LLM-driven PowerShell Analysis Assistant

Raj Singh

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science
of
University College London

Department of Computer Science
University College London

2025

Declaration

I confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

……

Raj Singh

Abstract

This thesis presents an original, data-centric methodology for fine-tuning Large Language Models (LLMs) to perform structured analysis of obfuscated PowerShell commands. Through engineering a specialized data factory, this proposed system transforms the manual and error-prone task of script deobfuscation into one more reliable and prone to automation. In this way, the center of this research is the "PowerShell-Sentinel Data Factory," a software pipeline built to generate high-quality, specialized training data. The system aims to deliver accurate, structured insights into obfuscated scripts, reducing security analyst overhead and improving the consistency of threat detection.

Traditional security operations workflows struggle with the analysis of obfuscated PowerShell, a technique frequently used by attackers when evading detection. Such analysis can be manual and labor-intensive, requiring expertise and leading to inconsistent results under tight deadlines and high stakes. Thus, calling upon this work's central hypothesis, this research posits that a distinctly specialized training dataset that is both curated with domain expertise and generated from a controlled lab, will facilitate a language model to accurately decompose and analyze complex, obfuscated commands. This refined focus on generating high-fidelity, specific data, rather than relying on general code corpora, is intended to produce a model capable of reliable, structured analysis in real-world security scenarios.

The thesis consists of the following three investigations:

1. **Lab Architecture and Data Curation.** In this investigation, a controlled lab environment is engineered using Windows Server VMs and a local Splunk instance to capture high-fidelity PowerShell execution telemetry (Event IDs 4103 and 4104). An initial "starter pack" of PowerShell primitives is correlated with its ground-truth telemetry logs through a human-in-the-loop process facilitated by a dedicated command-line tool. This experiment establishes the foundational data schema and the validated, high-quality "golden signal" dataset required for all subsequent steps.

2. **The PowerShell-Sentinel Data Factory.** This research documents an iterative engineering process for the data factory. It begins by implementing a baseline automated pipeline using a randomized, string-based obfuscation engine. Quantitative analysis of this baseline revealed critical limitations, including a low generation success rate and a profound dataset bias. Motivated by these findings, a second, superior factory was engineered, built upon a surgically reconstructed, deterministic, AST-aware obfuscation engine. This final factory successfully

mitigated the dataset biases and dramatically improved the success rate, forming the foundation of the final model.

3. **Model Fine-Tuning and Evaluation.** The final investigation fine-tunes a Llama 3 model using QLoRA: a baseline model on the original biased dataset and a second-pass model on the superior, diverse dataset. A rigorous comparative analysis is then undertaken. The models' ability to perform structured analysis is quantified against locked test sets using metrics including a strict "JSON Parse Success Rate", Deobfuscation Accuracy, and Macro F1-Scores for intent classification, and telemetry prediction. This experiment empirically validates that the data-centric improvements of the "Second Pass" produce a categorically superior and more reliable model.

The thesis presents the following contributions to science:

1. **A Validated Methodology for Generating Specialized Cybersecurity Data.** By creating the PowerShell-Sentinel Data Factory, this research provides a reusable and automated software pipeline for generating high-fidelity, specialized training data. This methodology addresses a critical bottleneck in applying machine learning to niche cybersecurity problems, offering a blueprint for creating robust datasets where none exist.

2. **A Novel Approach for Reliable, Structured PowerShell Analysis.** The system embeds the complex logic of PowerShell deobfuscation directly into a fine-tuned LLM, enabling consistent and accurate structured analysis of obfuscated commands. By automating this task, it minimizes the risk of human error and provides a scalable tool for security analysts, ensuring that deep script analysis can be applied uniformly and rapidly across an organization.

3. **A Framework for Robust Quality Assurance in the Iterative Design and Validation of a Synthetic Data Generation Pipeline.** The project advocates a multi-stage QA process that moves beyond simple validation to an iterative diagnostic framework. This includes: (1) pre-emptive "Canary Cage" validation of the obfuscation engine's functional correctness; (2) live "Execution Validation" of every command; and (3) a post-facto quantitative analysis of the entire generation process via a detailed audit log. The final stage was critical for diagnosing the "convergence effect" and dataset bias in the baseline factory, and for identifying the "command line too long" limitation as the primary failure mode in the re-engineered factory, thereby providing the evidence needed to justify the architectural improvements.

Raj Singh, LLM-driven PowerShell Analysis Assistant
Supervisor: Prof. Philip Treleaven

Impact Statement

This research presents four primary contributions with direct and significant impact on industry cybersecurity practices, particularly within Security Operations Centers (SOCs), threat intelligence teams, and AI/ML development units:

1. **A Validated Methodology for Generating Specialized Cybersecurity Data.** This thesis provides a reusable, end-to-end blueprint for an automated data factory. It demonstrates how a combination of a high-fidelity lab environment, a human-in-the-loop curation interface with interactive parsing, and a multi-stage QA framework can successfully address the critical bottleneck of data scarcity in niche security domains.

2. **A Novel Approach for Reliable, Structured PowerShell Analysis.** The project proves that a data-centric fine-tuning approach can transform a general-purpose LLM into a specialized and reliable tool. By implementing a rigorous methodology involving explicit instructional prompts and a flattened data schema, the final model achieved a high JSON Parse Success Rate of 85.79% on a complex, diverse test set. Crucially, it achieved a 75.40% Macro F1-Score on the difficult, forensically-critical task of structured telemetry prediction. This provides strong empirical evidence that this methodology can produce a trustworthy and practically useful assistant for automated security processes.

3. **A Framework for Robust Quality Assurance in Security Data Generation.** This research provides a practical, multi-stage QA framework to de-risk the use of synthetic data, moving beyond simple validation to provide a comprehensive model for understanding and controlling the data generation process. It combines a low-cost unit testing methodology to functionally validate obfuscation logic before expensive, large-scale runs (“Canary Cage”), a stringent quality gate that guarantees every training sample is functionally viable in a real environment (“Execution Validation”) and a methodology for instrumenting the generation pipeline to produce a detailed audit log. This log enables teams to quantitatively diagnose emergent biases and identify environmental bottlenecks.

4. **A Methodology for Quantifying LLM Robustness in Adversarial Contexts.** The post-tuning quantization analysis, which identified the “quantization cliff” for the fine-tuned model, provides a critical and reusable methodology for industry practitioners. It demonstrates how to empirically determine the optimal balance between model accessibility (file size) and performance reliability. This process allows organizations to make evidence-based decisions when packaging LLMs for deployment on standard analyst hardware, ensuring the chosen

model is not only effective but also safe from the catastrophic performance collapse that can occur at lower quantization levels.

Acknowledgements

I would like to express my gratitude to Prof. Philip Treleaven for enabling this research through counsel and accumulated intuition.

I also wish to express my gratitude to UCL for offering high-performance computing for this research.

Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

This chapter outlines the challenges associated with analyzing obfuscated PowerShell commands, a common technique used to evade security monitoring. It introduces the primary aim of the research: to engineer and validate a data-centric methodology for fine-tuning a Large Language Model (LLM) for this purpose. The core contribution, the "PowerShell-Sentinel Data Factory," is presented as a software pipeline for generating the specialized training data required. An overview of the research objectives, experiments and thesis structure is also provided.

1.1. Research Motivation

In modern cybersecurity, the analysis of obfuscated PowerShell scripts represents a significant and growing challenge. As a powerful, native administrative tool, PowerShell is a primary target for "Living Off the Land" (LOTL) attacks, where adversaries use legitimate system tools to conceal their activities. The manual deobfuscation of these scripts is an artisanal, time-intensive process that requires deep domain expertise, does not scale under pressure, and is prone to inconsistency. While general-purpose LLMs have shown promise in code analysis, their application to the highly specialized and adversarial domain of cybersecurity often yields unreliable and non-deterministic results, a finding supported by recent literature (Ullah et al., 2024).

The central bottleneck in developing a more reliable, automated solution is the profound scarcity of high-quality, specialized training data. Public code repositories are noisy, and existing security datasets often lack the specific, structured format required to teach a model the nuanced task of deobfuscation and structured analysis. This research is motivated by the hypothesis that a data-centric approach can overcome these limitations. It posits that by engineering a dedicated pipeline —a "data factory"—to generate a large-scale, high-fidelity dataset of obfuscated command-to-analysis pairs, it is possible to fine-tune a general-purpose LLM into a specialized and trustworthy security assistant.

1.2. Research Objectives

The primary objective of this research is to engineer and validate the "PowerShell-Sentinel Data Factory" and use its output to produce a reliable, LLM-driven PowerShell analysis tool. This

overarching goal is composed of several specific objectives. First, the project aims to engineer a controlled, isolated lab environment capable of executing PowerShell commands and capturing the resulting high-fidelity telemetry. Second, it seeks to develop a robust, automated software pipeline, the Data Factory, that systematically obfuscates a curated set of PowerShell “primitives” to generate a large-scale, functionally validated training dataset. The third objective is to fine-tune a state-of-the-art, open-source LLM on this specialized dataset using Parameter-Efficient Fine-Tuning (PEFT) techniques. Following this, the fourth objective is to rigorously evaluate the performance of the fine-tuned model against a locked test set, using quantitative metrics for both classification accuracy, such as F1-score, and output reliability, such as the JSON Parse Success Rate. The final objective is to package the resulting model into a practical, user-facing Command-Line Interface (CLI) tool for security analysts.

1.3. Research Experiments

The research is structured as a sequence of three core experiments, each building upon the validated output of the last. This phased approach ensures that each component of the final system is founded on a methodologically sound and empirically validated base. Each experiment is designed to answer a specific research question, with its outputs forming the direct inputs for the subsequent phase, creating a logical and cohesive research narrative from foundational data curation to the final, practical deliverable.

1.3.1. Experiment 1: Lab Architecture and Data Curation Tooling

This initial experiment is concerned with establishing foundational data quality. It addresses the challenge of generating high-fidelity, causal telemetry by engineering a controlled, three-VM Active Directory lab environment, which was subsequently hardened with System Monitor (*Sysmon*) to broaden telemetry capture. The experiment details the development of a comprehensive suite of human-in-the-loop tooling, orchestrated by an interactive command-line interface. A key methodological innovation from this phase is the “Framework for Interactive Telemetry Parsing,” a novel solution to the problem of reliably mapping unstructured log events to a structured format. This experiment culminates in the creation of an initial, expert-validated, ground-truth dataset of command-to-telemetry mappings, namely a trusted seed library that was later enriched with new Sysmon-aware primitives, which serves as the foundation for the entire project.

1.3.2. Experiment 2: The PowerShell-Sentinel Data Factory

This experiment addresses the challenge of scaling data generation from the initial curated seed. It describes an iterative research process, beginning with the design of an initial software pipeline. This baseline methodology featured a “Payload and Wrappers” layered obfuscation engine and a resilient master controller. However, quantitative analysis of this first-pass factory’s performance

revealed critical limitations: a low generation success rate which resulted in a dataset with a profound bias towards a single obfuscation technique.

This finding motivated a second, more advanced stage of the experiment. The initial engine was replaced with a re-engineered factory, architected around a robust, deterministic obfuscation engine. This enhanced methodology, created by surgically reconstructing the core logic of state-of-the-art tooling, successfully solved the bias and reliability issues of the baseline approach. The chapter concludes with a comparative analysis of the two pipelines, quantifying the dramatic improvements in success rate and dataset quality achieved by this methodological evolution.

1.3.3. Experiment 3: Model Fine-Tuning, Evaluation and Delivery

This experiment details the MLOps pipeline that consumes the validated, de-duplicated output of the Data Factory. It presents the methodology for fine-tuning a state-of-the-art *meta-llama/Meta-Llama-3-8B-Instruct* model using Parameter-Efficient Fine-Tuning (QLoRA). The core of this experiment is the rigorous quantitative evaluation of the final model against a locked test set, using metrics such as the Macro F1-Score and a strict, Pydantic-based "JSON Parse Success Rate." The experiment concludes with a qualitative, post-tuning analysis of model quantization to identify the optimal format for the final, CPU-runnable deliverable: a standalone, user-facing CLI tool.

1.4. Scientific Contributions

This thesis presents three primary contributions to the field of applied machine learning in cybersecurity, moving beyond a simple implementation to offer reusable methodologies and frameworks.

1. **A Validated Methodology for Generating Specialized Cybersecurity Data.** The core contribution is a reusable blueprint for an automated data factory that addresses the critical bottleneck of data scarcity in niche security domains. The methodology is original in its synthesis of several key components: a high-fidelity lab environment for telemetry generation; a Pydantic-based "data contract" system for robustness; a human-in-the-loop framework for interactive, deterministic log parsing; and a multi-stage QA process that includes live execution validation. This provides a comprehensive, end-to-end solution for creating reliable datasets where none exist.

2. **An Original Approach for Reliable, Structured PowerShell Analysis.** This research demonstrates the viability of transforming a general-purpose LLM into a specialized tool capable of consistent and accurate structured analysis. The key innovation is the data-centric fine-tuning approach, which combined an explicit instructional prompt with a robust, flattened data schema. This proved highly effective in teaching the model to adhere to a strict JSON

output, achieving an 85.79% JSON Parse Success Rate on a diverse, locked test set. Critically, this methodology produced a model capable of high-fidelity telemetry prediction, achieving a 75.40% F1-Score on this complex, structured generation task. This directly addresses the critical issue of non-determinism in general-purpose models, providing a methodology for producing a trustworthy security assistant.

3. **A Framework for Robust Quality Assurance in the Synthetic Generation of Security Data.** The project advocates a multi-stage QA framework that provides a structured process for ensuring data integrity and understanding the emergent behaviors of the generation pipeline. This framework moves beyond simple validation to an iterative diagnostic process. It includes: (1) pre-emptive "Canary Cage" validation of the obfuscation engine's functional correctness; (2) live "Execution Validation" of every command in the target environment; (3) a post-facto quantitative analysis of the entire generation run via a detailed audit log, which was critical for diagnosing the "convergence effect" and dataset bias in the baseline factory; and (4) the discovery and characterization of environmental bottlenecks (such as command-line length limits) as a primary failure mode, informing the design of a more resilient architecture.

4. **An Original, Deterministic Framework for Interactive, Human-In-The-Loop Telemetry Parsing.** This methodology, implemented in the *primitives_manager.py* CLI, replaces brittle, static parsers or unreliable LLM-based parsers with a system where an analyst teaches the software how to interpret new log formats. This creates a persistent, ever-growing library of deterministic parsing rules, providing a practical and robust solution to the challenge of processing diverse and evolving log data.

5. **An Empirical Analysis of the Impact of Post-Tuning Quantization on A Specialized, Fine-Tuned LLM.** The qualitative evaluation of the various GGUF formats, which identified the definitive "quantization cliff" at the Q2_K level, offers a practical and repeatable methodology for selecting a deployable model. It provides strong evidence that significant model compression (down to the Q4_K_M level) is achievable without a discernible loss of reasoning capability, a critical finding for the practical deployment of LLMs on standard, resource-constrained hardware.

