above 0.7, indicating that it only achieved strong similarity on a small subset of samples.

5.2.5 Fine-Tuning Impact

To assess the fine-tuning impact on the semantic performance of Qwen2.5-Coder, we compared two different versions of the LLM, the base version (before being fine-tuned), and the specialized version (after being fine-tuned with the reference ground truth dataset). Achieved scores, as illustrated in Figure 5.11, demonstrate the significant improvement of the semantic performance of the model after being fine-tuned with the reference dataset. In fact, the specialized LLM registered a significant improvement across all the output similarity metrics, when compared to the base version of the model. This highlights the strong and positive impact of the fine-tuning process in adapting the base model to generate malicious PowerShell following the semantic patterns of the reference samples.

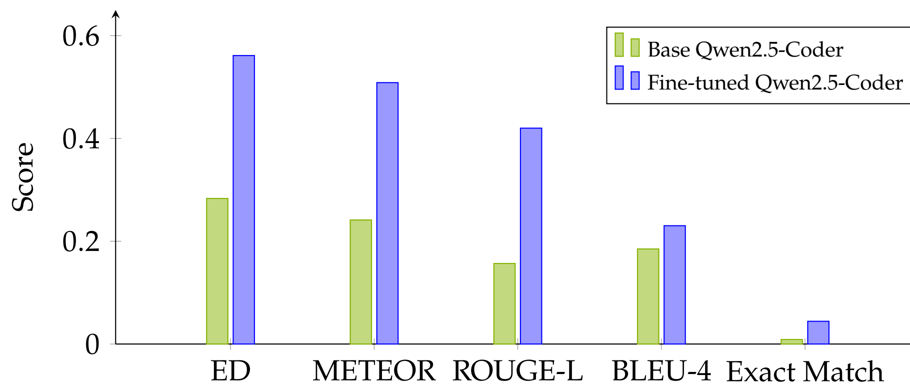

Figure 5.11: Semantic evaluation of the base and fine-tuned versions of Qwen2.5-Coder.

The positive impact of fine-tuning is also reflected by the distribution of ED scores of the base and fine-tuned versions of Qwen2.5-Coder, as represented in Figure 5.12. In particular, the base model only registered a few high-score outliers, with 50% of the scores exhibiting values higher than 0.2. In contrast, 50% of the samples produced by our specialized LLM registered similarity scores above 0.5.

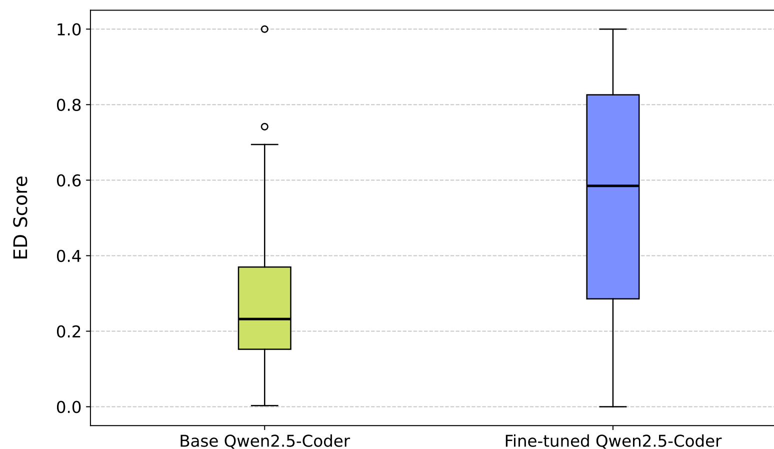

Figure 5.12: ED score distribution of the base and fine-tuned versions of Qwen2.5-Coder.

5.2.6 Extended Ground Truth Dataset Impact

Up to this point, conducted experiments focused on evaluating the selected models fine-tuned only with the reference ground truth dataset from [16]. To assess the training benefits of employing our extended dataset (previously introduced in Chapter 3), we compared the semantic evaluation performance of two different versions of Qwen2.5-Coder, one fine-tuned only with the original reference dataset, and the other fine-tuned with the extended ground truth dataset.

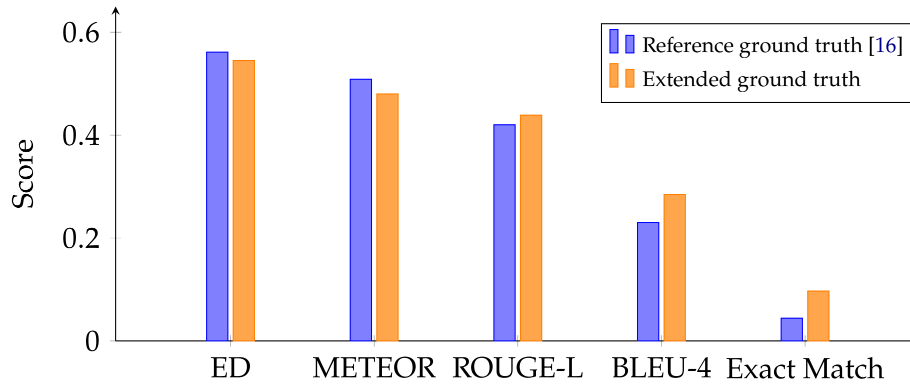

Figure 5.13: Semantic evaluation of Qwen2.5-Coder fine-tuned with different datasets.

As illustrated in Figures 5.13 and 5.14, the ED, METEOR and ROUGE-L scores exhibited negligible variation between the two versions of Qwen2.5-Coder. This happens since these metrics focus on overall structure and semantic similarity, areas where the model fine-tuned with the reference dataset already performed well. In contrast, the model fine-tuned with our extended dataset demonstrated more substantial improvements in BLEU-4 and Exact Match, stricter metrics that emphasize longer n-gram overlaps and exact matches. This highlights the positive impact of using a larger and more diverse training dataset, which primarily enhanced output precision rather than general structure or meaning.

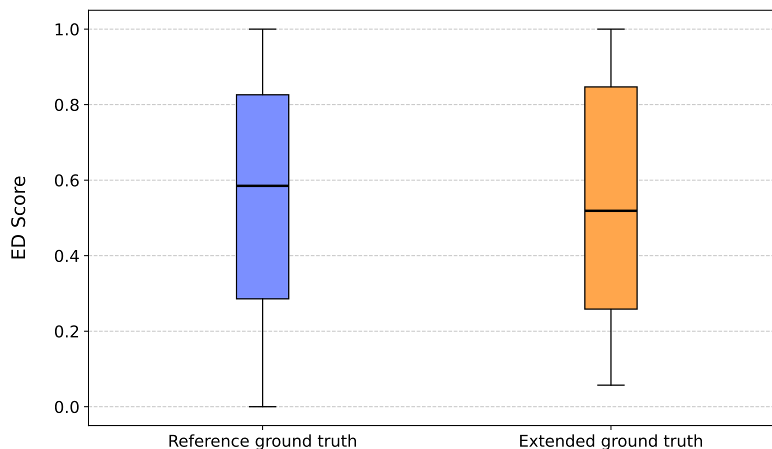

Figure 5.14: ED score distribution of Qwen2.5-Coder fine-tuned with different datasets.

5.3 Functional Assessment

Syntactic and semantic assessments provide useful insights into the quality of the snippets generated by RedShell. However, these evaluation procedures are insufficient to validate the real-world effectiveness of the produced samples, as syntactic and semantic correctness per se does not guarantee that the PowerShell code achieves its intended malicious behavior. To address this limitation, we conducted a functional evaluation of the malicious PowerShell snippets generated by our tool. The code samples were executed within a controlled environment that simulates a realistic pentesting scenario while ensuring safe and isolated PowerShell execution.

5.3.1 Controlled Environment

To ensure the safe execution of the malicious PowerShell samples, we built a controlled environment consisting of two Virtual Machines (VMs), the attacker VM and the target VM, representing the roles of a pentester and a potential victim, respectively. We deployed the testing VMs using the Proxmox Virtual Environment [132], an open-source platform that provides a web-based user interface for managing Linux and Windows VMs and containers. The VM creation process in Proxmox is powered by QEMU (Quick Emulator) and KVM (Kernel-based Virtual Machine), which together facilitate the management of multiple VMs, while optimizing the allocation of the computer resources required for their correct operation .

By conducting our experiments in virtualized environments, we aimed to take advantage of the isolation they provide to safely execute potentially harmful code. In particular, our controlled environment offers the following safety properties:

Software Isolation. The malicious code was executed exclusively on the attacker and target VMs, benefiting from the isolation level provided by virtualized environments.

State Management. Persistent malicious effects caused by the code execution, including unexpected bugs or threats to system integrity, were mitigated through checkpointing and rollback mechanisms, typically available in virtualization platforms such as Proxmox. In particular, we created a snapshot for both VMs in their initial clean state, to allow recovery if the system became damaged or compromised.