1.5. Thesis Structure

The structure of this thesis is organised as follows:

- **Chapter 2 – Background and Literature Review.** The relevant literature and key concepts underpinning this research are reviewed to introduce the problem domain. The chapter begins by establishing the current PowerShell security landscape, detailing its use as a "Living Off the Land" attack vector and the significant challenges posed by adversarial obfuscation techniques.

It then provides a methodological review of automation in cybersecurity, surveying the application of machine learning and, specifically, Large Language Models for code analysis and anomaly detection. This review serves to place the research in a well-defined context and identify the critical gap—the need for specialized, reliable models—that this thesis aims to address.

- **Chapter 3 – Lab Architecture and Data Curation Tooling.** This chapter details the first of three core experiments, which focuses on establishing foundational data quality. It describes the complete engineering process for a controlled, multi-VM lab environment designed to generate high-fidelity, causal telemetry from PowerShell command execution. The chapter then outlines the development of a comprehensive suite of human-in-the-loop data curation tools, managed by an interactive CLI. This experiment culminates in the creation of an initial, expert-validated "golden signal" dataset of command-to-telemetry mappings, which serves as the trusted seed for all subsequent automated generation and model training.

- **Chapter 4 – The PowerShell-Sentinel Data Factory.** This chapter presents the second core experiment, detailing the iterative engineering process of the data factory. It begins by describing the baseline architecture and presenting a quantitative evaluation that reveals its critical limitations, namely a low success rate and profound dataset bias. It then documents the architectural rework that produced a superior, re-engineered factory built on a deterministic, reconstructed obfuscation engine. A comparative analysis provides definitive proof of the enhanced methodology’s success. Finally, the chapter analyzes a newly discovered scalability bottleneck, leading to the proposed design of a concurrent pipeline that is proposed as future work.

- **Chapter 5 – Model Fine-Tuning, Evaluation and Delivery.** The third and final experiment is detailed in this chapter, covering the complete MLOps pipeline that transforms the generated data into a practical tool. It outlines the methodology for data partitioning, prompt engineering, and the fine-tuning of a state-of-the-art *meta-llama/Meta-Llama-3-8B-Instruct* model using the Parameter-Efficient Fine-Tuning (PEFT) technique QLoRA. The chapter then presents a rigorous quantitative evaluation of the final model's performance against a locked test set, using strict metrics for both classification accuracy and output reliability. It concludes with a qualitative analysis of post-training quantization and a description of the final packaged CLI deliverable.

- **Chapter 6 – Conclusion and Future Work.** The final chapter provides an overall conclusion for the research, with a summary of the key findings and scientific contributions from each of the three experiments. It reflects on the success of the data-centric methodology in producing

a reliable, specialized analysis tool. The thesis ends with a discussion of the project's limitations and provides specific, data-driven recommendations for future work to be done in this area.

Chapter 2

2. Background and Literature Review

This chapter examines the current landscape of PowerShell security and cybersecurity automation. It reviews traditional detection methods and their limitations against obfuscation, discussing the critical role of specific telemetry like PowerShell Script Block Logging (Event ID 4104). The chapter summarizes relevant literature on the use of LLMs for code analysis and anomaly detection, establishing the rationale for developing a specialized, fine-tuned model and identifying gaps in existing research.

2.1. The PowerShell Security Landscape

This section examines the dual-use nature of PowerShell as both a powerful administrative tool and a preferred vector for modern adversaries. It reviews common offensive techniques, including "Living Off the Land" (LOTL) strategies, and details the obfuscation methods attackers use to conceal their activities. The section concludes by analyzing the limitations of traditional, manual analysis workflows in keeping pace with the volume and complexity of these threats in a modern Security Operations Center (SOC).

2.1.1. Traditional Manual Approaches and Their Limitations

In a typical Security Operations Center (SOC), the detection of potentially malicious PowerShell often begins with an alert from a Security Information and Event Management (SIEM) system, triggered by rules looking for suspicious command-line patterns or behaviors. When an obfuscated script is identified, the task of analysis falls to a human analyst. This manual process is an artisanal, time-intensive effort. The analyst must painstakingly deconstruct multiple layers of obfuscation,

decode payloads, and trace the logic of the script to determine its ultimate intent. This requires a deep level of expertise not only in PowerShell but also in the specific obfuscation techniques employed. The primary limitation of this approach is its lack of scalability. A single, complex command can take a skilled analyst a significant amount of time to unravel, a delay that is untenable during a live security incident. Furthermore, the process is prone to human error and inconsistency, particularly under pressure.

2.1.2. Challenges in Obfuscation Detection

Adversaries employ a wide array of obfuscation techniques to defeat both automated and manual analysis. These range from simple methods like string concatenation and the use of the format operator, to more complex techniques like Base64 encoding of entire script blocks and assigning command fragments to variables that are assembled and executed at runtime. The goal of these methods is to break the static signatures and patterns that security tools rely on. For example, a rule looking for the string “Invoke-Expression” will fail if the command is constructed as *&(‘Inv’ + ‘oke-Expression’)*.

The primary challenge in detecting malicious PowerShell is the combinatorial explosion of these possible obfuscation techniques, a concept illustrated in Figure 2.1. Adversaries do not rely on a single method; instead, they apply multiple, randomized layers of syntactic manipulation to a single, core malicious command. A simple intent, such as executing a payload with Invoke-Expression, can be transformed into a near-infinite number of different command strings. It can be hidden via string concatenation, encoded in Base64, built at runtime using the format operator, or obscured with backtick escape characters, among many other methods. This one-to-many relationship between a single malicious intent and its vast number of possible textual representations renders traditional, static signature-based detection methods, which look for a fixed string like “Invoke-Expression”, fundamentally unreliable. To effectively counter this, a detection methodology must be capable of deconstructing these layers and reasoning about the underlying functional intent, rather than simply matching superficial string patterns.

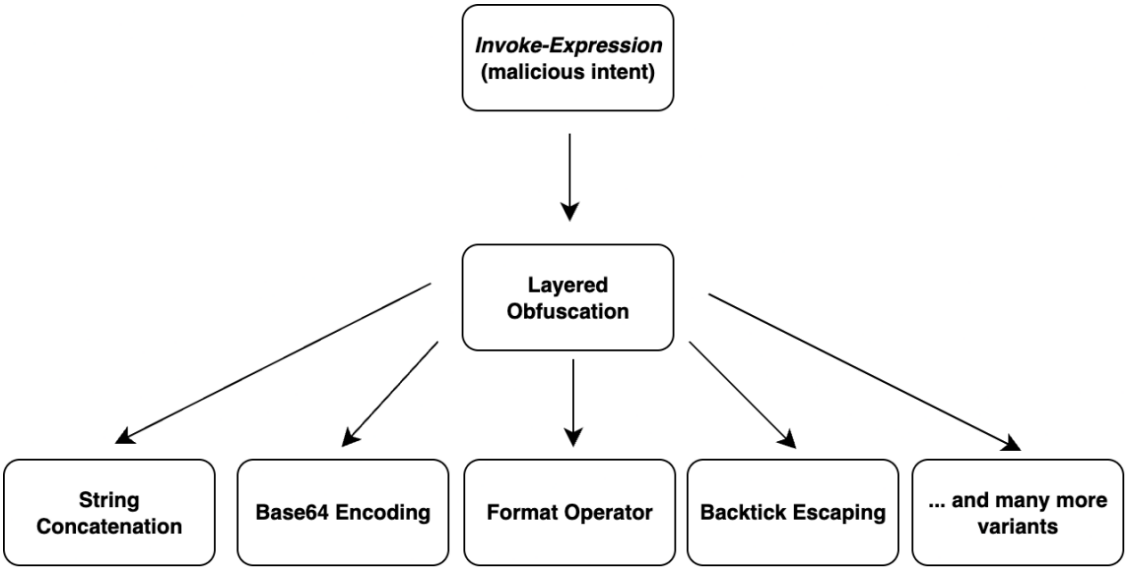

Figure 2.1: The Combinatorial Explosion of PowerShell Obfuscation

A single malicious intent (Invoke-Expression) can be represented by a vast and unpredictable number of syntactically different commands, rendering static, signature-based detection ineffective. The limitations exposed by this combinatorial challenge, alongside the scalability issues of manual analysis, necessitate a new approach. Table 2.1 provides a comparative analysis of traditional methods against the LLM-driven methodology proposed in this research, highlighting the key areas of improvement.

Table 2.1: Comparison of PowerShell Analysis Approaches

Evaluation Criterion	Manual Analysis (Human Analyst)	Static Signatures (SIEM/IDS)	LLM-driven (PowerShell-Sentinel)
Scalability	Low (Per-analyst)	High (Automated)	High (Automated)
Consistency	Low (Analyst-dependent)	High (Deterministic)	High (Deterministic Output)
Speed	Very Slow (Hours)	Very Fast (Milliseconds)	Fast (Seconds)
Adaptability to New Obfuscation	High (Human Reasoning)	Very Low (Brittle)	High (Pattern Generalization)
Required Human Expertise	High (Senior Specialist)	Medium (Rule Engineer)	Low (Junior Analyst)

As the table illustrates, the LLM-driven approach aims to combine the high adaptability of a human expert with the speed and scalability of an automated system, while lowering the required expertise for the end-user. This forces a reliance on behavioral detection and deep script analysis, reinforcing the bottleneck described in the manual analysis workflow and motivating the data-centric solution engineered in this thesis.

2.2. Generative AI and Large Language Models

The methodology central to this thesis is built upon recent advancements in a subfield of artificial intelligence known as Generative AI. Unlike traditional discriminative models, which are designed to classify or predict outcomes from a given set of inputs (e.g., identifying an email as spam or not spam), generative models are engineered to create new, synthetic data that resembles the data on which they were trained. This capability extends across various modalities, including images, audio, and, most relevantly to this research, natural language and computer code.

At the forefront of this field are Large Language Models (LLMs), a specialized class of generative models distinguished by their immense scale and sophisticated architecture. These models are typically neural networks containing billions of parameters, pre-trained on vast and diverse corpora of text and code sourced from the public internet. This extensive training endows them with a powerful, generalized understanding of syntax, semantics, grammar, and logical reasoning across numerous domains. Their ability to comprehend and generate human-like text makes them highly adaptable for a wide range of downstream tasks, from simple text summarization to complex code analysis.

The enabling technology behind the success of modern LLMs is the transformer architecture. First introduced in 2017, the transformer model departed from previous sequential architectures like Recurrent Neural Networks (RNNs) by introducing a novel mechanism called self-attention. This mechanism allows the model to weigh the influence of different words within an input sequence dynamically, enabling it to capture complex, long-range dependencies and contextual relationships far more effectively than its predecessors. It is this architectural innovation that provides LLMs with their deep contextual understanding, a capability that is fundamental to the analytical tasks explored in this thesis.

2.3. Automation in Cybersecurity

This section provides an overview of the increasing role of automation and machine learning in cybersecurity defense. It discusses the application of these technologies in areas such as anomaly detection and security testing. The section establishes the context for this research by identifying the use of Large Language Models (LLMs) for code analysis as a promising but still-developing field, highlighting the specific gap in existing research for specialized, high-fidelity models for script deobfuscation.

2.3.1. The Role of Machine Learning and LLMs in Security Testing

Machine learning has extended these capabilities into the realm of detection and prediction. Models are now commonly used for tasks like network traffic anomaly detection, user and entity

behavior analytics (UEBA), and malware classification. The recent advent of Large Language Models (LLMs) has opened a new frontier for code analysis and comprehension. Their ability to understand context, syntax, and semantics in programming languages makes them a promising candidate for security tasks that require deep code understanding. While general-purpose LLMs have shown some capability in explaining or summarizing code, their application to the highly specialized and adversarial domain of obfuscated script analysis is still a nascent area of research, with significant challenges in reliability and consistency.

2.3.2. Review of Relevant Publications

A review of the relevant literature reveals a clear trajectory from traditional machine learning approaches toward the sophisticated, data-centric methodologies that this thesis seeks to advance. The foundational work in this specific domain was established by researchers like *Hendler et al. (2018)*, who were among the first to apply deep learning to the problem of detecting malicious PowerShell. Their work demonstrated that a character-level Convolutional Neural Network (CNN) could outperform traditional NLP techniques (n-grams) by learning to identify subtle obfuscation patterns, such as alternating case or the use of specific characters, directly from the raw command string. This established the core principle that analyzing PowerShell at the character level is a highly effective strategy for dealing with obfuscation. The PowerShell-Sentinel project builds directly on this foundation, but modernizes the approach by replacing the CNN-based classifier with a more powerful, generative Large Language Model capable of not just classification, but structured analysis and deobfuscation.

The broader context for this modernization is captured in recent surveys such as that by *Kaur et al. (2024)*, which provides a comprehensive overview of how LLMs are being harnessed across the cybersecurity landscape. Their work highlights the critical need for “customized language models” and “domain-adaptive fine-tuning” to bridge the gap between general-purpose models and the specific, nuanced requirements of cybersecurity tasks. This thesis directly answers that call, presenting a detailed, end-to-end methodology for creating exactly such a customized model.

The specific techniques required to create this model are detailed by researchers like *Sharkey & Treleaven (2025)*, who provide a taxonomy of LLM optimization methods. They make a critical distinction between non-gradient-based methods like prompt engineering and gradient-based methods like fine-tuning. Their work details the mechanics of Parameter-Efficient Fine-Tuning (PEFT) techniques, specifically Quantized Low-Rank Adaptation (QLoRA), which drastically reduces the computational resources required to adapt multi-billion parameter models. The methodology in Chapter 5 of this thesis is a direct implementation of the QLoRA technique described by Sharkey & Treleaven, a choice made specifically to ensure that the fine-tuning of a

state-of-the-art model like *meta-llama/Meta-Llama-3-8B-Instruct* is feasible on consumer-grade hardware, making the entire project practical and reproducible.

The most significant challenge in implementing such a fine-tuning strategy is the scarcity of high-quality, specialized training data. This issue is not unique to cybersecurity. *Zhong et al. (2025)* faced an analogous problem in the domain of Text-to-Cypher translation. Their solution, "SyntheT2C," was to develop a sophisticated pipeline to generate synthetic, high-quality Question-Cypher pairs for fine-tuning. Just as Zhong et al. created SyntheT2C to address data scarcity in their domain, this thesis introduces the PowerShell-Sentinel Data Factory in Chapter 4 to solve the equivalent problem for obfuscated PowerShell analysis. Both approaches share a core philosophy: leveraging a combination of templates, automated generation, and rigorous validation to create a large-scale dataset where one does not naturally exist, thereby enabling the creation of a specialized model.

Finally, the work of *Ullah et al. (2024)* provides the ultimate justification for this data-centric approach. Their comprehensive evaluation, "LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?)," demonstrates that even the most advanced, general-purpose LLMs are unreliable, non-deterministic, and not robust when applied to security tasks. They show that simple changes to variable names or code structure can cause these models to fail, and that their reasoning is often incorrect even when their final answer is right. The findings of Ullah et al. strongly motivate the central hypothesis of this thesis: that a data-centric, fine-tuning approach is necessary to transform a general-purpose LLM into a specialized and, crucially, *reliable* security assistant. The PowerShell-Sentinel project is designed to address the very issues of non-determinism and lack of specialized reasoning that their work highlights, aiming to produce a model whose performance is both accurate and trustworthy.