Network Isolation. Both VMs were deployed within a segregated network, ensuring complete isolation for the experiments. This setup restricts the communication between VMs to the testing network, preventing their interaction with external hosts to avoid causing potential damage to systems outside the simulation environment. Additionally, both VMs have Internet access, to support the download of malicious software and offensive tools required to execute some of the generated snippets.

Secure Connections. The target VM is accessible only via a secure SSH connection from within the testing network. This happens since most of the malicious code originating from the attacker VM will be remotely executed in the target VM, meaning that external hosts must not be able to directly connect to this machine, to avoid potential damage. In contrast, the attacker VM is reachable via SSH from authorized external hosts, providing an entry point to the testing environment, from which we can control the attacker machine and conduct the pentesting simulation.

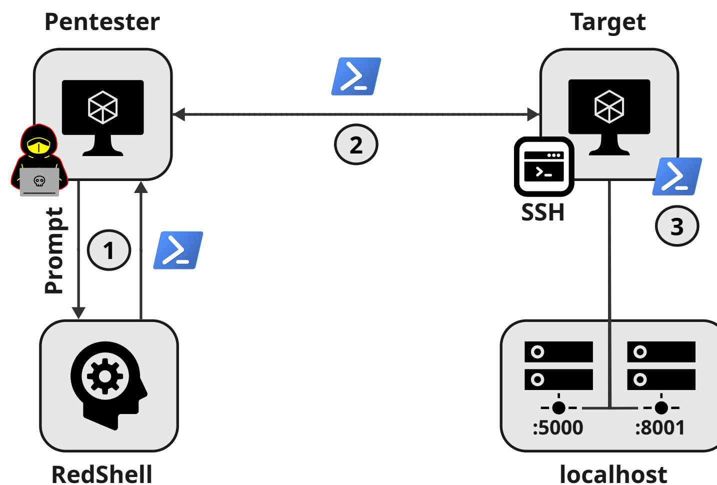

Figure 5.15: Controlled environment setup for the functional evaluation.

Figure 5.15 illustrates the setup of the controlled environment used for malicious PowerShell execution. The simulation consists of three sequential operations, represented by steps (*1*), (*2*), and (*3*) in Figure 5.15. The three operations can be defined as:

1. Pentester submits a prompt to RedShell, describing the intended malicious goal, and receives the corresponding code snippet generated by our tool.

2. The malicious code is remotely executed on the target host via SSH.

3. The code execution triggers an interaction with the target host, potentially targeting the services running on localhost. The produced events/outputs are manually validated by pentesters to ensure that the intended malicious effects were achieved.

The three operations supported by our controlled environment also illustrate a realistic RedShell use-case, describing how our tool can be employed by security professionals to perform pentesting activities with more automation. Furthermore, our testing setup was designed as a generic and realistic pentesting scenario, providing flexibility to be easily extended to more complex and large-scale scenarios.

We now present in detail the four individual components that are part of our simulation, describing how they can be easily adapted to address alternative scenarios:

RedShell. Our malicious PowerShell generator corresponds to the specialized version of Qwen2.5-Coder, fine-tuned with the extended ground truth dataset, described in Chapter 3. We selected Qwen2.5-Coder to be incorporated in our tool since it was the model that generated snippets with higher semantic correctness, when compared to the performance of the LLMs evaluated in Section 5.2. The machine used for training the model was also employed to host its fine-tuned version, which was loaded into VRAM to allow local inference. To generate malicious code, Pentesters must submit their prompts to the model through a web interface connected to the machine that hosts the LLM, which is safely located outside the testing environment. In an alternative scenario, RedShell could also be accessed via a REST endpoint.

Pentester VM. For the attacking machine, we employed a lightweight x86_64 Linux VM with two CPU cores, 1 GB of RAM, and 5 GB of disk storage. The VM was also equipped with PowerShell version 7.5.1, a recent version of PowerShell known as pwsh, which supports the creation of remote PowerShell sessions over SSH. Although the generated snippets target Microsoft Windows vulnerabilities, a Linux-based attacker system was chosen to introduce platform heterogeneity, capturing a realistic Pentesting environment that emphasizes PowerShell's cross-platform capabilities. Additionally, in our simulation, the pentester operated with root privileges within the attacker VM, under the assumption that the target network had already been breached and the attacking machine compromised. This setup reflects a typical post-exploitation scenario where PowerShell is widely employed by ethical hackers. Alternatively, this environment also illustrates an internal pentesting context, where the attack is conducted by an authorized security professional that emulates the behavior of a malicious insider such as an unsatisfied employee.

Target VM. For the target machine, we employed a Microsoft Windows 10 Pro VM with 4 CPU cores, 4 GB of RAM, and 32 GB of disk storage. The VM was also equipped with PowerShell versions 7.5.1 and 5.1. While the pre-installed PowerShell version 5.1 was employed to exploit security vulnerabilities of the target host, the pwsh version was used to enable the creation of remote PowerShell sessions over SSH. An alternative approach to support remote PowerShell execution in the testing environment would involve either using the Windows Remote Desktop Protocol (RDP) or employing the PowerShell Remoting utility via the Windows Remote Management (WinRM) service, rather than directly invoking the pwsh executable. The remaining system configurations in the target machine, including the settings of the Windows Defender antivirus, were retained in their default state.

Services in localhost. To perform the pentesting activities, we embedded security vulnerabilities into the target VM, creating two vulnerable Python REST servers running on localhost. In particular, the server running on port 5000 holds a leaked credential in

the headers of the HTTP GET request, while the server running on port 8001 allows the execution of PowerShell code with administrative privileges, a vulnerability that can be used to inject malicious payloads and achieve privilege escalation. Our setup captures a scenario of vulnerable software exploitation, where the target servers can either represent services running on development state, or installed software with unpatched security flaws. Additionally, in a real-world scenario, the target host could also be employed as an entry point for a malicious actor to compromise services hosted in Microsoft Azure cloud services, namely using PowerShell offensive modules such as MicroBurst [103].

5.3.2 Pentesting Methodology

To perform the functional evaluation of the snippets generated by RedShell, we followed a sequence of offensive steps commonly employed in ethical hacking scenarios. We designed an offensive pipeline to fully compromise the target VM, performing an effective intrusion into the target host, followed by the application of privilege escalation techniques to allow the intruder to obtain full administrative control over the machine. Once elevated privileges are obtained, the attacker is able to perform further unlimited malicious activities, including disabling defensive mechanisms such as the Windows Defender, executing arbitrary offensive code, and extracting legitimate credentials from the target VM.

In fact, our testing approach aligns with the methodologies commonly employed in ethical hacking competitions, such as CTF challenges, where participants try to perform the exploitation of a vulnerable VM. To complete the challenges, ethical hackers take advantage of various discovery and exploitation techniques to retrieve hidden flags from the target system. Similarly, in our setup, we employed hidden flags to assess whether an attack has successfully exploited a specific vulnerability embedded into the environment. Additionally, we manually verified whether the code execution triggered the expected malicious effects in the controlled environment.

The evaluation was conducted following a White-Box approach, under the assumption that the attacker possessed prior knowledge of the vulnerabilities present in the services running on the localhost of the target VM. For each step defined in the testing pipeline, we prompted RedShell for a specific PowerShell payload to perform the intended malicious action. The snippets that our tool generated for each attack phase also demonstrate how PowerShell can be employed by pentesters to fully compromise the target VM.

While real-world pentesting is usually conducted through a trial-and-error process of discovery and exploitation of security flaws, we chose to follow a more strict testing methodology that allowed us to benchmark RedShell's capabilities across different stages of an attack. Specifically, we designed a testing pipeline covering various tactics from the MITRE ATT&CK framework, to assess the diversity and relevance of the payloads generated by our tool across varying offensive scenarios. Furthermore, our testing methodology was able to capture the dependencies between sequential phases of an attack. For

instance, the successful execution of a network discovery operation can be perceived as a pre-condition for identifying exploitable hosts in the target network. By incorporating such dependencies in our functional assessment, we ensured that the evaluation reflected realistic attack progressions, thus providing a methodical and comprehensive validation of the effectiveness of RedShell in various offensive scenarios.

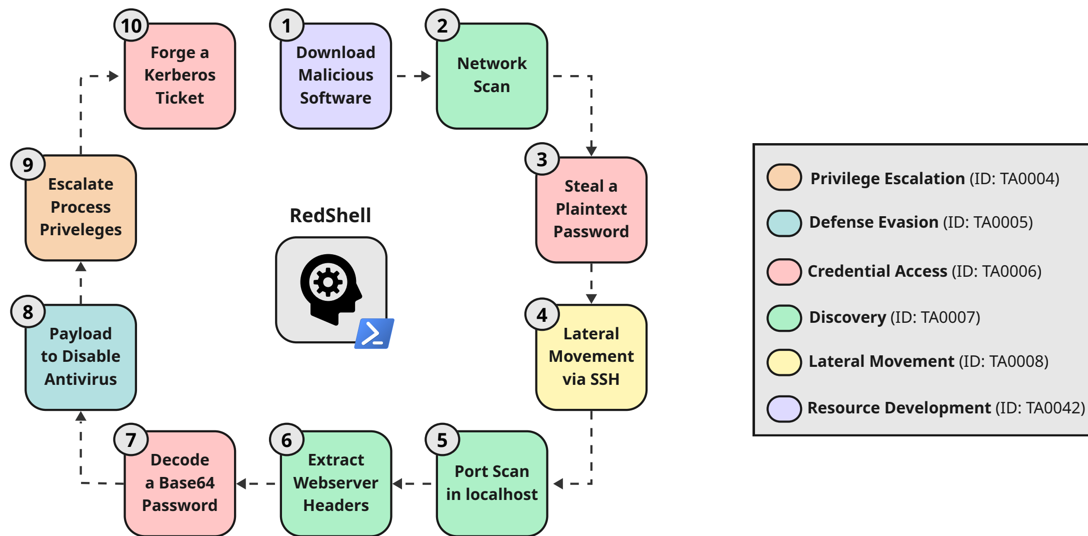

Figure 5.16: Offensive pipeline for the functional evaluation.

Figure 5.16 illustrates the offensive pipeline employed for the functional evaluation of the snippets generated by RedShell, consisting of 10 sequential steps to perform the exploitation of the target VM. Distinct colors denote the corresponding MITRE ATT&CK tactics associated with each phase of the attack, enumerated alongside with their respective unique identifiers as defined within the MITRE ATT&CK framework. We can perceive that the testing pipeline covers 6 different offensive tactics, representing almost half of the 14 tactics that compose the attacking matrix from MITRE ATT&CK [17]. In particular, the most covered tactics are *Discovery* and *Credential Access*, both represented in three distinct phases in the offensive pipeline. These tactics were also strongly covered by the ground truth dataset (Chapter 3) since they include offensive techniques often employed by ethical hackers in real-world pentesting scenarios.

We now present in more detail the offensive purpose of each step in the testing pipeline:

1. **Download Malicious Software.** In the initial phase of the attack, the pentester employs RedShell to generate a PowerShell snippet that downloads the script `Invoke-PortScan.ps1` from the Nishang [101] framework, to support the discovery of potentially exploitable hosts within the target network.

2. **Network Scan.** The pentester takes advantage of the utilities offered by the previous downloaded script to execute a ping sweep across the local network segment `192.168.1.0/24`. The scan specifically aims to identify hosts that have TCP port 22

open, indicating potential support for remote SSH connections, and the possibility of lateral movement within the target network.

3. **Steal a Plaintext Password.** The results of the preceding ping sweep reveal the IP address of the target VM, with TCP port 22 open and accepting SSH connections. It is assumed that the attacker has already obtained the username `admin`, associated with an administrative account on the target system, possibly through a prior phishing campaign. Since the target user account holds a weak password, the pentester is able to perform a brute-force attack using a tool such as Hydra [44]. This process can be executed entirely via PowerShell by downloading both the Hydra executable and a wordlist of weak passwords. Although Hydra is not a native PowerShell module, the downloaded executable can still be invoked by PowerShell.

4. **Lateral movement via SSH.** Using the previously obtained credentials, the attacker is able to start a remote PowerShell session in the target VM, effectively performing a lateral movement to the target system. The remote session can be created either by employing the SSH utilities from pwsh, or by establishing a standard SSH connection to the target VM, followed by the creation of a PowerShell process to allow remote PowerShell code execution.

5. **Port Scan in localhost.** Once the pentester gets access to the target system, enumeration techniques can be employed to identify local listening ports and active services, and discover further exploitation opportunities.

6. **Extract Webserver Headers.** The previous port scan operation identified two local Python servers running on ports 5000 and 8001. To perform the exploitation of the server running on local port 5000, the pentester may execute a PowerShell snippet to extract the headers of the corresponding HTTP GET request to the target server, unveiling a leaked credential encoded in the Base64 format.

7. **Decode a Base64 Password.** PowerShell utilities can be employed to decode the previously extracted password from its Base64 format to the corresponding plaintext string representation, effectively exposing a legitimate credential that can be used to obtain access to sensitive data and services.

8. **Payload to Disable the Antivirus.** More complex malicious activities on the target VM are typically blocked by the Windows Defender antivirus, thus preventing further exploitation of the system. Offensive actions such as downloading and executing malicious PowerShell modules on the target machine are blocked by the real-time protection mechanisms of Windows Defender, even when performed with administrative privileges. To circumvent these defenses, the attacker can execute a malicious PowerShell payload designed to disable the antivirus.

9. **Escalate Process Privileges.** Although the intruder operates a PowerShell remote session as an administrative user, the PowerShell process itself does not inherit administrative privileges, which are necessary for modifying system configurations such as the antivirus settings. To escalate privileges and execute the payload that disables Windows Defender protections, the attacker exploits the vulnerable Python server running on port 8001. This unpatched service accepts a PowerShell command as a query parameter for subsequent execution. Furthermore, the server runs with elevated privileges, allowing the injection and execution of malicious PowerShell payloads with administrative rights. By injecting the previously crafted payload, the attacker disables the protections provided by Windows Defender, causing the target VM to be fully exposed to further exploitation.

10. **Forge a Kerberos Ticket.** With the target VM fully compromised, the pentester is able to download malicious PowerShell modules such as Mimikatz [25], which can be employed to extract password hashes and sensitive data from memory. The extracted secrets can then be used to forge a Kerberos golden ticket, which allow attackers to request access to sensitive resources through a ticket granting service. In our testing scenario, we simulated the creation of a malicious golden ticket that could be employed by the attacker in further malicious activities.

5.3.3 Evaluation Metrics

To assess the functional performance of the PowerShell snippets generated by RedShell, we employed the following evaluation metrics:

- **Effectiveness.** Measures the capability of the generated snippets in producing the expected malicious effects within the controlled environment. Effectiveness was determined by having pentesters manually verifying if the code execution produced the expected outputs or events, which were represented by flags associated with each attack phase. A match between the observed events and the respective flag determines whether a snippet is considered to be effective or not, in the context of our pentesting simulation. Table 5.3 presents the flags associated with each step in the offensive pipeline, along with the corresponding MITRE ATT&CK tactics, techniques, and sub-techniques used to perform the intended malicious activities.

- **Effectiveness ED.** A distance metric was used to measure the magnitude of the code modifications required to fix the non-effective samples and restore their intended malicious capabilities. We computed the ED between the faulty snippets and their respective corrected versions, normalizing the resulting score between 0 (perfect mismatch) and 1 (perfect matching), as performed in the semantic assessment (Section 5.2). It is important to note that higher scores imply greater similarity between the faulty snippets and their corresponding corrected versions, and also suggest that only minor modifications were required to perform the intended corrections. While

this metric was defined under the assumption that non-effective samples would be manually corrected by pentesters, alternative approaches could leverage RedShell or another LLM to automate the same correction process as a post-processing feature.

- **Correctness.** Measures the adherence of the generated snippets to the best offensive PowerShell practices and pentesting methodologies. Correct samples must follow the expected tactics, techniques, and sub-techniques described by the MITRE ATT&CK framework, as presented in Table 5.3. Additionally, correct snippets must also be aligned with the most common pentesting approaches. For instance, malicious PowerShell modules should be loaded directly into memory rather than saved to disk, following a standard strategy to avoid leaving persistent traces of malicious activities and evade detection mechanisms. While samples that deviated from the expected offensive patterns were classified as incorrect, their execution may still produce the intended malicious effects and thus be considered effective.

It was important to consider both the effectiveness and correctness of the generated samples to achieve a more complete evaluation of their functional capabilities. This happens since effective snippets may still lack correctness if they do not adhere to the expected pentesting methodologies, while correct snippets can nevertheless fail to produce the intended offensive outcomes, even when they are aligned with the reference practices.

Table 5.3: Flags and tactics employed in the functional evaluation.