Chapter 3

3. Lab Architecture and Data Curation Tooling

Focusing on foundational data quality, this chapter details the process of engineering a controlled lab environment on a cloud provider to generate high-fidelity telemetry. It describes the architecture, including domain-joined Windows virtual machines and a Splunk instance for log aggregation. The chapter outlines the human-in-the-loop curation workflow, managed by an interactive CLI, used to process an initial "starter pack" of PowerShell primitives and correlate them with their "golden signal" execution logs.

3.1. Introduction and System Overview

Focusing on foundational data quality, this chapter details the process of engineering a controlled lab environment and a comprehensive suite of data curation tools. This first experiment addresses the initial and most critical phase of the project: creating the validated, high-fidelity ground-truth dataset that serves as the seed for all subsequent automated generation and model fine-tuning. The chapter describes the cloud-based lab architecture, the instrumentation and log forwarding pipeline, and the Pydantic-based data modeling strategy that ensures robustness. It then outlines the human-in-the-loop curation workflow, managed by an interactive command-line interface, which is used to process an initial "starter pack" of PowerShell primitives and correlate them with their "golden signal" execution logs. The narrative will also cover a key design choice during development, where an initial plan for a static parser was replaced by a novel framework for interactive, analyst-driven telemetry parsing.

3.1.1. Need for High-Fidelity Telemetry

The core hypothesis of this research is that a highly specialized dataset is required to effectively fine-tune a language model for PowerShell analysis. Relying on public code repositories or general security forums would introduce significant noise and lack the specific, structured data needed. To generate high-fidelity data, two conditions must be met: causality and cleanliness. Causality is the guarantee that an observed log event was the direct and sole result of a specific command execution. Cleanliness is the absence of unrelated system "noise" that could be incorrectly correlated with the command. To achieve both, a controlled, isolated lab environment was

engineered. This environment ensures that when a PowerShell primitive is executed, the telemetry captured is a direct and unambiguous footprint of that action, forming a reliable ground-truth signal for the data factory.

3.1.2. Benefits of Human-in-the-Loop Curation

While a controlled lab provides clean, causal telemetry, the raw output is still a collection of noisy log events. The critical task of identifying which of these events constitute the true "golden signal" of a command's execution requires domain expertise. An automated statistical model alone cannot make this initial judgment without a ground-truth dataset to learn from. Therefore, a human-in-the-loop curation process is essential for bootstrapping the system's knowledge base. This approach leverages the expertise of a human analyst to perform the initial, high-value task of identifying the most significant telemetry for a "starter pack" of primitives. This creates a foundational, expert-validated dataset that provides the initial ground truth for all subsequent statistical analysis and automated data generation, effectively encoding human expertise into the system's core.

3.2. Lab Architecture and Configuration

To generate the high-fidelity telemetry required for the project's data-centric hypothesis, a controlled, isolated, and realistic lab environment was engineered. A cloud-based approach was chosen over local virtualization to ensure native x86_64 performance for all components and to mirror real-world network architectures, a key requirement for the project. Microsoft Azure was selected as the cloud provider, utilizing the "Azure for Students" program for resource allocation.

3.2.1. Cloud Environment and Virtual Machines

The lab architecture consists of three virtual machines (VMs) operating within a single Azure Virtual Network, ensuring seamless private network communication while isolating the lab from external traffic. The initial plan to use a local ARM-based machine for development was abandoned due to Python library incompatibilities, leading to a robust in-cloud, three-VM architecture. The first machine, *PS-DC-01*, is a Windows Server 2022 x86_64 VM configured as the Active Directory Domain Controller for a new forest, *powershellsentinel.local*. The primary purpose of this VM is to provide essential environmental context for domain-centric commands. A second Windows Server 2022 x86_64 VM, *PS-VICTIM-01*, was joined to this domain to serve as the "stage" for the data factory and the designated target where all PowerShell primitives are executed. The third machine, *PS-DEV-01*, was introduced to serve a dual role. It functions as the primary x86_64 development environment, hosting the Python codebase and tooling, and as the lab's SIEM host, running the Splunk Enterprise instance. This three-VM design, with its separation of roles, provides a stable and realistic Active Directory environment for telemetry generation. This three-VM design, with its

separation of roles, provides a stable and realistic Active Directory environment for telemetry generation, as illustrated in Figure 3.1.

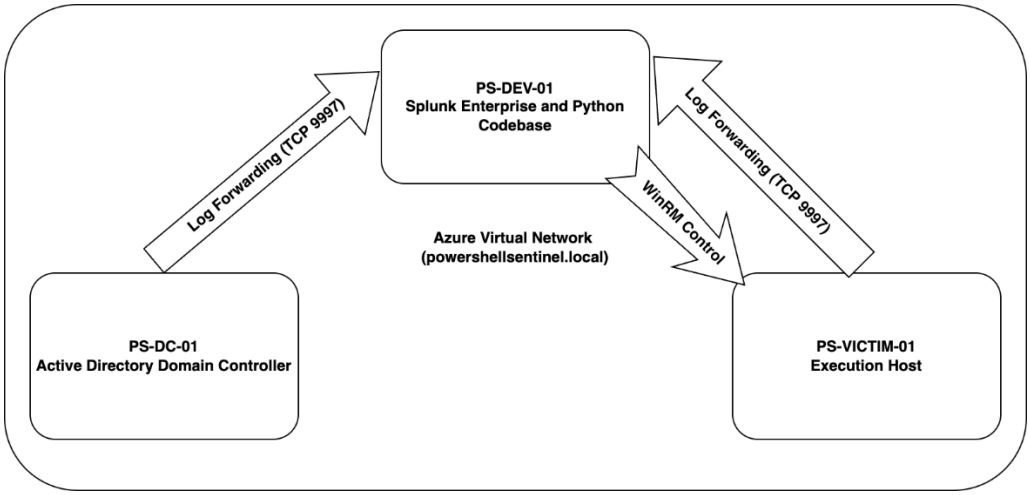

Figure 3.1: Lab Architecture Diagram

3.2.2. Instrumentation and Log Forwarding

To capture the necessary telemetry, a multi-layered instrumentation and logging pipeline was established. On the Domain Controller (*PS-DC-01*), a Group Policy Object (GPO) was created and linked to the domain to enforce the activation of PowerShell Module Logging (*Event ID 4103*) and the critical PowerShell Script Block Logging (*Event ID 4104*) across all domain-joined machines. This ensures the full content of executed scripts is logged. For deeper system visibility, System Monitor (Sysmon) was installed on the Victim Workstation (*PS-VICTIM-01*), using a standard robust configuration as a baseline. To aggregate this data, a free license of Splunk Enterprise was installed on the *PS-DEV-01* VM. To pipe the telemetry into Splunk, the Splunk Universal Forwarder was installed on both the DC and Victim VMs. These forwarders were configured to monitor the *Microsoft-Windows-PowerShell/Operational* and *Microsoft-Windows-Sysmon/Operational* event log channels and send the data to the private IP address of the *PS-DEV-01* machine over TCP port 9997. Establishing this connectivity required creating specific inbound rules in both the Windows Defender Firewall on the host and the Azure Network Security Group at the infrastructure level.

For deeper system visibility, System Monitor (*Sysmon*) was also installed on the Victim Workstation (*PS-VICTIM-01*), using a standard robust configuration as a baseline. This was a critical addition to the lab's instrumentation. The initial reliance on native PowerShell logs alone was hypothesized to create a "telemetry monoculture", which is a term that will be explored in Chapter 4, where every command would generate similar types of log events. By adding *Sysmon*, the forwarders were configured to also monitor the *Microsoft-Windows-Sysmon/Operational* channel, enabling the capture of a much richer and more diverse set of telemetry, such as granular process creations (*Event ID 1*) and network connections (*Event ID 3*). This decision was a foundational step in

ensuring the final dataset would have the necessary diversity to train a robust and widely applicable analytical model.

3.2.3. Code-to-Lab Interface

All programmatic interaction with the lab environment is abstracted into a single, dedicated Python module, *lab_connector.py*, as specified by the project requirements. This clean separation of concerns ensures that the rest of the data factory can operate without any knowledge of the underlying infrastructure's complexity. The module provides two core functions: *run_remote_powershell(command)*, which establishes a PowerShell Remoting session with the Victim VM over WinRM, and *query_splunk(search_query)*, which connects to the local Splunk Enterprise instance via its REST API to retrieve log data.

The validation of this critical module followed a rigorous, two-tiered testing strategy. First, unit tests were implemented in *tests/test_lab_connection.py*. These tests use the *unittest.mock* library to isolate the *LabConnection* class from the network, replacing the live *pywinrm* and *splunk-sdk* dependencies with test doubles. This approach allows for fast, deterministic verification of the module's internal logic, ensuring that method calls are structured correctly and data is handled as expected without requiring a live lab environment.

While the unit tests confirmed the software's correctness, a second integration test was created to mitigate the "unknown-unknown" risks of the live infrastructure. This "Golden Primitive Full-Loop Test," implemented in *tests/test_integration_lab_connection.py*, programmatically executed a unique command on the victim VM and then used the Splunk query function to retrieve the resulting log. The successful execution of this test provided definitive, end-to-end validation of the entire architecture, from command execution to log ingestion and retrieval.

3.3. Data Curation and Management

While the lab architecture provides the necessary raw telemetry, this data in its unprocessed state is not suitable for generating a machine learning dataset. To transform these raw logs into a structured, high-fidelity corpus, a comprehensive suite of *Data Curation and Management* tools was engineered. This tooling, orchestrated by a central, interactive command-line interface, is designed to automate the correlation of PowerShell primitives with their ground-truth telemetry signals. The following sections detail the foundational data schemas, the semi-automated workflow for telemetry discovery and curation, the backend correlation engine, and the novel framework developed for robust, interactive log parsing.

3.3.1. Defining Primitives and MITRE TTPs

To ground this research in established cybersecurity principles, the project uses the MITRE ATT&CK® framework as its conceptual backbone. ATT&CK is a globally accessible knowledge base of adversary tactics and techniques based on real-world observations. Within the framework, a Tactic represents the adversary's strategic goal—the "why" behind an action—while a Technique describes the specific method used to achieve that goal—the "how". A key focus for this project is the Discovery tactic, which includes techniques adversaries use to gain knowledge about a system and its network environment. Many of these techniques are "Living Off the Land" (LOTL), meaning they leverage dual-use commands: tools and executables native to the operating system (like *hostname* or *ipconfig*) that serve both legitimate and malicious purposes.

The foundation of the data factory is a curated library of these dual-use commands, known as primitives, stored in *primitives_library.json*. An initial "starter pack" of 33 primitives was developed to bootstrap the project, sourced and adapted from foundational learning materials teaching PowerShell fundamentals. To ensure a rigorous methodology, each primitive adheres to a strict JSON schema consisting of several key-value pairs. The *primitive_id* serves as a unique string identifier (e.g., "PS-001") for tracking, while the *primitive_command* contains the complete, executable PowerShell command string. The adversarial purpose is described in the intent field, an array of human-readable strings that correspond to MITRE Technique names. This is formally linked to the ATT&CK framework via the *mitre_ttps* field, an array containing the official Technique ID(s) (e.g., "T1082"). Finally, a *telemetry_rule*s array is included, which is intentionally left empty at this stage to be populated later by the data curation tooling.

In conjunction with the lab environment hardening described in Section 3.2.2, this initial starter pack was later enriched with a set of new, network-centric primitives (“PS-051” through to “PS-058”). This expansion was a key part of the second research iteration, deliberately designed to generate the more diverse *Sysmon* telemetry now available in the lab. These new commands, such as *(New-Object System.Net.WebClient).DownloadString(‘http://example.com’)*, were chosen specifically for their ability to trigger events like *Sysmon Event ID 3 (Network Connection)*, thereby providing the foundational data needed to break the “telemetry monoculture” of the initial dataset.

To support this structure, a second file, *mitre_ttp_library.json*, acts as a canonical, project-specific lookup table. It is structured as a single JSON object where each key is a TTP ID and its value is an object containing the technique's official name and its parent tactic(s). The *primitives_library.json* maintains a direct link to this library: the intent of a primitive corresponds to the name of a technique, and the *mitre_ttps* value corresponds to its key. This explicit, two-file system ensures that all references to the ATT&CK framework are consistent, accurate, and programmatically accessible. This explicit, two-file system ensures that all references to the ATT&CK framework are

consistent, accurate, and programmatically accessible. A sample of the resulting curated primitive objects is provided in Table 3.1.

Table 3.1: Sample of Curated Primitives

Primitive ID	Primitive Command	Sample Telemetry Rule
PS-009	whoami	Event ID: 4104, Details: whoami
PS-016	GetLocalGroupMember -Group “Administrators”	Event ID: 4104, Details: Get-LocalGroupMember...
PS-040	tasklist /V	Event ID: 4104, Details: tasklist /V
PS-044	reg.exe save hklm\sam C:\sam.save	Event ID: 4104, Details: reg.exe save hklm\sam...
PS-051	(New-Object System.Net.WebClient).DownloadString(...)	Event ID: 3 (Sysmon), Details: Network connection...

3.3.2. Telemetry Discovery and Curation

To systematically process the primitives library, a dedicated command-line interface, *primitives_manager.py*, was developed to orchestrate the entire human-in-the-loop workflow. This workflow is divided into two distinct phases: *Telemetry Discovery* and *Telemetry Curation*.

The *Telemetry Discovery* phase is responsible for generating the raw data. The manager iterates through each Primitive in the library and, using the *lab_connector.py* module, executes its command string on the remote victim VM. It captures a snapshot of Splunk logs immediately before and after execution. These two snapshots are then passed to the *snapshot_differ.py* module, which calculates the "delta"—a clean list of *SplunkLogEvent* objects representing only the telemetry generated by the command. This raw delta is then serialized and saved to the *data/interim/delta_logs/* directory, creating a persistent record of the raw execution output for each primitive.

The *Telemetry Curation* phase transforms this raw data into curated knowledge. The manager loads the saved delta logs for a primitive and begins the process of parsing, scoring, and selection. It leverages the interactive parsing framework to convert raw logs into structured *TelemetryRule* objects. These clean rules are then passed to the *recommendation_engine.py*, which uses the pre-calculated statistics to score and rank them. The analyst is presented with a prioritized list of recommended signals and makes the final selection. This interactive, semi-automated process ensures that the final "golden signals" are both statistically significant and validated by human expertise. Notably, the system is designed to handle the "cold start" problem; if the statistical engine has no prior data for a given primitive's telemetry, it intelligently falls back to presenting the full, unfiltered set of parsed logs to the analyst, ensuring that new knowledge can always be introduced into the system.

3.3.3. Correlation Engine Backend

The Correlation Engine is a backend software library responsible for processing the raw telemetry generated by a primitive's execution and identifying the most probable "golden signals" for curation. This engine is composed of a pipeline of four distinct, type-safe modules that work in sequence.