Attack	Tactical Group	Technique	Sub-Technique	Validation Flag
Phase 1	Resource Development	Obtain Capabilities	Tool	`download{Invoke-PortScan.ps1}`
Phase 2	Discovery	Remote System Discovery	—	`ip{192.168.1.171}`
Phase 3	Credential Access	Brute Force	Password Spraying	`pw{12345}`
Phase 4	Lateral Movement	Remote Services	Windows Remote Management	`ssh{192.168.1.171 @admin}`
Phase 5	Discovery	Network Service Discovery	—	`ports{5000,8001}`
Phase 6	Discovery	Process Discovery	—	`pw{U3Ryb25nQ3J1ZC 1zcWwtc2VydmVyMzMh}`
Phase 7	Credential Access	Brute Force	Password Cracking	`plaintext{Strong Cred-sql-server33!}`
Phase 8	Defense Evasion	Impair Defenses	Disable or Modify Tools	`disable{windows-defender}`
Phase 9	Privilege Escalation	Exploitation for Privilege Escalation	—	`process{admin}`
Phase 10	Credential Access	Steal or Forge Kerberos Tickets	Golden Ticket	`forge{ticket}`

5.3.4 Experimental Results

We present in Table 5.4 the results of the functional evaluation of the malicious PowerShell snippets produced by RedShell. In particular, we report the effectiveness, edit distance, and correctness of the snippets generated by our tool for each attack phase in our testing pipeline. The code samples were manually classified as effective or non-effective, and correct or incorrect, according to the evaluation metrics described earlier, with the non-effective snippets being reported along with their respective flaws.

Table 5.4: Functional evaluation of RedShell.

Attack Phase	Effectiveness	Edit Distance	Correctness
1. Download Malicious Software	✗ Wrong URL	0.7862	✓
2. Network Scan	✓	—	✓
3. Steal a Plaintext Password	✗ Wrong URL	0.5808	✓
4. Lateral Movement via SSH	✗ Invalid args	0.7656	✓
5. Port Scan in localhost	✓	—	✓
6. Extract Webserver Headers	✓	—	✓
7. Decode a Base54 Password	✓	—	✓
8. Payload to Disable Antivirus	✓	—	✓
9. Escalate Process Privileges	✓	—	✓
10. Forge a Kerberos Ticket	✓	—	✓

The snippets generated by RedShell exhibited strong effectiveness and correctness throughout the pentesting simulation. This demonstrates the robust capabilities of our tool in producing PowerShell code that successfully performs the intend malicious activities while being closely aligned with reference pentesting techniques.

Notably, the produced snippets were effective in 7 of the 10 attack phases, and failed to reproduce the expected outcomes in only 3 phases. In particular, the samples generated for steps (*1*) and (*3*) of the pipeline failed to download offensive utilities due to the use of incorrect source URLs. Following that failure, the execution of further malicious activities is also compromised due to the functional dependencies between sequential attack phases. For phase 1, specifically, the failed download of the script `Invoke-PortScan.ps1` from Nishang may compromise the execution of the network scanning operation that was planned for phase (*2*). Additionally, in phase (*3*), the failure to download a list of weak passwords may prevent the execution of the expected SSH brute-force attack using Hydra, thus compromising the lateral movement that was scheduled for phase (*4*).

The lateral movement within the target network is also threatened by the fact that RedShell generated for phase (*4*) a PowerShell command containing invalid arguments. Specifically, the generated command contains the arguments `ComputerName` and `Credential`, which are commonly used by older versions of PowerShell to enter a remote session via the

WinRM service. However, in our testing pipeline, the attacker was expected to employ the new features of pwsh to start a remote PowerShell session via SSH, replacing the legacy arguments by the values `Hostname` and `UserName`, respectively.

Although the identified flaws compromise the expected functionality of the generated snippets, the corresponding EDs to their corrected versions all exceeded a score of 0.5. This suggests than only minor updates would be required to fix the faulty code and produce the intended malicious outcomes. Additionally, correctness was consistently observed across all attack phases, emphasizing the model's strong alignment with the standard offensive techniques described by the MITRE ATT&CK framework.

Further experimentation is required to assess the functional capabilities of our tool in more complex and large-scale offensive scenarios. However, the controlled environment and the baseline scenario that we built for our functional experiments illustrate well a typical use-case where RedShell could be employed by security professionals to conduct real pentesting activities with more automation. In fact, the strong effectiveness and correctness exhibited by the generated samples highlight the potential applicability of our tool across a wide range of offensive cybersecurity operations.

5.3.5 Comparison with a Proprietary Model

Table 5.5 presents the results of the functional performance comparison between RedShell and the popular closed model ChatGPT-3.5 from OpenAI. Both alternatives produced a small number of non-effective snippets (three in the case of RedShell, and two for ChatGPT), demonstrating comparable effectiveness in our pentesting scenario.

Table 5.5: Functional evaluation of ChatGPT-3.5 and RedShell.

	Effectiveness		Correctness	
Attack Phase	**RedShell**	**ChatGPT-3.5**	**RedShell**	**ChatGPT-3.5**
1. Download Malicious Software	✗ Wrong URL	✗ Wrong URL	✓	✗ Save to disk
2. Network Scan	✓	✗ Invalid args	✓	✓
3. Steal a Plaintext Password	✗ Wrong URL	✓	✓	✓
4. Lateral Movement via SSH	✗ Invalid args	✓	✓	✗ No PSSession
5. Port Scan in localhost	✓	✓	✓	✓
6. Extract Webserver Headers	✓	✓	✓	✓
7. Decode a Base54 Password	✓	✓	✓	✓
8. Payload to Disable Antivirus	✓	✓	✓	✓
9. Escalate Process Privileges	✓	✓	✓	✓
10. Forge a Kerberos Ticket	✓	✓	✓	✗ Redundant

In the first step of the pipeline, both models failed to download a malicious script from Nishang due to the use of incorrect source URLs. Additionally, ChatGPT was unable to execute the network scanning procedure scheduled for phase (*2*), as the code it generated employed the arguments `Hosts`, `Ports`, and `Ping`, while the correct command required the use of the values `StartAddress`, `EndAddress`, `ResolveHost`, and `ScanPort`.

ChatGPT outperformed the effectiveness of our tool in phases (*3*) and (*4*). However, it is important to note that, in phase (*4*), the model from OpenAI executed the intended lateral movement following a strategy that lacked correctness. Specifically, the generated snippet opened a standard SSH connection to the target VM without creating an interactive PowerShell remote session, as requested by the testing prompt. This fails to convene with the expected approach of employing the pwsh SSH utilities to directly start a PowerShell session supporting remote code execution in the target host.

Further correctness limitations of ChatGPT were also identified in phases (*1*) and (*10*) of the testing pipeline. In phase (*1*), particularly, the model tried to perform a malicious file download into disk instead of RAM, potentially leaving traces of offensive activities that could be detected by security mechanisms. In phase (*10*), it was assumed that the Mimikatz module was already loaded into memory as a dependency for the expected code execution. However, the snippet generated by ChatGPT attempted to install the command dependencies, despite the testing prompt specifying only the intended execution. The correctness flaws observed in the snippets generated by ChatGPT for phases (*4*) and (*10*) illustrate a case where the proprietary model produced effective samples that nonetheless lacked correctness, as they did not fully align with the expected offensive methodologies. In contrast, all the snippets generated by RedShell were classified as correct, highlighting the specialized knowledge and good offensive coding practices offered by our tool, in opposition to ChatGPT.

Table 5.6: Functional evaluation of the non-effective code from ChatGPT-3.5 and RedShell.

	Effectiveness Flaws		**Edit Distance**	
Attack Phase	**RedShell**	**ChatGPT-3.5**	**RedShell**	**ChatGPT-3.5**
1. Download Malicious Software	Wrong URL	Wrong URL	0.7862	0.4897
2. Network Scan	—	Invalid args	—	0.4222
3. Steal a Plaintext Password	Wrong URL	—	0.5808	—
4. Lateral Movement via SSH	Invalid args	—	0.7656	—

Furthermore, RedShell was able to generate malicious PowerShell without any ethical restrictions while the model from OpenAI presented more reluctance in producing offensive code. In those cases, additional prompting was required to clarify the task intentions and manipulate the model's ethical safeguards. Notably, RedShell was also able to achieve a comparable effectiveness performance to ChatGPT, a much larger and complex model. In addition to that, our solution offers properties that closed LLMs fail to provide, such as

privacy, specialization, and the absence of ethical boundaries, as described in Chapter 4.

Table 5.6 presents the EDs between the non-effective snippets generated by ChatGPT and RedShell, and their respective corrected versions. Reported values show that all the faulty samples generated by our tool achieved similarity scores above 0.5, while ChatGPT scores failed to reach the same mark. In fact, despite providing comparable performances in terms of code effectiveness, RedShell obtained higher EDs for the non-effective samples, when compared to results exhibited by ChatGPT. This suggests that our solution was able to produce PowerShell code with stronger alignment with the expected snippets, thus facilitating the correction process in the cases where the output failed to reproduce the intents of the attack.

5.3.6 Fine-Tuning Impact

Table 5.7 presents the results of the functional performance comparison between the base and the fine-tuned versions of Qwen2.5-Coder, providing insights into the fine-tuning impact on the quality of the generated samples.