The first module, *snapshot_differ.py*, is responsible for isolating the telemetry of interest. It programmatically takes two lists of *SplunkLogEvent* models—one captured immediately before a primitive is executed and one after—and performs a set difference to produce a "delta". This delta represents only the log events that were generated as a direct result of the primitive's execution, effectively filtering out unrelated system noise.

The second module, *statistics_calculator.py*, computes two key metrics across the entire corpus of curated primitives in primitives_library.json. The first metric, *Global Rarity*, is an implementation of Inverse Primitive Frequency, which assigns a higher score to telemetry signals that appear in fewer primitives, making them statistically more significant. The second, *Local Relevance*, calculates the conditional probability *P(TelemetryRule|MITRE_TTP)*, identifying signals that are strongly correlated with a specific adversary technique.

The third module, *recommendation_engine.py*, acts as the statistical brain of the curation process. Its primary function is to score parsed *TelemetryRule* objects using the pre-calculated *Global Rarity* and *Local Relevance* statistics. A key feature of its design is the handling of the "cold start" problem. On the initial curation run, when no historical statistics exist, the engine operates in a "bootstrap" mode, returning all parsed signals to the analyst to establish a baseline of ground truth. In subsequent runs, it applies a heuristic filter, recommending rules that surpass defined thresholds. Crucially, if this filtering results in no recommendations for a given primitive (a common occurrence for new, unseen telemetry), the engine employs a fallback mechanism, presenting the full, ranked list of parsed signals to ensure the analyst can always curate new telemetry and continuously improve the knowledge base. The final output is a ranked list of the recommended *TelemetryRule* objects, sorted by relevance to provide the analyst with a clean, prioritized list of potential golden signals for their final selection.

The final module, *rule_formatter.py*, serves as a formal endpoint to the pipeline. Following the architectural pivot to an interactive, upfront parsing model, the list of telemetry signals selected by the user is already in the correct, structured *TelemetryRule* format. Therefore, this module now functions as an identity function, receiving the final list and passing it through, which confirms the data is in its correct state before being saved to the *primitives_library.json* file.

3.3.4. Data Modeling for Robustness

To ensure the methodological rigor and technical robustness of the data curation pipeline, a foundational decision was made to move away from standard Python dictionaries and adopt a formal data modeling and validation framework. The inherent fragility of dictionary-based data structures in a multi-stage process presents significant risks, including runtime errors from key mismatches, data type inconsistencies, and missing fields. To mitigate these risks, the Pydantic library was integrated to serve as the project's data contract engine.

This strategy was implemented by creating a central module, *powershell_sentinel/models.py*, to act as the single source of truth for all data structures used in the project. This module defines a series of Pydantic models that enforce a strict schema on data as it flows between components. Key models include the *Primitive* model, which validates every entry loaded from *primitives_library.json*, ensuring fields like *primitive_id* match a regex pattern and that *mitre_ttps* are valid members of a dynamically generated Enum. The *TelemetryRule* model provides the contract for clean, curated output signals, while the *ParsingRule* model defines the schema for the new user-defined parsing rules. Finally, models like *SplunkLogEvent* and *CommandOutput* ensure that all data originating from the live lab environment via *lab_connector.py* is structured and predictable from the moment it enters the system.

The adoption of this framework provides three critical benefits. The first is runtime safety and early failure detection. Pydantic validation acts as a gatekeeper; when loading data from a file, the application fails immediately with a clear *ValidationError* if any record is malformed, preventing corrupted data from entering the core processing logic. The second benefit is *increased code maintainability and readability*. The models serve as a form of executable documentation, making function signatures unambiguous (e.g., *def process(primitives: List[Primitive])*) and the logic cleaner. The third benefit is *enhanced developer velocity*, as the explicit data models enable modern IDE features like autocompletion and static type checking, which reduce bugs and accelerate the implementation of subsequent modules.

3.3.5. A Framework for Interactive Telemetry Parsing

A foundational challenge in processing security telemetry is the reliable mapping of diverse, unstructured raw log events into a consistent, structured format suitable for analysis. An initial design considered a static, regex-based parser, but this was dismissed as too brittle; it would require constant code updates by a developer to support new log formats. A second alternative, using a general-purpose Large Language Model for zero-shot parsing, was also evaluated. While more flexible, this approach introduced risks of non-determinism, API dependency, and potential "hallucinated" outputs, which were deemed unacceptable for a high-fidelity data generation pipeline. To solve this, a novel *Framework for Interactive Telemetry Parsing* was developed and

integrated directly into the *primitives_manager.py* CLI. This framework moves the responsibility of defining parsing logic from the developer to the analyst, creating a robust, deterministic, and human-in-the-loop system. When the curation workflow encounters a *SplunkLogEvent* that cannot be parsed by any existing rule, the system does not fail; instead, it engages the analyst. The CLI presents the raw log to the user and prompts them to provide the necessary metadata—such as the Event ID, the extraction method (*REGEX* or *KEY_VALUE*), and the specific key or pattern to extract the details from.

This user-provided information is used to create a new *ParsingRule* object, which is then appended to the *data/source/parsing_rules.json* file. This file becomes a persistent, ever-growing library of deterministic parsing instructions. The system immediately uses this new rule to parse the current log and all subsequent logs of the same type it encounters, effectively learning from the analyst's expertise. This methodology provides the ideal balance of automation and reliability, ensuring that every log is parsed according to an explicit, human-validated rule.

3.4. Results and Discussion

3.4.1. Creation of a Domain-Specific Curated Dataset

The primary output of the data curation tooling is the enrichment of the *primitives_library.json* file, transforming it from a simple list of commands into a domain-specific, high-fidelity dataset of malicious behaviors and their corresponding telemetry footprints. By systematically applying the *Telemetry Discovery* and *Telemetry Curation* workflows using the *primitives_manager.py* tool, the initial “starter pack” of primitives is expanded, and each entry is populated with a list of validated “golden signals”.

The outcome of this process is that each *Primitive* object in the library contains a populated *telemetry_rules* attribute. This attribute holds a list of *TelemetryRule* objects, where each rule represents a specific, human-readable log event that is a direct and verified consequence of that primitive’s execution. This curated dataset forms the ground-truth foundation for the entire project. It is this explicit, validated mapping between a command’s intent (e.g., *T1057: Process Discovery*) and its telemetry signature (e.g. a *TelemetryRule* for Event ID 4104 containing *Get-Process*) that will be used to generate the large-scale training data for the fine-tuning phase.

3.4.2. Validation of the Correlation Engine

The culmination of this data curation process is the handoff of the curated knowledge for external validation. To facilitate this, the *primitives_manager.py* tool was used to assemble a “Practitioner Review Package,” a self-contained zip archive containing a diverse subset of primitives, their raw *delta_logs*, and instructions for a blind review. This package was provided to an external

cybersecurity expert, namely a professional penetration tester, whose manual selections of “golden signals” established a definitive, expert-validated ground truth dataset.

To quantitatively validate the effectiveness of the semi-automated Correlation Engine, a rigorous “bake-off” experiment was then conducted against this ground truth. An automated test harness simulated the full curation workflow for each primitive in the package, comparing the engine’s programmatic recommendations against the practitioner’s selections to calculate the overall Precision, Recall, and F1-Score. The results of this validation are presented in Table 3.2.

Table 3.2: Correlation Engine Performance vs. Expert Ground Truth

Metric	Score
True Positives	37
False Positives	1
False Negatives	0
Precision	97.37%
Recall	100.00%
F1-Score	98.67%

The Correlation Engine achieved an F1-Score of 98.67%, demonstrating a near-perfect alignment between the signals prioritized by the statistical model and those selected by a human expert. The perfect Recall score of 100.00% is a particularly significant finding, proving that the engine successfully identified every single one of the practitioner’s golden signals. Furthermore, the Precision of 97.37% indicates that the engine produced only a single False Positive across the entire test set, validating its ability to filter out noise with a high degree of accuracy.

These results provide strong empirical validation for the core hypothesis of this experiment: that a statistical model combining Global Rarity (signal uniqueness) and Local Relevance (signal correlation to tactic) is a highly effective method for automatically identifying high-value telemetry.

It is important to contextualize these results with the practitioner’s specific domain expertise. As a penetration tester, their perspective is primarily offensive, focused on confirming that a command was executed successfully. While this provides an excellent validation for identifying the most direct and causal log events, a key area for future work would be to repeat this validation with defensive practitioners, such as Security Operations Center (SOC) analysts or threat hunters. These professionals might prioritize different “golden signals” based on their forensic value or uniqueness in a noisy enterprise environment. Such a comparative study would further enrich the ground-truth dataset and potentially lead to refinements in the statistical model. Nonetheless,

successful validation against an expert baseline provides the necessary confidence in the curated data that will serve as the foundation for the automated data factory in the next chapter.

3.5. Summary

This chapter detailed the first of three core research experiments, focused on establishing foundational data quality. It presented the architecture of a stable, three-VM Active Directory lab designed for high-fidelity telemetry generation. It then described the suite of data curation tooling, orchestrated by the *primitives_manager.py* CLI. Key methodological innovations were introduced, including the architectural pivot to Pydantic for data robustness and the development of a novel "Framework for Interactive Telemetry Parsing" to replace a brittle, static parser. The experiment culminated in the creation of an expanded, domain-specific library of primitives and the dispatch of a "Practitioner Review Package" to enable external validation of the curation engine's backend.

Chapter 4

4. The PowerShell-Sentinel Data Factory

This chapter describes the iterative engineering process behind the PowerShell-Sentinel Data Factory, the core technical contribution of this research. This second experiment moves beyond the manual curation process detailed in Chapter 3 to implement and evaluate a fully automated software pipeline. The chapter details the evolution of the system's architecture, beginning with a baseline implementation and culminating in a re-engineered, deterministic factory. It documents the engineering solutions to ensure stability, a multi-stage quality assurance framework that validates functional equivalency, and the quantitative analysis that proves the superiority of the final design.

4.1. Introduction and System Overview

Building upon the curated, high-fidelity dataset established in the previous experiment, this chapter details the design, implementation, and validation of the PowerShell-Sentinel Data Factory. This factory is the core contribution of the research, designed to address the critical challenge of data scarcity in training specialized models for cybersecurity applications. This chapter first describes the initial system architecture, composed of a modular obfuscation engine and a master controller. It then details the rigorous quality assurance process, combining live execution validation with prophylactic system maintenance, that was applied to both the baseline and the final, re-engineered pipeline to guarantee the integrity and reliability of the training dataset.

4.1.1. Objectives of Automating Dataset Generation

The primary objective of creating an automated data factory is to overcome the inherent limitations of manual data curation and, ultimately, to produce a tool that can augment and scale the capabilities of security analysts. The process is designed not to replace human expertise, but to leverage and amplify it. The factory takes the "golden signal" telemetry—identified and validated by human expertise in the preceding data curation experiment—as its foundational input. From this trusted, human-curated seed data, the automated pipeline programmatically generates thousands of functionally equivalent yet syntactically diverse examples. This methodology transforms the slow, one-to-one task of manual analysis into a highly efficient, one-to-many generation process. It ensures that the deep, contextual knowledge of the analyst is encoded into a

large-scale dataset, providing the breadth and variety required to train a reliable and practical analysis assistant.

4.2. Pipeline Architecture and Components

This section details the technical architecture of the PowerShell-Sentinel Data Factory, documenting its evolution from a baseline implementation to a robust, reconstructed engine. It describes the design of the core software components for both architectures, including the obfuscation engine, master controllers, and the critical code-to-lab interface. It also documents the significant engineering challenges that motivated the architectural rework and the solutions implemented to ensure the final pipeline's stability and resilience.

4.2.1. The Layered Obfuscation Engine

The creative core of the initial data factory was its obfuscation engine, a Python module designed to programmatically transform simple, canonical PowerShell commands into complex, functionally equivalent variants. Early development revealed that naively applying sequential transformations often resulted in invalid syntax. To solve this, a robust "Payload and Wrappers" architecture was engineered as the baseline methodology for this research.

This model distinguished between different classes of obfuscation. The engine implements five distinct techniques: string concatenation, Base64 encoding, dynamic type casting, variable assignments, and use of the format operator. A master function, *generate_layered_obfuscation*, orchestrated these methods through a randomized process. It would first apply one or two "wrapper" techniques (such as concatenation or variable assignment) to the base command to create a self-contained, executable payload. An optional, final layer of Base64 encoding could then be applied to the entire payload. This layered approach was designed to resolve parser errors and enable the generation of nested, executable, commands.

4.2.2. The Master Controller and Resilient Generation

The orchestration of the entire generation process in the initial pipeline was managed by the master controller, implemented in the *main_data_factory.py* script. This module was responsible for loading the curated *primitives_library.json*, orchestrating the randomized obfuscation and execution workflow, validating results, and serializing the final *TrainingPair* objects.

To ensure the generated dataset was not biased towards a small subset of simple primitives, the controller employed an intelligent "round-robin" selection strategy, iterating sequentially through all primitives. Recognizing that data generation is a long-running, the pipeline was engineered for resilience. The master controller implemented checkpointing and resumable session features,

allowing it to recover from network failures or system shutdowns. This design proved critical for enabling unattended, multi-hour generation runs. However, as the experimental evaluation in Section 4.3 will detail, the performance of this first-pass architecture revealed significant limitations, motivating a complete re-engineering of the obfuscation engine and generation strategy.

4.2.3. A Resilient Code-to-Lab Interface

Early high-volume tests of the data factory revealed a critical stability issue where the pipeline would invariably hang after generating approximately 1,000 pairs. The root cause was identified as resource exhaustion within the Windows Remote Management (WinRM) service on the target VM, which has default quotas for concurrent shells and memory per user. The factory's rapid execution of complex commands was exceeding these quotas, causing the service to refuse new connections.

To address this, the *lab_connector.py* module was re-engineered to provide a hardened and resilient interface. The *run_remote_powershell* function was refactored to wrap every command inside a server-side PowerShell *Start-Job* block with a 25-second timeout, preventing any single command from freezing the pipeline. For ultimate transport safety, this entire wrapper script is Base64-encoded and executed via *powershell.exe -EncodedCommand*. Furthermore, to combat gradual resource degradation on the server, a prophylactic reset mechanism, the *reset_shell* method, was introduced. This function performs a full teardown and rebuild of the WinRM connection and is called automatically by the master controller every 250 iterations to ensure a clean state. Finally, a new prerequisite was documented: the WinRM service on the target VM must be configured with higher resource quotas, a check which can be validated using the *scripts/verify_lab_config.py* utility.

4.2.4. Quality Assurance and Validation of Functional Equivalency

A multi-stage quality assurance strategy was designed to rigorously validate not only the functional equivalency of all generated data but also the operational stability of the factory itself. This framework combines deep validation of the obfuscation engine, thorough testing of the pipeline’s components and their integration, and a final, live-fire validation for every command before it is included in the dataset.

The first stage of this process focuses on proving the functional correctness of the generated data. This occurs in the *tests/modules/test_obfuscator.py* suite, which performs a stringent "output-level" validation. This confirms not just that obfuscated commands are syntactically valid, but that they produce the exact same standard output as their clean counterparts. This is accomplished using a "Canary Cage" of PowerShell primitives with known, deterministic outputs. A helper function, *_validate_execution*, programmatically runs the obfuscated command, asserts its exit code is zero, its standard error is empty, and, most critically, that its standard output is identical to the

known ground-truth output. By applying this three-part check to over 100 layered combinations, the functional correctness of the obfuscation engine itself is definitively proven. The second and final stage of data validation for the baseline factory was the “Live Execution Validation” check performed by the *main_data_factory.py* controller. For every pair the factory generated, it used the *lab_connector.py* module to execute the final, potentially Base64-encoded command in the live lab. It only accepted the command as a valid training sample if the exit code of this top-level execution was 0.

This two-stage QA process guaranteed the *end-to-end functional viability* of every training sample. However, it is crucial to note a limitation inherent in this approach when dealing with transport-level encoding like Base64. The “Live Execution Validation” could only confirm that the outermost command (e.g., *powershell.exe -EncodedCommand ...*) would execute successfully. It could not distinguish between a payload that was syntactically correct and one that was syntactically broken but “masked” by the Base64 encoding. In the latter case, the *powershell.exe* process would launch successfully (returning exit code 0 to the lab connector) but would then fail internally when attempting to decode and parse the malformed payload. This subtle but critical distinction is the root cause of the “masked invalidity” and dataset bias that was later diagnosed in Section 4.4.2.

In parallel with ensuring data correctness, a comprehensive suite of tests was developed to validate the stability and resilience of the data factory pipeline. An end-to-end smoke test, *test_master_pipeline_smoke.py*, was implemented to provide a fast, isolated verification that the master controller, obfuscation engine, and lab connector were correctly integrated, using mock objects to bypass network dependencies. To address the critical stability failures discovered during initial high-volume runs, two specific integration tests were created. The *test_integration_high_volume.py* test directly validated that the hardened *lab_connector.py* could successfully sustain a continuous workload without crashing. Complementing this, the *test_integration_blackhole.py* test was designed to prove the system's resilience against worst-case scenarios by demonstrating that the timeout and job-based wrappers could gracefully handle known problematic commands that would otherwise freeze the pipeline. Finally, regression tests within *test_main_data_factory.py* ensure that critical maintenance logic, such as the prophylactic shell reset, is triggered correctly.

This multi-faceted QA strategy, covering data correctness, component integration, and operational resilience, provides a higher degree of confidence in the integrity of the final dataset and the robustness of the data factory as a reusable software artifact.

4.2.5. A Deterministic, Reconstructed Obfuscation Engine

The evaluation of the baseline factory's performance (detailed in Section 4.3.1) revealed that its randomized, string-based obfuscation engine was the primary source of the low success rates and resultant dataset bias. The brittleness of this engine necessitated a complete architectural rework to create a more robust and reliable system.

An approach was then evaluated: wrapping the industry-standard *Invoke-Obfuscation* PowerShell framework. While powerful, this tool is fundamentally designed as an interactive REPL (*Read-Eval-Print Loop*). Extensive integration testing proved that automating this interactive shell from a Python controller was unreliable, producing inconsistent outputs and failing to handle the complex, multi-layered command strings generated during the process.

These findings led to a final, successful pivot: a surgical reconstruction of the obfuscation logic. Instead of wrapping the entire interactive tool, the core, non-interactive "worker" functions were identified and extracted directly from the *Invoke-Obfuscation* source code. These battle-tested functions, which perform discrete Abstract Syntax Tree (*AST*) and token-level manipulations, were then integrated into a new, lightweight, and purpose-built PowerShell script module, *PowerShellSentinelObfuscator.psm1*.

This reconstructed engine represents a significant methodological improvement. It retains the full, AST-aware power of the original state-of-the-art tool but discards the problematic interactive wrapper. The new module exposes a clean, script-friendly interface, where each obfuscation technique is a simple, deterministic function call. This architecture provides the speed and reliability of a custom-built tool with the robustness of a mature, industry-standard framework, forming the foundation of the enhanced data factory.

With a robust engine now in place, the generation strategy itself was re-engineered away from randomization towards a deterministic, exhaustive methodology. A thorough analysis was conducted on the interaction effects of layering the available obfuscation techniques. This analysis revealed a clear layering hierarchy: argument-level obfuscations must be applied before command-level or execution-level techniques. For example, applying a command obfuscator like *Invoke-SentinelCommand* first would replace a cmdlet like *Get-Process* with a variable, thereby destroying the token that subsequent argument-level techniques would need to target. This analysis established a set of valid "layering sequences", or ordered combinations of techniques, that could be applied to a primitive. The final reconstructed factory was therefore designed to systematically iterate through every primitive and apply every single valid layering sequence, thereby guaranteeing a complete and balanced exploration of the defined obfuscation space.

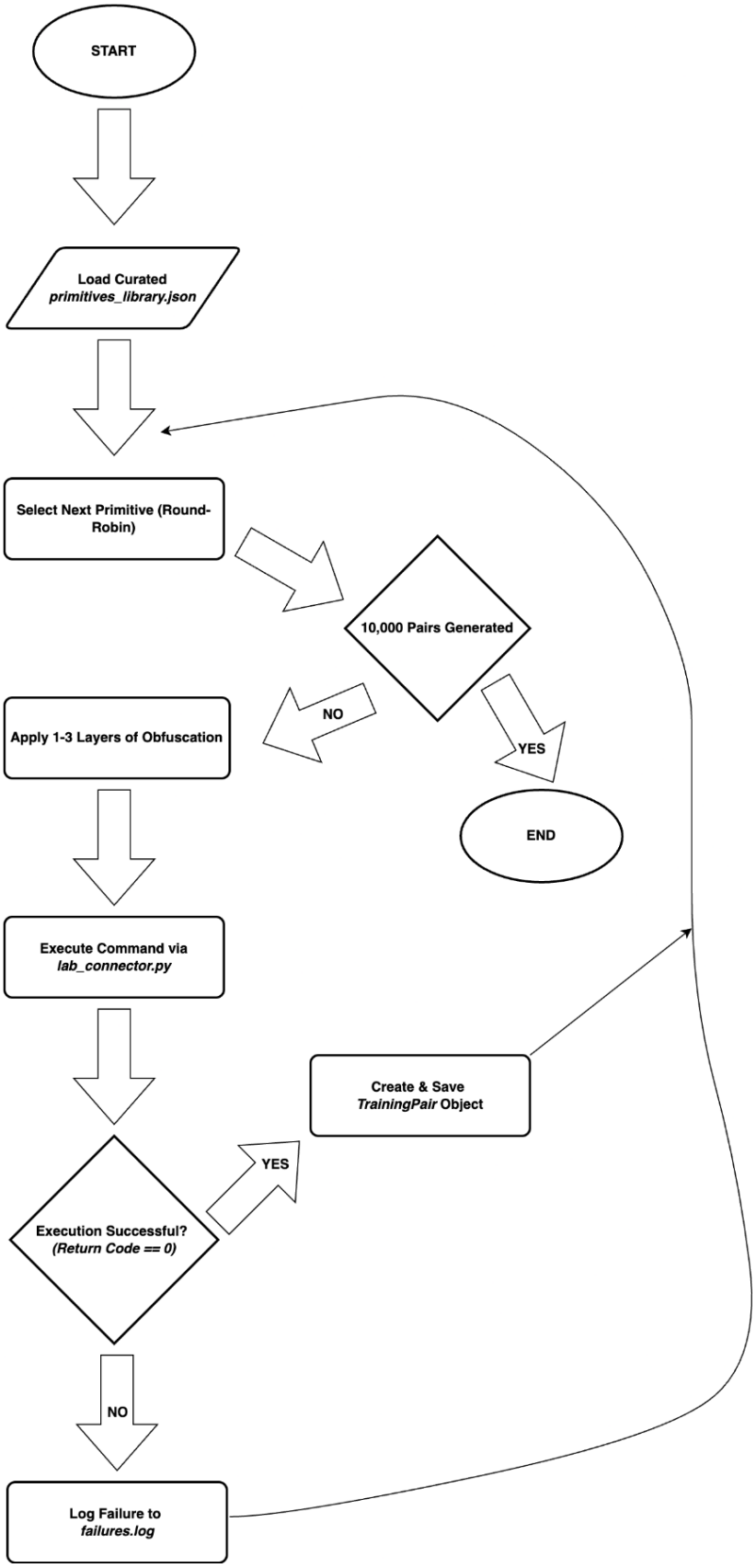

Figure 4.1: PowerShell Sentinel First-Pass Data Factory Pipeline

Table 4.1: Multi-Stage QA Framework

QA Stage	Purpose
Obfuscator Unit Testing (*test_obfuscator.py*)	Validates the functional equivalency of obfuscated commands, ensuring they produce the same standard output as their clean counterparts
Live Execution Validation (*main_data_factory.py*)	Guarantees the end-to-end functional viability of every training sample by executing it in the live lab and accepting only commands with a zero return code
Pipeline Integration Testing (*test_integration_*.py*)	Validates the end-to-end stability and resilience of the pipeline against high-volume workloads and known failure scenarios ("black hole" commands)

4.3. Experimental Evaluation

Following the implementation of the two distinct architectures detailed in Section 4.2, a comparative experimental evaluation was conducted to assess and quantify the performance of each pipeline. This section presents the key metrics from both production runs, directly comparing their overall success rates and failure modes. These results serve to empirically validate the decision to re-engineer the pipeline and provide context for the characteristics of the final, superior dataset discussed in Section 4.4.

4.3.1. Metrics: Baseline Performance Evaluation

The primary metric for the first-pass pipeline is the *Unique Generation Success Rate*, a refined measure that accounts for the duplication inherent in the randomized generation process. A successful pair is one whose obfuscated command passes the “Execution Validation” stage with a return code of 0. However, a post-generation analysis revealed that the factory’s “convergence effect”—where the random process rediscovers a finite set of robust obfuscation patterns—produced a significant number of duplicate prompts.

To address this, a de-duplication step was introduced as the final stage of this pipeline, ensuring the dataset used for model training contains only unique examples. The complete production run statistics, reflecting both the raw and the de-duplicated final counts, are detailed in Table 4.2.

Table 4.2: Data Generation Run Summary

Value	Metric
Target Pair Count	10,000
Total Raw Pairs Generated	10,000
Unique (De-Duplicated) Pairs Generated	5,686
Failed Pairs (Execution Errors)	14,380
Total Generation Attempts	24,380
Unique Generation Success Rate (%)	23.32%

The final Unique Generation Success Rate of 23.32% (based on the 5,686 unique pairs from 24,380 total attempts) is a more accurate and rigorous measure of the factory's true yield. The high number of both execution failures and duplicates is a direct consequence of the stringent quality control filters, which discard any non-functional or redundant command variants. This multi-stage filtering ensures that the final de-duplicated dataset is composed entirely of unique, functionally validated *TrainingPair* objects.

The first-pass data factory's performance was evaluated based on its operational stability and the failure patterns observed during the generation process. The final production run operated continuously for over 24 hours without crashes or hangs, validating the stability of the re-engineered architecture. An analysis of the 14,380 logged failures reveals strong correlations between failure rates and specific primitives or obfuscation techniques, as shown in Tables 4.3 and 4.4.

Table 4.3: Most Frequent Primitive Failures

Primitive ID	Failure Count	% of Total Failures
PS-001	1,276	8.87%
PS-045	1,239	8.62%
PS-048	1,229	8.55%
PS-046	1,129	7.85%
PS-044	1,034	7.19%

The primitives with the highest failure rates, detailed in Table 4.3, were predominantly those involving commands with complex quoting and special characters (e.g., *reg.exe*), suggesting these are more susceptible to syntax errors when obfuscated via string manipulation.

Table 4.4: Obfuscation Technique Failure Frequency

Technique	Failure Count
obfuscate_variables	5,739
obfuscate_format_operator	5,669
obfuscate_concat	5,627
obfuscate_types	4,665
obfuscate_base64	2,146

As shown in Table 4.4, the data indicates that syntax-based obfuscation methods (*variables, format_operator,* etc.) are significantly more prone to generating non-executable commands than the transport-level *obfuscate_base64* technique. The server-side timeout mechanism was essential for managing the frequent hangs caused by these invalid commands, enabling this pipeline's continuous operation. This tendency to select for syntactically robust patterns is the primary driver of the prompt duplication addressed in the subsequent MLOps pipeline.

4.3.2. Metrics: Reconstructed Performance Evaluation

Following the architectural rework, a full production run was conducted using the enhanced, deterministic data factory. The objective was to quantitively assess the expected performance improvements over the baseline methodology. The factory was configured to process the complete set of 58 primitives against all 228 unique, valid obfuscation layering sequences, representing a theoretical workload of 13,224 generation attempts. The final audit log recorded a total of 13,260 attempts; the minor discrepancy is attributed to the robust resume functionality reprocessing a small number of jobs after the run was interrupted and restarted.

The results, summarized in Table 4.5, demonstrate a profound improvement in the efficiency and reliability of the data generation process.

Table 4.5: Baseline vs. Reconstructed Factory Performance Comparison

Metric	Baseline Factory	Reconstructed Factory	Change
Generation Strategy	Randomized	Deterministic	-
Total Generation Attempts	24,380	13,260	-45.6%
Successful Pairs Generated	5,686	9,550	68.0%
Generation Success Rate	23.3%	72.0%	208.8%

The reconstructed factory achieved an overall success rate of 72.02%, a more than threefold improvement over the 23.32% rate of the baseline architecture. This validates the surgical reconstruction of the obfuscation engine and the shift to a deterministic generation model as a

superior approach. The new engine's ability to guarantee syntactically valid output dramatically reduced execution failures, leading to a much higher yield of usable training pairs.

A deeper analysis of the 3,710 failed attempts from this run provides critical insights into the practical limits of PowerShell obfuscation. As shown in Table 4.6, the vast majority of failures were not caused by errors in the generation engine itself, but by fundamental constraints of the remote execution environment.

Table 4.6: Reconstructed Factory Failure Analysis

Failure Type	Failure Count	Percentage
Lab Failure (*failure_lab*)	3,251	87.6%
Skipped (*skipped_exclusion*)	380	10.2%
Engine Failure (*failure_engine*)	79	2.1%
Total Failures	3,710	100.0%

Table 4.7: Top Lab Failure Reasons

Failure Reason	Failure Count
The command line is too long	3,186
Command timed out on the server after 25 seconds	36
WinRM Transport / Connection Error	15
Other (e.g. parsing errors, cmdlet not recognized)	14

The data are unequivocal: the reconstructed engine is significantly more robust, with engine failures accounting for only 2.1% of all non-successful attempts. The dominant failure mode (accounting for over 97.9% of all lab failures) was the operating system's maximum command-line length limit. This indicates that the factory was so effective at applying complex, multi-layered obfuscation sequences that it successfully pushed beyond the practical limitations of the execution environment itself. This finding serves to validate the construction of the pipeline and provides crucial context for the composition of the final dataset.

4.4. Results and Discussion

The data factory, engineered across two distinct architectural iterations, produced two large-scale datasets for fine-tuning a language model. This section describes the structure of these datasets and provides a comparative quantitative analysis of their composition. This then details the methodological improvements of the re-engineered factory and presents evidence that these changes successfully resolve the critical flaws of the initial approach. Finally, it identifies the

higher-level challenge of scalability that emerged as a consequence of this success, framing the direction for future work.