Table 5.7: Functional evaluation of the the base version of Qwen2.5-Coder and RedShell.

	Effectiveness		Correctness	
Attack Phase	**RedShell**	**Base Model**	**RedShell**	**Base Model**
1. Download Malicious Software	✗ Wrong URL	✗ Wrong URL	✓	✗ Save to disk
2. Network Scan	✓	✗ Invalid args	✓	✓
3. Steal a Plaintext Password	✗ Wrong URL	✗ Wrong URL	✓	✓
4. Lateral Movement via SSH	✗ Invalid args	✗ Invalid args	✓	✓
5. Port Scan in localhost	✓	✓	✓	✓
6. Extract Webserver Headers	✓	✓	✓	✓
7. Decode a Base54 Password	✓	✓	✓	✓
8. Payload to Disable Antivirus	✓	✓	✓	✗ Redundant
9. Escalate Process Privileges	✓	✓	✓	✓
10. Forge a Kerberos Ticket	✓	✓	✓	✓

The base model exhibited a strong performance in terms of effectiveness, producing functional snippets for 6 of the 10 attack phases. This highlights the strong offensive PowerShell knowledge already offered by the original LLM. The non-effective snippets were generated in the first four steps of the offensive pipeline. In particular, the snippets generated for phases (*1*) and (*3*) failed to download the intended utilities due to the use of incorrect source URLs, while the commands produced for phases (*2*) and (*4*) employed incorrect arguments.

RedShell demonstrated effectiveness comparable to the base version of Qwen2.5-Coder, generating 7 effective snippets compared to the 6 functional samples produced by the base model. Notably, the fine-tuned model showed a significant improvement in phase (*2*), where legacy command arguments were correctly replaced with the expected values. This suggests that the fine-tuning process enhanced the capabilities of the original LLM by integrating additional specialized knowledge. Further positive effects of fine-tuning were observed in the correctness improvement of the generated samples. For instance, the base model performed a download into disk instead of RAM in phase (*1*), and executed a redundant command to disable Windows Defender in phase (*8*). These correctness limitations were effectively mitigated through fine-tuning, as reflected in the reference offensive patterns observed in the snippets generated by our tool.

All the effectiveness flaws previously identified occurred within the first four attack phases, including those observed in the samples produced by ChatGPT. This may indicate a performance pattern across different models. Specifically, all the LLMs struggled to perform the download of malicious software due the use of incorrect source URLs. This limitation likely arises from the fact that URLs are frequently updated over time, while LLMs typically perform better in generalization tasks rather than memorization. From a practical point of view, pentesters are expected to replace the placeholder URLs with those appropriate to their context and requirements. Similarly, the frequently use of legacy or incorrect command parameters suggests that further training with updated code examples could be beneficial for enhancing the capabilities of RedShell.

Table 5.8 presents the EDs between the non-effective snippets generated by Qwen2.5-Coder and RedShell, and their respective corrected versions. Reported values show that all the faulty samples generated by our tool exhibited higher similarity to their references when compared to the ones produced by the base model.

Table 5.8: Functional evaluation of the non-effective code from the base version of Qwen2.5-Coder and RedShell.

	Effectiveness Flaws		Edit Distance	
Attack Phase	**RedShell**	**Base Model**	**RedShell**	**Base Model**
1. Download Malicious Software	Wrong URL	Wrong URL	0.7862	0.4897
2. Network Scan	—	Invalid args	—	0.3778
3. Steal a Plaintext Password	Wrong URL	Wrong URL	0.5808	0.4494
4. Lateral Movement via SSH	Invalid args	Invalid args	0.7656	0.6296

Similarly to ChatGPT, Qwen2.5-Coder achieved a comparable performance to the one registered by RedShell in terms of code effectiveness. However, our tool outperformed the similarity scores achieved by the base model for the non-effective samples. This suggests that the faulty snippets generated by RedShell could be corrected through minor updates, while the outputs of the base LLM would require more substantial modifications.

5.4 Summary

In this chapter, we validated the generation capabilities of our specialized models by conducting three different experimental procedures. First, we evaluated the syntactic correctness of the produced snippets by employing a static PowerShell code checker from Microsoft, the PSScriptAnalyzer. Next, we performed a semantic evaluation by computing the standard output similarity metrics between the generated samples and their respective references in the test dataset. The fine-tuned models all exhibited a strong performance for both syntactic and semantic assessments, with Qwen2.5-Coder outperforming Llama3.1 and Qwen2.5 across all the output distance metrics. Following that observation, we selected the fine-tuned version of Qwen2.5-Coder as the best candidate to be incorporated in RedShell. Finally, we evaluated the functional quality of the snippets generated by our tool by performing their execution in a controlled environment that simulated a realistic pentesting scenario. While further experimentation is required to assess the functional capabilities of RedShell in larger contexts, the observed malicious effects within our testing environment suggests that our tool can be effectively employed by pentesters to provide more automation to a wide range of offensive cybersecurity operations.

6 Conclusions and Future Work

LLMs have registered a high popularity in recent years, with the AI-community actively searching for new approaches to create small, and yet powerful, generative models. In addition, engineers are taking advantage of these novel technologies to provide more automation to complex and time-consuming tasks. In the offensive cybersecurity field, various AI-based solutions have been proposed to assist ethical hackers in pentesting scenarios. Generative models, in particular, provide a strong baseline to automate malicious code generation in pentesting scenarios.

6.1 Conclusions

In this thesis, we introduced RedShell, a novel tool designed to assist pentesters by generating malicious PowerShell code. In particular, we provided the following contributions to the ongoing research on the application of LLMs in offensive cybersecurity scenarios:

- Extended a dedicated ground truth dataset for malicious PowerShell, establishing a valuable resource for future research on fine-tuning LLMs in offensive code generation.
- Created a novel tool named RedShell leveraging a specialized LLM to assist ethical hackers targeting Microsoft Windows vulnerabilities through malicious PowerShell generation.
- We demonstrated that RedShell is capable of producing syntactically valid and semantically meaningful malicious snippets, closely aligned with reference samples.
- Finally, we validated RedShell's functional capabilities by performing the execution of the generated snippets in a controlled environment, demonstrating that the code samples effectively reproduced the intended malicious behaviors within our pentesting simulation. This highlights the potential applicability of our tool in more vast offensive contexts, as an effective approach to provide more automation to traditional cybersecurity workflows.

6.2 Future Work

Future work might extend the functionality of RedShell by improving its generation capabilities. Furthermore, the work presented in this thesis is expected to serve as a baseline for further research in the application of Generative AI, in general, and LLMs, in particular, in the cybersecurity field. Hopefully, our work may inspire additional efforts to employ state-of-the-art AI-based approaches to enhance both defensive and offensive cybersecurity, and contribute to prevent users in the cyberspace to be harmed by malicious actors employing offensive AI as an unethical tool for their disrupting activities. We now present in more detail possible future directions for the work presented in this thesis:

- Build novel malicious code ground truth datasets and provide further extensions to our training data, either by incorporating novel snippets or by employing state-of-the-art techniques such as synthetic data generation (possibly through generic LLMs such as ChatGPT).
- Extend our experimental evaluation to other state-of-the-art LLMs, including the updated versions of the models that we selected for our assessment, particularly Qwen3-Coder and the GPT-5 version of ChatGPT.
- Assess the functional capabilities of our tool in larger offensive scenarios, having real pentesters actively using RedShell to produce the intended malicious effects within the target systems.
- Integrate RedShell in a larger agentic framework composed of multiple LLMs fine-tuned in different tasks, to automate all the phases of a pentest audit, performing vulnerability assessment, exploitation, and reporting, with minimal human intervention.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (cit. on pp. 1, 13, 18).

[3] OpenAI. *ChatGPT: Overview and Features*. 2025. URL: https://openai.com/chatgpt/overview/ (visited on 2025-05-22) (cit. on pp. 1, 3, 4, 20, 23, 51).

[4] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Scao, S. Gugger, and A. Rush. "Transformers: State-of-the-Art Natural Language Processing". In: *EMNLP 2020 - Conference on Empirical Methods in Natural Language Processing, Proceedings of Systems Demonstrations*. 2020, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6 (cit. on pp. 1, 13).

[5] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. "Large Language Models for Software Engineering: Survey and Open Problems". In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53. DOI: 10.1109/ICSE-FoSE59343.2023.00008 (cit. on pp. 1, 22).

[6] P. Vats, M. Mandot, and A. Gosain. "A Comprehensive Literature Review of Penetration Testing and Its Applications". In: *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. IEEE, 2020, pp. 674–680. DOI: 10.1109/ICRITO48877.2020.9197961 (cit. on pp. 1, 5, 7).