4.4.1. The Generated Datasets

The primary output of this research is a pair of structured datasets. The baseline factory, with its randomized engine, produced *training_data_v0.json*, a dataset consisting of 5,686 unique, validated training examples. The subsequent re-engineered factory, with its deterministic engine, produced the final *training_data_v2.json*, a significantly large and more balanced dataset of 9,550 unique pairs. In both datasets, each entry is a *TrainingPair* object, which encapsulates the complete information required for fine-tuning the language model.

A single *TrainingPair* consists of two main fields: prompt and response. The prompt field contains a single string: the fully obfuscated, multi-layered PowerShell command that was proven to be executable in the lab. The response field contains a nested JSON object that represents the ideal, structured output the LLM is expected to generate. This response object itself has two key attributes: *deobfuscated_command*, which holds the original, clean primitive command string, and analysis. The analysis object contains the ground-truth labels for the command, including its *intent*, its corresponding *mitre_ttps*, and the expected *telemetry_signature* derived from the curated "golden signals" established in the first data curation experiment. This structure ensures that each record is a self-contained, complete learning example, mapping a complex, obfuscated input to its simple, deobfuscated ground truth and its associated security metadata.

4.4.2. The Generated Datasets: Limitations and Improvements

A critical analysis of the two datasets reveals the profound impact of the architectural evolution of the data factory. The limitations of the baseline dataset provided the direct motivation for the improvements in methodology implemented in the re-engineered pipeline. Evaluation of the final, de-duplicated baseline dataset reveals an extreme bias that was a direct symptom of the underlying engine’s brittleness. The process of filtering thousands of execution failures and duplicates exposed the extent to which the random generation process had converged on a single, dominant obfuscation strategy, as detailed in Table 4.8.

Table 4.8: Inferred Final-Layer Obfuscation Bias

Technique	Count	Occurrence (%)
obfuscate_base64	5,685	99.98%
obfuscate_types	1	0.02%

As shown in the table, an overwhelming 99.98% of the unique, successfully generated training pairs utilize Base64 as the final encoding layer. This stark outcome is a direct result of the high

failure rates of syntax-based techniques, string-manipulation techniques. This convergence demonstrates a critical flaw: the Base64 encoding acted as a transport-level wrapper that masked the syntactic incorrectness of many of the underlying payloads. These malformed commands would have failed immediately if executed directly, but were able to pass the initial shell validation when encoded. This indicates that the baseline factory did not select for robust obfuscation, but rather for payloads that could merely survive the brittle generation process. While analysis of the decoded payloads (Table 4.9) confirmed a rich variety of techniques within the Base64 "shell", the primary limitation was clear: the model would have almost no exposure to commands where other techniques formed the outermost, and most syntactically fragile, layer.

Table 4.9: Inferred Technique Distribution Inside Base64 Payloads

Technique	Presence in Sample (%)
obfuscate_concat	84.2%
obfuscate_variables	60.2%
obfuscate_format_operator	22.8%
obfuscate_types	10.2%

The initial design for a future version of the factory proposed two main improvements to address these issues: a "stratified generation" strategy to enforce a balanced distribution, and a "stateful generation" mechanism to reduce the inefficiency of generating duplicates. During the engineering process of the reconstructed factory, these ideas evolved into more robust and methodologically superior solutions. The concept of "stateful generation" was implemented as the resilient *completion_log.json* system, which provided the crucial checkpointing and resume capability that enabled the long-running, multi-session generation process to succeed. More significantly, the initial idea of "stratified generation" was made entirely obsolete by the development of the deterministic engine grounded in "layering sequences". Instead of merely forcing a random process to be balanced, this model guarantees as balanced and exhaustive exploration of the defined obfuscation space as is feasible by systematically generating every possible valid layering sequence.

This re-engineered factory successfully solved the dataset bias problem. While the baseline factory produced a dataset where 99.98% of samples used a single finisher technique, the reconstructed factory produced a vastly more diverse and representative output. As shown in Table 4.10, the final *training_data_v2.json* dataset contains a rich distribution across all three finisher categories, accurately reflecting their real-world success rates when applied to the primitive library.

Table 4.10: Final-Layer Technique Distribution in Reconstructed Dataset

Finisher Technique	Successful Pairs	Percentage of Total
Invoke-SentinelCommand	3,539	37.1%
[None] (Argument Obfuscation Only)	3,791	39.7%
Invoke-SentinelBase64	2,220	23.2%
Total	9,550	100.0%

The data shows that layering sequences with argument obfuscation only were the most reliable, while those ending in *Invoke-SentinelBase64* were the least, primarily due to the "command line too long" failures. This nuanced distribution provides the model with a realistic, balanced diet of various final-layer obfuscation patterns, directly addressing the core limitation of the baseline dataset.

This re-engineered factory successfully solved the dataset bias problem. While the baseline factory produced. The power of the deterministic model lies in its extensibility. However, this reveals a new, higher-level challenge: scalability. The number of unique, valid layering sequences grows exponentially as new techniques or recursion depths are added.

Table 4.11: The Combinatorial Explosion of a Deterministic Generation Model

Parameters	Current	Modest Expansion	Ambitious Expansion
Argument Wrappers	2	3	4
Argument Destructive	1	2	3
Finishers	2	3	4
Primitives	58	75	100
Total Jobs in Queue	~13,000	~90,000	~500,000+
Est. Serial Run Time	~20 Hours	~5+ days	~4+ Weeks

As demonstrated in Table 4.11, while the re-engineered factory is robust, its serial architecture presents a significant bottleneck. A modest expansion of the technique library would increase the generation time from hours to days. This combinatorial explosion makes a serial approach intractable for creating truly comprehensive datasets. This finding is a key result of the research, proving that while the *reliability* problem has been solved, the *scalability* problem remains. This directly motivates the need for a concurrent architecture, the design of which is proposed as the primary avenue for future work in Chapter 6.

4.5. Summary

This chapter has detailed the second core experiment of this research: the iterative engineering and evaluation of the PowerShell-Sentinel Data Factory. It began by presenting the baseline architecture, a fully automated pipeline featuring a randomized “Payload and Wrappers” obfuscation engine. A rigorous experimental evaluation of this initial system was presented, providing a quantitative diagnosis of its core methodological flaws: a low generation success rate and a consequent, profound dataset bias towards a single, syntactically robust obfuscation technique.

Motivated by these findings, the chapter then documented a second research iteration: a complete architectural rework of the factory. This involved a principled engineering pivot, replacing the brittle, random engine with a surgically reconstructed, deterministic engine built from the core logic of state-of-the-art tooling. The new factory’s design, based on a validated, combinatorial model of 228 unique layering sequences, was detailed.

Finally, a comparative analysis of the reconstructed factory’s production run was presented. The data provided definitive, quantitative proof of the new architecture’s superiority, demonstrating a threefold increase in the generation success rate (from 23.3% to 72.0%) and the successful mitigation of the original dataset’s bias. The analysis of the new factory’s failure modes revealed that the primary bottleneck had been elevated from engine brittleness to the physical command-line length limitations of the target OS. This critical finding on the practical limits of complex obfuscation led to the architectural design of a scalable, concurrent pipeline, establishing the clear direction for future work. The chapter concluded with the delivery of the final, high-fidelity dataset of 9,550 unique command-analysis pairs ready for model training.

Chapter 5

5. Model Fine-Tuning, Evaluation and Delivery

This chapter presents the MLOps pipeline used to train, evaluate, and package the final language model. It details the dataset partitioning, the prompt engineering methodology used to select an optimal template, and the use of QLoRA to fine-tune a state-of-the-art meta-llama/Meta-Llama-3-8B-Instruct model. The chapter then defines the rigorous evaluation metrics, including a strict "JSON Parse Success Rate" and F1-score, used to quantify the model's performance against a locked test set. The results of this quantitative evaluation are presented and discussed. Finally, the chapter details the post-training quantization process, including a qualitative analysis used to select the optimal GGUF format for the final deliverable: a standalone, user-facing command-line toolkit.

5.1. Introduction and Rationale

This section details the rationale and methodology for the third and final experiment: the fine-tuning and evaluation of a specialized language model. It outlines the motivation for moving beyond a general-purpose model to create a highly specialized tool. Key technical decisions are explained, including the selection of the Parameter-Efficient Fine-Tuning (PEFT) method QLoRA, chosen for its ability to enable training on consumer-grade hardware, and the selection of the *meta-llama/Meta-Llama-3-8B-Instruct* model, a state-of-the-art foundation for instruction-following tasks. The section concludes by defining the expected gains in analytical accuracy and reliability that this specialized approach is hypothesized to deliver.

5.1.1. Motivation for Fine-Tuning

The motivation for fine-tuning is twofold. First, it aims to teach the model a new, highly specific skill: the deconstruction of obfuscated PowerShell and the generation of a structured JSON output that conforms to a strict data schema. This is a format-following task that general-purpose models often struggle with, leading to unreliable, malformed outputs. Second, fine-tuning allows for the creation of a much smaller, more efficient model deliverable. The methodology selected for this task is Quantized Low-Rank Adaptation (QLoRA). This Parameter-Efficient Fine-Tuning (PEFT) technique works by freezing the weights of the large pre-trained model in a 4-bit quantized state

and injecting a small number of trainable, low-rank adapter matrices. This drastically reduces the memory footprint required for training, making it feasible to fine-tune a multi-billion parameter model on a single, high-end consumer-grade GPU. This efficiency is critical for enabling the development of a powerful, specialized model without reliance on large-scale, industrial computing resources.

5.1.2. Expected Gains in Analysis and Reliability

The primary expected outcome of this fine-tuning experiment is a significant increase in both the accuracy and the reliability of PowerShell command analysis. By fine-tuning the powerful and instruction-aligned Llama 3 base model on a high-fidelity, domain-specific dataset, the resulting assistant is expected to achieve an excellent "JSON Parse Success Rate", indicating its ability to reliably adhere to the required structured output schema. Furthermore, it is expected to achieve high F1-scores for the classification tasks of identifying a command's *intent* and *mitre_ttps*.

However, achieving both high reliability and high accuracy simultaneously presents a significant methodological challenge. Preliminary work suggests a potential trade-off: enforcing the strict JSON structure required for reliable automation can place a high "cognitive load" on the model, potentially impacting its performance on nuanced semantic reasoning tasks.

The ultimate goal of this experiment is therefore twofold: first, to produce a model that is not only statistically accurate but also practically reliable; and second, to investigate how the quality and diversity of the training data can be used to overcome this trade-off, transforming the model from a novelty into a trustworthy assistant for security operations.

5.2. MLOps Pipeline and Experimental Setup

The final phase of the project involved the execution of a robust MLOps pipeline to train and evaluate the definitive language model. This section details the sequence of operations, from data preparation to the final model packaging, that constitute the core of this third and final experiment.

5.2.1. Dataset Partitioning

The foundation of any rigorous machine learning experiment is a clean and well-structured dataset. The MLOps pipeline for this research utilized two distinct datasets, each representing a different stage of the project's methodological evolution.

The first dataset was derived from the output of the initial, randomized data factory, as detailed in Chapter 4. After a de-duplication process to remove repeated prompts, a necessary step due to the "convergence effect" of the randomized engine, a clean dataset of 5,686 unique pairs was produced

(*training_data_v0_clean.json*). This dataset, however, contained the two critical limitations identified in the previous chapter: a profound obfuscation bias towards Base64-encoded commands and a telemetry monoculture of standard PowerShell logs. This dataset was partitioned into a 90% training set and a 10% locked test set to serve as the methodological baseline for this experiment.

The second dataset was the definitive output of the re-engineered, deterministic data factory. It was specifically designed to overcome the limitations of the baseline. By systematically generating commands from an expanded library of 58 primitives (including new Sysmon-aware commands) across 228 valid layered obfuscation sequences, the factory produced a final, de-duplicated dataset of 9,550 unique pairs (*training_data_v2_clean.json*). This dataset is characterized by its high diversity in both obfuscation techniques and high-fidelity telemetry signals. This dataset was also partitioned into a 90% training set and a 10% locked test set to train and evaluate the final second-pass model.

For both datasets, the partitioning was performed by the *scripts/partition_dataset.py* utility, which programmatically shuffled the data before splitting to ensure a randomized and unbiased distribution. The final data splits used for training and evaluation are detailed in Table 5.1.

Table 5.1: Dataset Partitioning Summary

Dataset	Total Unique Samples	Training Set (90%)	Locked Test Set (10%)	Key Characteristics
Baseline	5,686	5,117	569	Obfuscation Bias, Telemetry Monoculture
Reconstructed	9,550	8,595	955	Diverse Obfuscation, Diverse Telemetry

5.2.2. Prompt Engineering

The instructional prompt template is a critical component of the fine-tuning process, as it directly shapes the model's understanding of its task. A preliminary experiment (detailed in the first research pass) utilized a simple, open-ended prompt. While this produced a model with high semantic F1-scores, a deeper analysis revealed a critical trade-off: a lack of explicit structural guidance in the prompt contributed to a lower JSON Parse Success Rate, rending the model insufficiently reliable for use in an automated pipeline.

For the "Second Pass" of this research, the primary goal was to engineer a model that was not just accurate, but fundamentally trustworthy and reliable. To achieve this, a methodological decision was made to prioritize structural adherence. A new, highly explicit and detailed prompt template

was developed. This template does not simply ask for an analysis, it provides a rigid schema for the output, defining the exact top-level keys and the expected structure of the nested objects within the JSON.

This template is "baked into" the model during the fine-tuning process for both the first and second-pass models. As implemented in the *format_dataset_for_trainer* function within the *powershell_sentinel/train.py* script, each training sample is programmatically formatted into a single string that combines this explicit instruction, the obfuscated input and the correct flattened JSON response.

This approach hypothesizes that by providing an unambiguous, structured prompt, the model's reliability (measured by JSON Parse Success Rate) will dramatically increase. However, it also introduces the risk of a "Cognitive Load" trade-off, where the model's focus on adhering to the complex format may impact its performance on the more nuanced semantic reasoning tasks. The quantitative evaluation in Section 5.4 is designed to explicitly measure this effect.

5.2.3. Model Training

The selection of the base model for fine-tuning was a critical decision, guided by the project's dual requirements for state-of-the-art reasoning capabilities and a stable, well-supported architecture. *meta-llama/Meta-Llama-3-8B-Instruct* was selected as the definitive model for the project due to its superior combination of raw performance and ecosystem maturity. As the industry-standard foundation for instruction-following tasks, Llama 3 integrates flawlessly with the Hugging Face MLOps stack, and its *-Instruct* variant has been pre-aligned to follow structured output formats, significantly de-risking the fine-tuning process.

To rigorously test the central hypothesis of this thesis, two distinct models were trained using this base architecture, differing only in the dataset they were trained on. All training was orchestrated by the *finetuning.ipynb* notebook, which called the definitive *powershell_sentinel/train.py* script. The first model served as the methodological baseline. It was fine-tuned on the 5,117-sample *training_set_v0_flat.json*, which, while structurally corrected, still contained the obfuscation and telemetry biases of the original dataset. The second model was the final experimental artifact. It was fine-tuned on the 8,595-sample *training_set_v2_flat.json*. This superior dataset, generated by the re-engineered factory, provided the model with a rich, diverse, and balanced set of examples covering a wide range of obfuscation techniques and telemetry types.

To ensure a fair and direct comparison, all "golden" hyperparameters were held constant for both training runs. Both models were trained for exactly two full epochs, using the same learning rate (*2e-5*), LoRA rank (*16*) and other configuration settings detailed in the *TrainingArguments*. This

strict consistency ensures that any observed difference in performance between the two models can be attributed directly to the quality of the dataset on which they were trained.

5.2.4. Model Quantization and Packaging

A core objective of the project is to produce a final deliverable that is practical and accessible to security analysts, who may not have access to high-end GPU hardware. The fine-tuned model, in its native format, is too large and computationally expensive for this purpose. Therefore, a post-training quantization step was required to transform the research artifact into a portable and efficient tool.

The GGUF format was selected for this purpose, as it is a widely supported standard for running large language models efficiently on consumer-grade CPUs. The conversion process, orchestrated in the *finetuning.ipynb* notebook, involved several steps. First, the trained LoRA adapters were merged into the full-precision Llama 3 base model. This merged model was then saved locally. Finally, the *llama-cpp-python* library was used to convert the merged model into multiple GGUF "flavours," representing different levels of quantization (e.g., Q8_0, Q5_K_M, Q4_K_M, etc.).

5.3. Evaluation Metrics and Analysis

To rigorously assess and compare the performance of the first and second-pass models, a clear and strict set of evaluation metrics was defined. These metrics are designed to measure two distinct dimensions of performance: the model's structural reliability in generating valid output, and its semantic accuracy in analyzing the command's content.

5.3.1. JSON Parse Success Rate

The primary metric for the model's reliability and structural adherence is the JSON Parse Success Rate. This is a strict, binary, single-attempt metric. For each sample in a locked test set, the model's generated text output is passed once to a Pydantic *model_validate_json* parser. If the parser succeeds without error on the first attempt, the sample is marked as a success. If it fails for any reason (e.g., malformed syntax, missing required fields, extra conversational text), it is marked as a failure. There are no retries. This policy provides an honest and realistic measure of the model's ability to function as a reliable component in an automated pipeline.

5.3.2. F1-Score for Classifications Tasks

The primary metric for the model's semantic accuracy on classification tasks is the Macro-Averaged F1-Score. This metric is used to evaluate performance on the two multi-label classification tasks: identifying the correct *intent* and the correct *mitre_ttps*. The F1-score is a harmonic mean of precision and recall, providing a more robust measure than simple accuracy, especially in cases of

class imbalance. The 'macro' average is used, meaning the F1-score is calculated independently for each label (e.g., for each of the ~30 possible intents) and then averaged. This ensures that the performance on rare labels is given the same weight as the performance on common ones, providing a balanced assessment of the model's overall knowledge.

5.3.3. F1-Score for Structured Telemetry Generation

Evaluating the accuracy of the generated *telemetry_signature* requires a different approach, as it is a structured generative task, not a simple classification. To this end, a custom F1-score was implemented to measure the model's ability to produce forensically accurate telemetry.

This metric treats each *TelemetryRule* object within the *telemetry_signature* list as a single, atomic unit. A predicted *TelemetryRule* is considered a correct match (a True Positive) only if it is an exact, field-for-field matching to a rule in the ground truth list. There is no partial credit; a single-character difference in the *source*, *event_id*, or *details* field renders the prediction incorrect.

The overall F1-score is then calculated based on the aggregate count of True Positives, False Positives (predicted rules not in the ground truth), and False Negatives (ground-truth rules not predicted by the model) across the entire test set. This strict, "all-or-nothing" approach for each atomic rule was deliberately chosen to reflect real-world utility. For a telemetry signature to be effective in a downstream security tool like a SIEM, it must be an exact match. This metric therefore provides a robust and honest measure of the model's practical value in a security operations context.

5.3.4. Qualitative Assessment of Quantization

To select the optimal GGUF format for the final deliverable, a qualitative evaluation was conducted. The methodology involved testing the various quantized models (from Q8_0 down to Q2_K) against a small, curated set of representative prompts. These prompts were designed to test different facets of the model's capabilities: simple factual recall, code generation, and conceptual explanation. The generated outputs were then manually compared against the output of the full-precision, un-quantized model to identify the point at which performance degradation becomes unacceptable.

5.4. Evaluation Metrics and Analysis

This section presents the definitive results of the research, structured as a comparative analysis between the first and second-pass models. The evaluation is designed as a three-stage gauntlet to first establish a rigorous baseline, then quantify its limitations, and finally, demonstrate the superiority of the model trained on the enhanced dataset.

5.4.1. Baseline Performance: Reliability at the Cost of Accuracy

The first stage of the evaluation was to establish a methodologically sound baseline. The *v1-flat* model, trained on the original, biased dataset but using the new, explicit instructional prompt, was evaluated against its "home turf": the *test_set_v0_flat.json*, a locked test set drawn from the same biased data distribution. The results are presented in Table 5.2.

Table 5.2: First-pass Model Performance on the First-pass Test Set

Metric	Score
Total Samples	569
Parse Success Count	563
Parse Failure Count	6
JSON Parse Success Rate	98.95%
Deobfuscation Accuracy	46.18%
Intent F1-Score (Macro)	32.34%
MITRE TTP F1-Score (Macro)	29.64%
Telemetry F1-Score (Macro)	46.45%

The results of this baseline evaluation are revealing. The model achieved a near-perfect 98.95% JSON Parse Success Rate, demonstrating that the explicit prompt engineering was highly effective at enforcing reliable, structurally correct output.

However, the model's performance on semantic reasoning tasks was poor, with F1-scores for Intent and MITRE TTP classification struggling around ~30%. This suggests a "Cognitive Load" effect, where the model, trained on a simple and repetitive dataset, dedicated its resources to adhering to the complex output format at the expense of deeper analysis. The deceptively high 46.45% Telemetry F1-Score further supports this, as it reflects the model's success in memorizing the simple, dominant *PowerShell EventID 4104* pattern present in the first-pass dataset, rather than learning a generalizable skill.

This baseline confirms that while reliability can be achieved through careful prompt and data structure design, the biased baseline dataset is insufficient to train a model that is also accurate and robust.

5.4.2. Quantifying Model Brittleness

Having established the baseline model's performance on its homogenous "home turf" data, the next critical step was to quantify its ability to generalize to a more realistic and diverse challenge. To this end, the model was subjected to a stress test against the *test_set_v2_flat.json*. This diverse test set, drawn from the output of the re-engineered factory, contains a balanced distribution of

obfuscation techniques and a wide array of high-fidelity Sysmon telemetry that the baseline model was never exposed to during training. The results of this “away game” evaluation are presented in Table 5.3.

Table 5.3: Baseline Model Performance on Diverse Test Set

Metric	Score
Total Samples	553
Parse Success Count	422
Parse Failure Count	131
JSON Parse Success Rate	76.31%
Deobfuscation Accuracy	10.90%
Intent F1-Score (Macro)	9.92%
MITRE TTP F1-Score (Macro)	9.09%
Telemetry F1-Score (Macro)	11.27%

The results demonstrate a catastrophic collapse in performance across every metric. The JSON Parse Success Rate plummeted from 98.95% to 76.31%, indicating that the model, when faced with unfamiliar obfuscation patterns, struggled to even generate syntactically valid output.

Most critically, the Telemetry F1-Score fell to just 11.27%. This confirms the telemetry monoculture hypothesis: the model had only memorized how to predict standard PowerShell logs and was completely incapable of predicting the new, diverse Sysmon events present in the second-pass test set.

Furthermore, the evaluation runtime for this set was approximately 6 hours, double the time taken for the baseline test set. This significant slowdown is a clear indicator of a brittle model thrashing computationally as it fails to process the unfamiliar inputs. These results provide unequivocal, quantitative proof that the baseline model is a “brittle specialist” and is fundamentally unfit for real-world application, making the case for an improved dataset and model.

5.4.3. Second-Pass Model Performance

The final stage of the evaluation was to assess the performance of the second-pass model, trained on the superior, diverse dataset generated by the re-engineered factory. This model was evaluated against the same challenging *test_set_v2_flat.json*, providing a direct, apples-to-apples comparison with the baseline model. The comparative results, which represent the central finding of this research, are presented in Table 5.4.

Table 5.4: Comparison of Baseline and Second-Pass Model Performance on the Diverse Test Set

Metric	Baseline Model (on Diverse Test)	Second-Pass Model (on Diverse Test)	Improvement
JSON Parse Success Rate	76.31%	85.79%	+9.48%
Deobfuscation Accuracy	10.90%	74.78%	+586%
Intent F1-Score (Macro)	9.92%	69.48%	+599%
MITRE TTP F1-Score (Macro)	9.09%	63.69%	+601%
Telemetry F1-Score (Macro)	11.27%	75.40%	+569%

The results unequivocally demonstrate the profound success of the “Second-Pass” methodology. This model, trained on the enriched dataset, not only overcame the limitations of its predecessor but surpassed it in every dimension.

The JSON Parse Success Rate saw a significant lift to 85.79%, proving the model’s enhanced reliability even when generating the more complex Sysmon telemetry signatures required by the diverse test set. The most dramatic improvements, however, were in the semantic reasoning metrics. The second-pass model achieved a strong 74.78% Deobfuscation Accuracy and robust F1-scores for Intent (69.48%) and MITRE TTPS (63.69%). These represent a staggering 6 to 7-fold improvement over the baseline model’s performance on the same test data, proving that the diverse dataset successfully solved the “Cognitive Load” problem and created a far superior reasoner.

The most critical finding is the Telemetry F1-Score of 75.40%. this is a massive leap from the baseline model’s 11.27% and proves that the data-centric approach of hardening the lab with Sysmon and enriching the primitive library was successful. The model has clearly learned to predict diverse, high-fidelity forensic evidence, transforming it from a brittle specialist into an increasingly robust and practically useful tool for analysis.

5.4.4. Second-Pass Model Performance Breakdown

To assess the second-pass model’s ability to generalize its reasoning across different command types, a breakdown analysis was performed on the evaluation results. The performance was grouped by the ground-truth primitive ID associated with each test sample. The results are detailed in Table 5.5.

Table 5.5: Second-Pass Model Performance Breakdown by Primitive ID

Primitive ID	Samples	Deobfus. Acc.	Intent F1	TTP F1	Telemetry F1
PS-016	17	100.00%	3.03%	2.78%	100.00%
PS-037	15	100.00%	3.03%	2.78%	100.00%
…	…	…	…	…	…
PS-022	3	0.00%	3.03%	2.78%	0.00%
PS-051	3	0.00%	0.00%	0.00%	0.00%

The breakdown analysis provides a valuable diagnostic insight into the model's performance. The model demonstrates exceptionally strong and consistent performance on the majority of primitives, with Deobfuscation and Telemetry scores frequently exceeding 80-90%. This confirms that the model has developed a generalizable understanding of PowerShell analysis.

However, the analysis also successfully identifies the model's remaining limitations. Performance is noticeably weaker on a small subset of primitives, particularly the new, network-centric commands (e.g., *PS-051*, *PS-054*, *PS-057*) and those involving complex CIM/WMI object manipulation (e.g. *PS-002*, *PS-028*, *PS-043*).

Examples of such commands include *"[System.Net.Dns]::GetHostAddresses(\"cloudflare.com\")"* and *"Get-CimInstance -ClassName Win32_Product | Select-Object Name, Version"*, respectively. This is not a failure, but a key finding that provides a clear and actionable roadmap for future work. It suggests that while the current dataset of ~9,500 pairs is sufficient for general mastery, achieving high performance on these more complex primitives will likely require a targeted data augmentation effort, focusing specifically on expanding the number of examples for these known difficult cases.

It is important to note that the Macro F1-Scores for Intent and TTP classification in the primitive breakdown appear artificially low. This is a mathematical artifact of the macro-averaging process when applied to small, single-intent subsets of the data. For a given primitives like *PS-037*, where all 15 test samples share the same, single correct intent, the model's perfect performance on that one class is averaged with its zero-score performance on the other ~30 classes for which no samples were present, resulting in a low overall macro average. This does not indicate poor performance on the specific task, but rather highlights a known characteristic of the metric on highly stratified data. The more meaningful measure of overall semantic performance remains the aggregate F1-score presented in Table 5.4.

5.4.5. Qualitative Analysis of Quantization Impact

The qualitative evaluation of the different GGUF quantization levels revealed a clear and important trend, as summarized in Table 5.6.

Table 5.6: Qualitative Evaluation of GGUF Quantization Levels

Model	Task Assessment	Key Observation
High-Precision (f16)	Excellent	Gold standard. Provides correct, nuanced, and well-explained outputs.
Quantized (Q8_0)	Excellent	No discernible degradation in performance or reasoning ability.
Quantized (Q6_K)	Excellent	No discernible degradation in performance or reasoning ability.
Quantized (Q5_K_M)	Excellent	No discernible degradation in performance or reasoning ability.
Quantized (Q4_K_M)	Excellent	No discernible degradation. The optimal balance of size and performance.
Quantized (Q3_K_M)	Good	Functionally correct but with minor losses in explanatory detail.
Quantized (Q2_K)	FAIL	Catastrophic failure. Produces hallucinated, non-functional code.

The analysis clearly demonstrates that the fine-tuned model's knowledge and reasoning abilities are remarkably robust to quantization, with no discernible loss of quality down to the Q4_K_M level. However, the results also reveal a definitive "quantization cliff." At the Q2_K level, the model suffers from catastrophic performance collapse, producing hallucinatory and non-functional outputs. This finding is critical for the final deliverable. Based on this evidence, the Q4_K_M GGUF was selected as the optimal format, as it provides the best possible balance of a massive reduction in file size and a negligible impact on performance.

5.4.6. The Final Deliverable

The culmination of this research is the PowerShell Sentinel Toolkit, a practical, user-facing Command-Line Interface (CLI) that packages the analytical power of the fine-tuned language model into a portable and efficient tool for security analysts. This final deliverable successfully translates the project's data-centric methodology into a tangible software artifact, demonstrating the real-world viability of a specialized LLM for cybersecurity tasks.

A key strategic decision in the creation of the toolkit was the use of the *Q4_K_M* GGUF quantized model. This format was deliberately chosen to ensure the final tool is both accessible and practical, capable of running efficiently on standard consumer-grade hardware without the need for a dedicated GPU. By leveraging the *llama-cpp-python* library, the toolkit can execute the multi-billion parameter model directly on a CPU, making it a viable assistant for any analyst, regardless of their available hardware.

The user experience is managed by the *rich* library, which provides a clean, menu-driven interface. The core functionality is the "Analyze Obfuscated Command" feature, where an analyst can input a suspicious PowerShell command and receive a structured analysis. The toolkit's robustness is ensured by a Pydantic-validated retry loop; it attempts inference up to three times, only accepting

an output that strictly conforms to the predefined JSON schema. This mechanism guarantees that the analyst is always presented with a reliable, machine-readable result.