[7] R. Bessa, J. Trindade, R. Claro, and J. M. Lourenço. "RedShell: A Generative AI-Based Approach to Ethical Hacking". In: *Proceedings of the 16th Simpósio de Informática*. 2025. URL: https://www.hlt.inesc-id.pt/~eugenio.ribeiro/inforum2025/papers/INForum_2025_paper_5.pdf (cit. on pp. 3, 4).

[8] A. Applebaum, D. Miller, B. Strom, H. Foster, and C. Thomas. "Analysis of automated adversary emulation techniques". In: *Proceedings of the Summer Simulation Multi-Conference*. Society for Computer Simulation International, 2017, pp. 169–180. DOI: 10.22360/summersim.2017.scsc.016 (cit. on p. 5).

[9] G. Weidman. *Penetration testing: a hands-on introduction to hacking*. No starch press, 2014. ISBN: 9781593275648 (cit. on pp. 5, 7, 11).

[10] S. G. Bianou and R. G. Batogna. "PENTEST-AI, an LLM-Powered Multi-Agents Framework for Penetration Testing Automation Leveraging Mitre Attack". In: *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2024, pp. 763–770. DOI: 10.1109/CSR61664.2024.10679480 (cit. on pp. 5–8, 24).

[11] G. Lyon. *Nmap: Network Mapper*. 1997–2025. URL: https://nmap.org/ (visited on 2025-05-22) (cit. on pp. 5, 12).

[12] I. Tenable. *Nessus: Vulnerability Assessment Solution*. 2025. URL: https://www.tenable.com/products/nessus (visited on 2025-05-22) (cit. on pp. 5, 12).

[13] P. Ltd. *Burp Suite: Application Security Testing Software*. 2025. URL: https://portswigger.net/burp (visited on 2025-05-22) (cit. on pp. 5, 12).

[14] Metasploit. *Metasploit Framework*. 2024. URL: https://www.metasploit.com/ (visited on 2025-05-22) (cit. on pp. 5, 12).

[15] R. E. L. De Jimenez. "Pentesting on web applications using ethical - Hacking". In: *2016 IEEE 36th Central American and Panama Convention (CONCAPAN XXXVI)*. IEEE, 2016, pp. 1–6. DOI: 10.1109/CONCAPAN.2016.7942364 (cit. on p. 7).

[16] P. Liguori, C. Marescalco, R. Natella, V. Orbinato, and L. Pianese. "The Power of Words: Generating PowerShell Attacks from Natural Language". In: *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. USENIX Association, 2024, pp. 27–43. URL: https://www.usenix.org/conference/woot24/presentation/liguori (cit. on pp. 8, 11, 22, 26, 28, 30–32, 35, 50, 51, 54).

[17] M. Corporation. *MITRE ATT&CK Framework*. 2024. URL: https://attack.mitre.org/ (visited on 2025-05-22) (cit. on pp. 8, 24, 25, 30, 59).

[18] B. E. Strom, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas. "MITRE ATT&CK: Design and Philosophy". In: *Technical report*. The MITRE Corporation, 2018. URL: https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf (cit. on p. 8).

[19] N. Lasky, B. Hallis, M. Vanamala, R. Dave, and J. Seliya. “Machine Learning Based Approach to Recommend MITRE ATT&CK Framework for Software Requirements and Design Specifications”. In: *arXiv preprint arXiv:2302.05530* (2023). URL: https://arxiv.org/abs/2302.05530 (cit. on p. 10).

[20] A. B. Ajmal, S. Khan, M. Alam, A. Mehbodniya, J. Webber, and A. Waheed. “Toward Effective Evaluation of Cyber Defense: Threat Based Adversary Emulation Approach”. In: *IEEE Access* 11 (2023), pp. 70443–70458. DOI: 10.1109/ACCESS.2023.3272629 (cit. on p. 10).

[21] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen. “ExploitGen: Template-augmented exploit code generation based on CodeBERT”. In: *Journal of Systems and Software* 197 (2023). DOI: 10.1016/j.jss.2022.111577 (cit. on pp. 10, 26, 27).

[22] N. Antunes and M. Vieira. “Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services”. In: *2011 IEEE International Conference on Services Computing*. IEEE, 2011, pp. 104–111. DOI: 10.1109/SCC.2011.67 (cit. on p. 10).

[23] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang. “Fuzzing: A Survey for Roadmap”. In: *ACM Computing Surveys* 54.11s (2022). DOI: 10.1145/3512345 (cit. on p. 10).

[24] P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic. “EVIL: Exploiting Software via Natural Language”. In: *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 321–332. DOI: 10.1109/ISSRE52982.2021.00042 (cit. on pp. 11, 22, 26, 27).

[25] B. Delpy. *Mimikatz*. 2011. URL: https://github.com/gentilkiwi/mimikatz (visited on 2025-05-22) (cit. on pp. 11, 33, 61).

[26] Fortra. *Cobalt Strike*. 2012. URL: https://www.cobaltstrike.com/ (visited on 2025-05-22) (cit. on p. 11).

[27] Atomic Red Team. *Atomic Red Team: Adversary Emulation for Cybersecurity*. 2024. URL: https://www.atomicredteam.io/ (visited on 2025-05-22) (cit. on pp. 12, 30).

[28] K. Linux. *Kali Linux - Penetration Testing and Security Auditing*. 2024. URL: https://www.kali.org/ (visited on 2025-05-22) (cit. on p. 12).

[29] L. " Faletra and the Parrot Dev Team. *Parrot OS*. 2013. URL: https://parrotsec.org/ (visited on 2025-05-22) (cit. on p. 12).

[30] Kali Linux. *HPing3*. 2025. URL: https://www.kali.org/tools/hping3/ (visited on 2025-05-22) (cit. on p. 12).

[31] SecTools. *SuperScan*. 2025. URL: https://sectools.org/tool/superscan/ (visited on 2025-05-22) (cit. on p. 12).

[32] T. N. Project. *Ncat - A Feature-packed Networking Utility*. 2025. URL: https://nmap.org/ncat/ (visited on 2025-05-22) (cit. on p. 12).

[33] O. Foundation. *OWASP ZAP (Zed Attack Proxy)*. 2025. URL: https://www.zaproxy.org/ (visited on 2025-05-22) (cit. on p. 12).

[34] T. T. Group. *Tcpdump*. 2025. URL: https://www.tcpdump.org/ (visited on 2025-05-22) (cit. on p. 12).

[35] T. W. Foundation. *Wireshark*. 2025. URL: https://www.wireshark.org/ (visited on 2025-05-22) (cit. on p. 12).

[36] Netcraft. *Netcraft*. 2025. URL: https://www.netcraft.com/ (visited on 2025-05-22) (cit. on p. 12).

[37] T. L. D. Project. *whois - Whois client for querying domain information*. 2025. URL: https://linux.die.net/man/1/whois/ (visited on 2025-05-22) (cit. on p. 12).

[38] T. L. D. Project. *nslookup - Query Internet Name Servers*. 2025. URL: https://linux.die.net/man/1/nslookup/ (visited on 2025-05-22) (cit. on p. 12).

[39] G. Project. *GDB: The GNU Debugger*. 1986. URL: https://www.gnu.org/software/gdb/ (visited on 2025-05-22) (cit. on p. 12).

[40] G. Binutils. *objdump: A GNU Binary Utility*. 1991. URL: https://sourceware.org/binutils/ (visited on 2025-05-22) (cit. on p. 12).

[41] P. Kranenburg and Contributors. *strace: A Diagnostic, Debugging and Instructional Userspace Tracer*. 1991. URL: https://strace.io/ (visited on 2025-05-22) (cit. on p. 12).

[42] SecTools. *Brutus Password Cracker*. 2025. URL: https://sectools.org/tool/brutus/ (visited on 2025-05-22) (cit. on p. 12).

[43] O. Project. *John the Ripper*. 2025. URL: https://www.openwall.com/john/ (visited on 2025-05-22) (cit. on p. 12).

[44] Kali Linux. *Hydra*. 2025. URL: https://www.kali.org/tools/hydra/ (visited on 2025-05-22) (cit. on pp. 12, 60).

[45] O. Project. *OpenSSL: The Open Source toolkit for SSL/TLS*. 1998. URL: https://www.openssl.org/ (visited on 2025-05-22) (cit. on p. 12).

[46] GCHQ. *CyberChef: The Cyber Swiss Army Knife*. 2016. URL: https://gchq.github.io/CyberChef/ (visited on 2025-05-22) (cit. on p. 12).

[47] C. Developers. *Cryptii: Modular Conversion, Encoding and Encryption*. 2012. URL: https://cryptii.com/ (visited on 2025-05-22) (cit. on p. 12).

[48] Kali Linux. *Veil*. 2025. URL: https://www.kali.org/tools/veil/ (visited on 2025-05-22) (cit. on p. 12).

[49] T. E. Project. *Ettercap*. 2025. URL: https://www.ettercap-project.org/ (visited on 2025-05-22) (cit. on p. 12).

[50] Kali Linux. *Arjun*. 2025. URL: https://www.kali.org/tools/arjun/ (visited on 2025-05-22) (cit. on p. 12).

[51] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press. URL: http://www.deeplearningbook.org (cit. on pp. 13, 16).

[52] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006. ISBN: 9780387310732 (cit. on pp. 13–15).

[53] R. Collobert and J. Weston. "A unified architecture for natural language processing: deep neural networks with multitask learning". In: *Proceedings of the 25th International Conference on Machine Learning*. Association for Computing Machinery, 2008, pp. 160–167. DOI: 10.1145/1390156.1390177 (cit. on p. 17).

[54] D. Bahdanau, K. H. Cho, and Y. Bengio. "Neural machine translation by jointly learning to align and translate". In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (2015), pp. 1–15. URL: https://arxiv.org/abs/1409.0473 (cit. on p. 17).

[55] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Vol. 1. Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423 (cit. on p. 19).

[56] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. "CodeBERT: A pre-trained model for programming and natural languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139 (cit. on pp. 20, 21).

[57] Microsoft. *CodeGPT*. 2021. URL: https://github.com/microsoft/CodeGPT (visited on 2025-05-22) (cit. on p. 20).

[58] EleutherAI. *GPT-Neo*. 2021. URL: https://github.com/EleutherAI/gpt-neo (visited on 2025-05-22) (cit. on p. 20).

[59] OpenAI. *Codex*. 2021. URL: https://openai.com/research/codex (visited on 2025-05-22) (cit. on pp. 20, 22).

[60] H. Face. *CodeParrot: A Code Generation Model*. 2022. URL: https://huggingface.co/models (visited on 2025-05-22) (cit. on p. 20).

[61] Google. *PaLM-Coder*. 2022. URL: https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html (visited on 2025-05-22) (cit. on p. 20).

[62] BigScience. *BLOOM: BigScience Large Open-Science Open Access Multilingual Language Model*. 2022. URL: https://bigscience.huggingface.co/blog/bloom (visited on 2025-05-22) (cit. on p. 20).

[63] OpenAI. *GPT-4 Technical Report*. 2023. URL: https://openai.com/research/gpt-4 (visited on 2025-05-22) (cit. on pp. 20, 23).

[64] H. Face. *StarCoder*. 2023. URL: https://huggingface.co/StarCoder (visited on 2025-05-22) (cit. on p. 20).

[65] Microsoft. *WizardCoder*. 2023. URL: https://github.com/microsoft/WizardCoder (visited on 2025-05-22) (cit. on p. 20).

[66] Meta. *Code Llama: Open Foundation Models for Code*. 2023. URL: https://ai.meta.com/models/code-llama (visited on 2025-05-22) (cit. on p. 20).

[67] Anthropic. *Claude 3: Advancing Safe AI*. 2024. URL: https://www.anthropic.com/index/claude (visited on 2025-05-22) (cit. on p. 20).

[68] Meta. *Llama 3: Language Models for Open Research*. 2024. URL: https://ai.meta.com/llama3 (visited on 2025-05-22) (cit. on p. 20).

[69] H. Face. *StarCoder2-Instruct*. 2024. URL: https://huggingface.co/StarCoder2-Instruct (visited on 2025-05-22) (cit. on p. 20).

[70] M. AI. *Introducing Codestral*. 2025. URL: https://mistral.ai/news/codestral/ (visited on 2025-05-22) (cit. on pp. 20, 21).

[71] F. N. Motlagh, M. Hajizadeh, M. Majd, P. Najafi, F. Cheng, and C. Meinel. "Large language models in cybersecurity: State-of-the-art". In: *arXiv preprint arXiv:2402.00891* (2024). URL: https://arxiv.org/pdf/2402.00891 (cit. on p. 21).

[72] C. S. Eze and L. Shamir. "Analysis and Prevention of AI-Based Phishing Email Attacks". In: *Electronics (Switzerland)* 13.10 (2024). DOI: 10.3390/electronics13101839 (cit. on p. 21).

[73] H. Hanif and S. Maffeis. "VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection". In: *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2022, pp. 1–8. DOI: 10.1109/IJCNN55064.2022.9892280 (cit. on p. 21).

[74] M. Sladić, V. Valeros, C. Catania, and S. Garcia. "LLM in the Shell: Generative Honeypots". In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2024, pp. 430–435. DOI: 10.1109/EuroSPW61312.2024.00054 (cit. on p. 22).

[75] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. S. Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li. "Textbooks Are All You Need". In: *arXiv preprint arXiv:2306.11644* (2023). URL: https://arxiv.org/abs/2306.11644 (cit. on p. 22).

[76] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin. "AceCoder: An Effective Prompting Technique Specialized in Code Generation". In: *ACM Transactions on Software Engineering and Methodology* 33 (2024). DOI: 10.1145/3675395 (cit. on pp. 22–24).

[77] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati. "Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks". In: *arXiv preprint arXiv:2310.10508* (2023). URL: https://arxiv.org/abs/2310.10508 (cit. on p. 23).

[78] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. "Retrieval-augmented generation for knowledge-intensive NLP tasks". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf (cit. on p. 23).

[79] L. Alotaibi, S. Seher, and N. Mohammad. "Cyberattacks Using ChatGPT: Exploring Malicious Content Generation Through Prompt Engineering". In: *2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETSIS)*. IEEE, 2024, pp. 1304–1311. DOI: 10.1109/ICETSIS61505.2024.10459698 (cit. on p. 23).

[80] H. Yoo, Y. Yang, and H. Lee. "Code-Switching Red-Teaming: LLM Evaluation for Safety and Multilingual Understanding". In: *openreview.net preprint openreview.net:d1PtojR26j* (2024). URL: https://openreview.net/forum?id=d1PtojR26j (cit. on p. 23).

[81] GitHub. *GitHub Copilot: Your AI Pair Programmer*. 2025. URL: https://github.com/features/copilot (visited on 2025-05-22) (cit. on p. 23).

[82] A. W. Services. *AWS Developer Portal*. 2025. URL: https://aws.amazon.com/pt/q/developer/ (visited on 2025-05-22) (cit. on p. 23).

[83] M. F. Mohamed Firdhous, W. Elbreiki, I. Abdullahi, B. H. Sudantha, and R. Budiarto. "WormGPT: A Large Language Model Chatbot for Criminals". In: *2023 24th International Arab Conference on Information Technology (ACIT)*. IEEE, 2023, pp. 1–6. DOI: 10.1109/ACIT58888.2023.10453752 (cit. on p. 23).

[84] P. V. Falade. "Decoding the Threat Landscape : ChatGPT, FraudGPT, and WormGPT in Social Engineering Attacks". In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* (2023), pp. 185–198. DOI: 10.32628/cseit2390533 (cit. on p. 23).

[85] F. Rustam, P. Ranaweera, and A. D. Jurcut. "AI on the Defensive and Offensive: Securing Multi-Environment Networks from AI Agents". In: *ICC 2024 - IEEE International Conference on Communications*. IEEE, 2024, pp. 4287–4292. DOI: 10.1109/ICC51166.2024.10622943 (cit. on p. 23).

[86] Google. *Google Gemini*. 2025. URL: https://gemini.google.com/ (visited on 2025-05-22) (cit. on p. 23).

[87] Y. Yigit, W. J. Buchanan, M. G. Tehrani, and L. Maglaras. "Review of Generative AI Methods in Cybersecurity". In: *arXiv preprint arXiv:2403.08701* (2024). URL: https://arxiv.org/abs/2403.08701 (cit. on pp. 23, 24).

[88] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass. "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing". In: *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024, pp. 847–864. ISBN: 9781939133441. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/deng (cit. on p. 24).

[89] P. Liguori, C. Improta, R. Natella, B. Cukic, and D. Cotroneo. "Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators". In: *Expert Systems with Applications* 225 (2023). DOI: 10.1016/j.eswa.2023.120073 (cit. on pp. 24, 25, 47).

[90] C. Guo, X. Liu, C. Xie, A. Zhou, Y. Zeng, Z. Lin, D. Song, and B. Li. "RedCode: Risky Code Execution and Generation Benchmark for Code Agents". In: *Advances in Neural Information Processing Systems*. Vol. 37. Curran Associates, Inc., 2024, pp. 106190–106236. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/bfd082c452dffb450d5a5202b0419205-Paper-Datasets_and_Benchmarks_Track.pdf (cit. on pp. 24, 25).

[91] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim. "A Survey on Large Language Models for Code Generation". In: *arXiv preprint arXiv:2403.08701* (2024). URL: https://arxiv.org/abs/2406.00515 (cit. on p. 25).