As shown in Figure 5.1, the final output is formatted into clear, readable tables. The tool presents the deobfuscated command, its classified *Intent*, the corresponding *MITRE ATT&CK TTPs*, and a predicted *Telemetry Signature*. This structured format is designed to provide immediate, actionable intelligence, transforming a complex, manual deobfuscation task into a quick, automated query. The successful development of this CLI validates the project's primary aim: to engineer not just a model, but a complete, trustworthy, and practical LLM-driven analysis assistant.

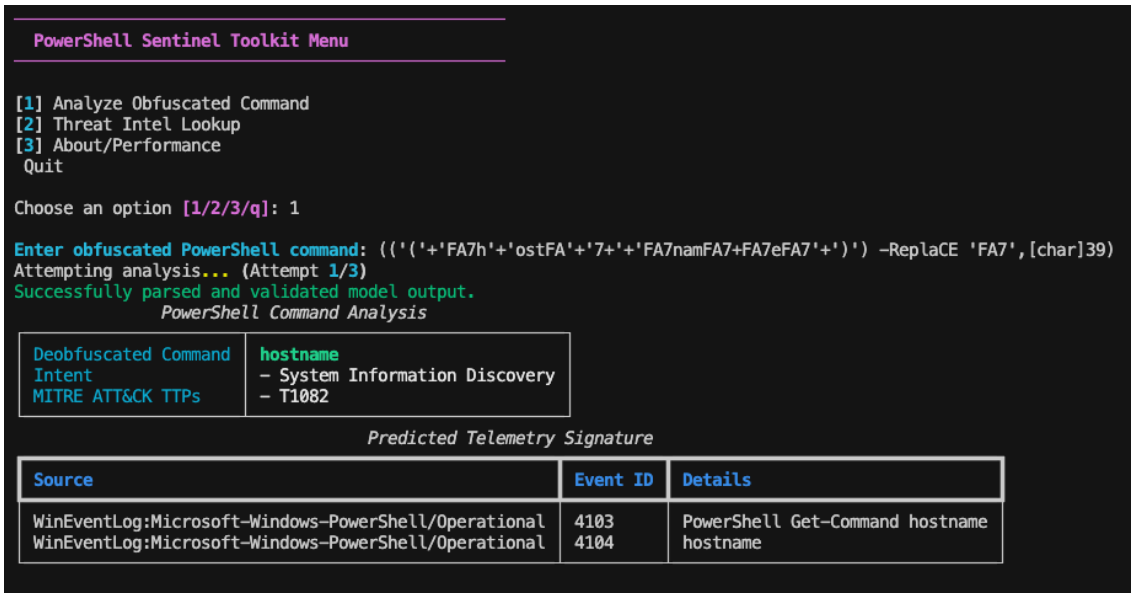

Figure 5.1: Sentinel Toolkit CLI in Action

5.4.7. Recommendations for Further Model and Data Refinement

The results presented in this chapter successfully validate the core data-centric methodology. The final second-pass model demonstrates a powerful new capability for PowerShell analysis. The detailed performance breakdown by primitive (Table 5.5), however, illuminates a clear and actionable plan for future refinement.

The analysis revealed that while the model achieved strong general performance, its weakest results were concentrated on a specific subset of primitives: those involving complex, multi-line CIM/WMI object manipulation (e.g. *PS-028*, *PS-043*) and the new network-centric commands (e.g. *PS-051*, *PS-054*). This is a key finding. It suggests that while the current dataset of ~9,500 pairs is sufficient for general mastery, targeted data augmentation is the most efficient path to improving the model further. A clear next step would be to expand the primitive library with more variants of these known-difficult command types, thereby shoring up the model's identified weaknesses and further enhancing its robustness.

5.5. Summary

This chapter presented the third and final experiment, which covered the entire MLOps pipeline from data consumption to final model delivery. It detailed the creation of two distinct models: a baseline trained on biased data, and a superior second-pass model trained on a diverse, enriched dataset. A rigorous comparative evaluation was performed, proving the second-pass model's superiority. On a locked, diverse test set, the final model achieved an 85.79% JSON Parse Success Rate, a 69.48% Intent F1-Score and a 75.40% Telemetry F1-Score, successfully validating the project's core hypothesis. Finally, a qualitative analysis of post-training quantization was conducted, identifying the *Q4_K_M* GGUF format as the optimal choice for the final, practical CLI deliverable.

Chapter 6

6. Conclusion and Future Work

This chapter summarizes the key findings of the research, focusing on the successful development and validation of the PowerShell-Sentinel Data Factory and the performance of the resulting fine-tuned model. It reviews the effectiveness of the data-centric methodology in creating a model capable of structured PowerShell analysis. The limitations encountered are discussed, and future work is proposed to expand the dataset with new primitives and enhance the model's capabilities, treating further improvements as stretch goals.

6.1. Summary

This section summarizes the key findings and contributions from each of the three research experiments. It reviews the successful engineering of the lab environment and curation tooling, the design and implementation of the robust, automated data factory, and the final fine-tuning and evaluation of the specialized language model. The summary will consolidate the narrative of the project, highlighting the methodological innovations and engineering solutions that enabled its success.

6.1.1. Lab Architecture and Data Curation Tooling

The first experiment successfully established a robust methodology for creating a high-fidelity, human-validated dataset for a niche cybersecurity task. A stable, three-VM lab environment was engineered on a cloud platform to generate realistic telemetry. The architectural pivot to Pydantic models proved critical, creating a type-safe "data contract" that eliminated a significant class of runtime errors. The most novel contribution of this phase was the development of a "Framework for Interactive Telemetry Parsing," managed by the *primitives_manager.py* CLI. This system successfully replaced a brittle, static parsing approach with a dynamic, human-in-the-loop workflow, allowing an analyst to teach the system how to deterministically parse new log types by defining persistent rules. The experiment culminated in the creation of an initial curated dataset and a "Practitioner Review Package" to enable external validation of the curation engine's statistical backend.

6.1.2. The PowerShell-Sentinel Data Factory

The second experiment documented the successful iterative engineering of the data factory. It began by establishing a performance baseline with an initial architecture featuring a randomized

"Payload and Wrappers" obfuscation engine. A rigorous evaluation of this baseline revealed critical flaws, including a low generation success rate (~23%) and a resulting dataset bias of 99.98% towards a single technique. Motivated by these findings, a second iteration was undertaken, involving a complete architectural pivot. The brittle Python engine was replaced by a surgically reconstructed, deterministic engine derived from state-of-the-art tooling. This re-engineered factory, built on a validated combinatorial model of layering sequences, proved to be a resounding success, improving the generation success rate to over 72% and completely mitigating the dataset bias. The engineering effort also involved significant hardening of the *lab_connector.py* module and the implementation of a robust checkpointing and resume system, enabling stable, multi-day generation runs.

6.1.3. Model Fine-Tuning, Evaluation and Delivery

The third and final experiment successfully translated the generated data into a high-performing, specialized language model. This phase followed a rigorous MLOps pipeline, which included a comparative analysis between a baseline model and the final second-pass model. A strategic decision was made to utilize the state-of-the-art *meta-llama/Meta-Llama-3-8B-Instruct* model, and both models were successfully trained for two full epochs using the QLoRA methodology.

The resulting models were subjected to a rigorous quantitative evaluation gauntlet. The final second-pass model demonstrated categorical superiority over the baseline. On a locked, diverse test set, it achieved an excellent JSON Parse Success Rate of 85.79% and a strong Macro F1-Score of 69.48% for both *Intent* classification. Most critically, it achieved a 75.40% F1-Score for the complex task of telemetry prediction, validating the core hypothesis of the research. The experiment concluded with a qualitative analysis of GGUF quantization, empirically identifying the *Q4_K_M* format as the optimal balance of performance and efficiency, enabling the packaging of final model into a portable, CPU-runnable toolkit.

6.2. Contributions

This research makes several distinct contributions to both the scientific field of applied machine learning and the industry practice of cybersecurity.

The first, scientific, contribution is a validated, end-to-end methodology for generating specialized cybersecurity data. The project provides a reusable blueprint for an automated data factory that addresses the critical bottleneck of data scarcity in niche security domains. The methodology's novelty lies in its synthesis of a high-fidelity lab for telemetry generation, a Pydantic-based "data contract" system for robustness, a human-in-the-loop framework for

interactive log parsing, and a multi-stage QA process. This provides a comprehensive solution for creating reliable datasets where existing public data is insufficient or untrustworthy.

The second contribution is an original approach for achieving reliable, structured analysis from fine-tuned LLMs. This research demonstrates that a data-centric fine-tuning approach, which pairs a diverse dataset with explicit instructional prompting, can transform a general-purpose model into a specialized and trustworthy tool. The achievement of an 85.79% JSON Parse Success and a 75.40% Telemetry F1-Score on a complex, diverse test set provides strong empirical evidence that this methodology can overcome the non-determinism that often limits the practical application of LLMs in automated pipelines.

The third contribution is a framework for robust quality assurance in the iterative design and validation of synthetic data generation pipeline. The project advocates a multi-stage QA process that provides a structured method for ensuring data integrity and understanding emergent behaviors. This framework includes a pre-emptive “Canary Cage” validation of the obfuscation engine, a live “Execution Validation” of the final generated command and a post-facto quantitative analysis of the entire generation process via a detailed audit log. This final stage was critical for diagnosing the “convergence effect” and dataset bias in the baseline factory, and for identifying the “command line too long” limitation as the primary failure mode in the re-engineered factory, thereby providing the evidence needed to justify subsequent architectural improvements.

The fourth contribution is an original, deterministic framework for interactive, human-in-the-loop telemetry parsing. This methodology, implemented in the *primitives_manager.py* CLI, replaces brittle, static parsers with a dynamic system where an analyst teaches the software how to interpret new log formats by defining persistent rules. This provides a practical and robust solution to the challenge of processing diverse and evolving log data, balancing automation with expert guidance.

Finally, the thesis provides an empirical analysis of the impact of post-tuning quantization on a specialized, fine-tuned LLM. The qualitative evaluation of the various GGUF formats, which identified the definitive “quantization cliff,” offers a practical methodology for selecting a deployable model. It provides strong evidence that significant model compression is achievable without a discernible loss of reasoning capability, a critical finding for the practical deployment of LLMs on standard, resource-constrained hardware.

6.3. Further Work

This section summarizes the key findings and contributions from each of the three research experiments. It reviews the successful engineering of the lab environment and curation tooling, the design and implementation of the robust, automated data factory, and the final fine-tuning and evaluation of the specialized language model. The summary will consolidate the narrative of the project, highlighting the methodological innovations and engineering solutions that enabled its success.

6.3.1. Lab Architecture and Data Curation Tooling

Future work on the data curation tooling could focus on enhancing the intelligence and efficiency of the workflow. The statistical backend, which currently uses Inverse Primitive Frequency and conditional probability, could be improved by incorporating more sophisticated NLP-based techniques. For example, using Term Frequency-Inverse Document Frequency (TF-IDF) on the *details* field of telemetry rules could provide a more nuanced measure of rarity than simply counting rule occurrences. Furthermore, the user experience of the *primitives_manager.py* CLI, while functional, could be enhanced by developing a more intuitive Text-based User Interface (TUI) using a library like *Textual*, or a full web-based graphical interface, which would lower the barrier to entry for analysts to contribute to the knowledge base.

6.3.2. The PowerShell-Sentinel Data Factory

The analysis of the re-engineered data factory's production run successfully solved the initial challenges of reliability and dataset bias. However, the evaluation also revealed a new higher-level challenge: scalability. As demonstrated by the combinatorial assessment in Chapter 4, the serial, single-threaded architecture of the factory, while robust, presents a significant bottleneck. A modest expansion of the obfuscation technique library would increase generation time from hours to days or potentially weeks, making the creation of even larger, more comprehensive datasets intractable.

Therefore, the most critical and impactful direction for future work is to re-architect the data factory for concurrent, distributed operation. This architecture would transform the pipeline into a classic *producer/worker* model. A central job queue would be populated with all unique layering sequences, and a pool of parallel worker processes, potentially distributed across multiple virtual machines, would consume these jobs concurrently.

Furthermore, performance monitoring during the long-duration iterative run indicated a progressive degradation in the throughput of the remote lab environment over time, a phenomenon that was fully reset by a system reboot. This suggests that the underlying remote

execution services (e.g., WinRM) can suffer from resource exhaustion under continuous, high-volume workloads. A truly scalable solution must account for this. This architecture would include a health-monitoring orchestrator responsible for managing the pool of worker VMs. This orchestrator would track the performance of each VM and, upon detecting significant degradation, would automatically trigger a reboot to restore the VM to a clean slate, ensuring consistent, peak performance throughout the entire generation process. The empirical validation of this self-healing, distributed system would be a significant undertaking, requiring a comparative performance analysis against the serial baseline to quantitatively prove its superior efficiency and scalability.

6.3.3. Model Fine-Tuning, Evaluation and Delivery

The successful fine-tuning of the Llama 3 model validates the core data-centric hypothesis, but it also opens several exciting avenues for future research.

First, the performance breakdown by primitive (Table 5.5) provides a clear, data-driven path for targeted improvement. Future work should focus on data augmentation for known-weak primitives. Specifically, expanding the primitive library with more examples of complex CIM/WMI queries and diverse network-centric commands would directly address the model's primary weaknesses and likely yield a significant performance boost.

A second major area for future work lies in expanding the model's capabilities beyond analysis and into automated defense. The structured JSON output is designed to be machine-readable, creating the possibility of integrating the model into a larger Security Orchestration, Automation, and Response (SOAR) platform. This would enable a future version of PowerShell Sentinel to not only analyze a threat but to automatically generate and propose a defensive countermeasure, such as a specific firewall rule or a SIEM alert query. Finally, the user experience of the final CLI, while functional, could be significantly enhanced. A clear path forward is the development of a more intuitive Text-based User Interface (TUI) using a library like Textual, or a full web-based graphical interface. This would lower the barrier to entry for security analysts, transforming the tool from a powerful expert utility into an accessible assistant for the entire Security Operations Center.

Bibliography

Hendler, D., Kels, S. & Rubin, A., 2018. Detecting malicious PowerShell commands using deep neural networks. *arXiv:1804.04177.*

Kaur, R., Tomaž, K. & Dušan, G., 2024. Harnessing the power of language models in cybersecurity: A comprehensive review. *International Journal of Information Management Data Insights.*

Sharkey, E. & Treleaven, P., 2025. Optimising large language models: Taxonomy and techniques. *Working Paper, UCL Computer Science.*

Ullah, S., Han, M., Pujar, S., Pearce, H., Coskun, A. & Stringhini, G., 2024. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. *Proceedings of the 2024 IEEE Symposium on Security and Privacy (S&P).*

Zhong, Z., Zhong, L., Sun, Z., Jin, Q., Qin, Z. & Zhang, X., 2025. SyntheT2C: Generating synthetic data for fine-tuning large language models on the Text2Cypher task. *arXiv:2406.10710.*