[92] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh. *Can we generate shellcodes via natural language? An empirical study*. Vol. 29. Springer US, 2022. ISBN: 0123456789. DOI: 10.1007/s10515-022-00331-3 (cit. on p. 26).

[93] G. Yang, X. Chen, Y. Zhou, and C. Yu. "DualSC: Automatic Generation and Summarization of Shellcode via Transformer and Dual Learning". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 361–372. DOI: 10.1109/SANER53432.2022.00052 (cit. on p. 26).

[94] A. Chowdhary, K. Jha, and M. Zhao. "Generative Adversarial Network (GAN)-Based Autonomous Penetration Testing for Web Applications". In: *Sensors* 23.18 (2023), pp. 1–18. ISSN: 14248220. DOI: 10.3390/s23188014 (cit. on pp. 26, 28).

[95] R. Natella, P. Liguori, C. Improta, B. Cukic, and D. Cotroneo. "AI Code Generators for Security: Friend or Foe?" In: *IEEE Security and Privacy* 22.5 (2024), pp. 73–81. ISSN: 15584046. DOI: 10.1109/MSEC.2024.3355713 (cit. on pp. 26, 29).

[96] Hacktricks. *Hacktricks*. 2025. URL: https://book.hacktricks.xyz/ (visited on 2025-05-22) (cit. on p. 30).

[97] R. T. Recipe. *PowerShell tips & tricks*. 2025. URL: https://redteamrecipe.com/powershell-tips-tricks/ (visited on 2025-05-22) (cit. on p. 30).

[98] I. Matter. *PowerShell commands for pentesters*. 2025. URL: https://www.infosecmatter.com/powershell-commands-for-pentesters/ (visited on 2025-05-22) (cit. on p. 30).

[99] MITRE. *CALDERA plugin: Stockpile*. 2025. URL: https://github.com/mitre/stockpile (visited on 2025-05-22) (cit. on p. 30).

[100] E. Project. *Empire*. 2025. URL: https://github.com/EmpireProject/Empire (visited on 2025-05-22) (cit. on p. 30).

[101] N. Mittal. *Nishang - Offensive PowerShell for Red Teams*. 2018. URL: https://github.com/samratashok/nishang (visited on 2025-05-22) (cit. on pp. 31, 33, 34, 59).

[102] NetSPI. *PowerUpSQL - A PowerShell Toolkit for SQL Server*. 2025. URL: https://github.com/NetSPI/PowerUpSQL/wiki/PowerUpSQL-Cheat-Sheet (visited on 2025-05-22) (cit. on pp. 31, 33, 34).

[103] NetSPI. *MicroBurst*. 2025. URL: https://github.com/NetSPI/MicroBurst (visited on 2025-05-22) (cit. on pp. 31, 33, 34, 58).

[104] Whitecat18. *PowerShell Scripts for Hackers and Pentesters*. 2023. URL: https://github.com/Whitecat18/Powershell-Scripts-for-Hackers-and-Pentesters (visited on 2025-05-22) (cit. on p. 33).

[105] S1ckB0y1337. *Active Directory Exploitation Cheat Sheet*. 2025. URL: https://github.com/S1ckB0y1337/Active-Directory-Exploitation-Cheat-Sheet (visited on 2025-05-22) (cit. on p. 33).

[106] Medium. *Medium - Where good ideas find you*. 2025. URL: https://medium.com/ (visited on 2025-05-22) (cit. on p. 33).

[107] TryHackMe. *TryHackMe - Learn Cybersecurity, Penetration Testing, and Ethical Hacking*. 2025. URL: https://tryhackme.com/ (visited on 2025-05-22) (cit. on p. 33).

[108] Microsoft. *Az PowerShell Module*. 2025. URL: https://learn.microsoft.com/en-us/powershell/azure/az-ps/?view=az-ps (visited on 2025-05-22) (cit. on p. 33).

[109] Microsoft. *AzureAD PowerShell Module*. 2025. URL: https://learn.microsoft.com/en-us/powershell/azure/azuread/?view=azuread-ps (visited on 2025-05-22) (cit. on p. 33).

[110] Microsoft. *MSOnline PowerShell Module*. 2025. URL: https://learn.microsoft.com/en-us/powershell/module/msonline/?view=azure-ps (visited on 2025-05-22) (cit. on p. 33).

[111] PowerShellMafia. *PowerView.ps1*. 2015. URL: https://github.com/PowerShellMafia/PowerSploit/blob/master/Recon/PowerView.ps1 (visited on 2025-05-22) (cit. on p. 33).

[112] Microsoft. *Active Directory Module for PowerShell*. 2025. URL: https://learn.microsoft.com/en-us/powershell/module/activedirectory/?view=windowsserver2025-ps (visited on 2025-05-22) (cit. on p. 33).

[113] darkoperator. *Posh-SecMod*. 2025. URL: https://github.com/darkoperator/Posh-SecMod/tree/master (visited on 2025-05-22) (cit. on p. 33).

[114] PowerShellMafia. *PowerSploit*. 2024. URL: https://github.com/PowerShellMafia/PowerSploit (visited on 2025-05-22) (cit. on p. 33).

[115] StationX. *StationX - Cyber Security Courses and Training*. 2025. URL: https://www.stationx.net/ (visited on 2025-05-22) (cit. on p. 33).

[116] Kevin-Robertson. *Powermad*. 2025. URL: https://github.com/Kevin-Robertson/Powermad (visited on 2025-05-22) (cit. on p. 33).

[117] PowerShellMafia. *PowerUp.ps1*. 2015. URL: https://github.com/PowerShellMafia/PowerSploit/blob/master/Privesc/PowerUp.ps1 (visited on 2025-05-22) (cit. on p. 33).

[118] N. Mittal. *RACE: A PowerShell module for executing ACL attacks against Windows targets*. 2019. URL: https://github.com/samratashok/RACE (visited on 2025-05-22) (cit. on p. 33).

[119] SecAbstraction. *PowerCat*. 2024. URL: https://github.com/secabstraction/PowerCat (visited on 2025-05-22) (cit. on p. 33).

[120] H. Face. *Hugging Face*. 2025. URL: https://huggingface.co/ (visited on 2025-05-22) (cit. on p. 37).

[121] Q. Team. *Qwen2.5: A Party of Foundation Models*. 2024. URL: https://qwenlm.github.io/blog/qwen2.5/ (visited on 2025-05-22) (cit. on p. 37).

[122] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang, X. Ren, X. Ren, J. Zhou, and J. Lin. *Qwen2.5 Coder Technical Report*. 2024. URL: https://arxiv.org/abs/2409.12186 (visited on 2025-05-22) (cit. on p. 37).

[123] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, and A. Korenev. *The Llama 3 Herd of Models*. 2024. URL: https://arxiv.org/abs/2407.21783 (visited on 2025-05-22) (cit. on p. 37).

[124] Unsloth AI. *Unsloth: Open Source Fine-Tuning for LLMs*. 2025. URL: https://unsloth.ai/ (visited on 2025-05-22) (cit. on p. 38).

[125] Unsloth Team. *LoRA Hyperparameters Guide — Unsloth Docs*. 2024. URL: https://docs.unsloth.ai/get-started/fine-tuning-guide/lora-hyperparameters-guide (visited on 2025-05-22) (cit. on p. 38).

[126] Microsoft Corporation. *PSScriptAnalyzer*. 2025. URL: https://github.com/PowerShell/PSScriptAnalyzer (visited on 2025-05-22) (cit. on p. 43).

[127] S. Mora. *ROUGE: A pure Python implementation of the ROUGE metric*. 2019. URL: https://pypi.org/project/rouge/ (visited on 2025-05-22) (cit. on p. 47).

[128] Hugging Face. *evaluate: A Python library for model evaluation and comparison*. 2025. URL: https://pypi.org/project/evaluate/ (visited on 2025-05-22) (cit. on p. 47).

[129] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. L. and Lidong Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *CoRR* abs/2102.04664 (2021). URL: https://www.microsoft.com/en-us/research/publication/codexglue-a-machine-learning-benchmark-dataset-for-code-understanding-and-generation/ (cit. on p. 47).

[130] kuangzh. *pylcs: A super fast C++ implementation of classic LCS problems using dynamic programming*. 2023. URL: https://pypi.org/project/pylcs/ (visited on 2025-05-22) (cit. on p. 47).

[131] DeepSeek-AI. *DeepSeek Chat Platform*. 2025. URL: https://chat.deepseek.com/ (visited on 2025-05-22) (cit. on p. 51).

[132] Proxmox Server Solutions GmbH. *Proxmox Virtual Environment: Overview*. 2025. URL: https://www.proxmox.com/en/products/proxmox-virtual-environment/overview (visited on 2025-05-22) (cit. on p. 55).

NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY