Perseus: An Automated Malware Deobfuscation Tool

Andrew Romans
Georgia Institute of Technology
aromans6@gatech.edu

***Abstract*—As society becomes increasingly interconnected through digital systems and virtually managed data, the risk of cybercriminals exploiting vulnerabilities to deploy malware and compromise victims continues to grow. To evade detection and hinder reverse engineering, cybercriminals employ a range of obfuscation techniques. Rapid malware analysis is critical, particularly during incident response, when timely action can prevent significant financial loss. However, due to increasingly sophisticated obfuscation methods, manual reverse engineering of malware is becoming increasingly complex. Current research has focused on one or two obfuscation techniques but has primarily used heuristic-based approaches to solve this problem. Here, I present Perseus, a tool that leverages large language models to identify patterns across three obfuscation techniques: Mixed-Boolean Arithmetic, Control Flow Flattening, and Virtualization. Using LoRA fine-tuning of models Qwen1.5B, Qwen7B, DeepSeek6.7B, and Codestral22B, Perseus attempts to provide semantically equivalent and easy-to-understand assembly from its obfuscated counterpart. Each model trains on obfuscated and clean assembly pairs derived from C code collected from popular GitHub repositories. Perseus demonstrates promising results in deobfuscating Mixed-Boolean Arithmetic and Control Flow Flattening. Virtualization, however, remains an open challenge with opportunities for future improvement.**

I. Introduction

The vast interconnected digital systems today have enabled the virtual management of critical infrastructure and personal data, whether on personal devices or through cloud-hosting services. As reliance on these systems grows, so does the number of individuals seeking to exploit their vulnerabilities. Cybercriminals frequently attempt to exploit these weaknesses to deploy malware and compromise victims.

Recent reports indicate that over 560,000 new malware samples are detected daily [1] and there has been a 26% increase in network-based malware [2]. Malware includes various forms, such as ransomware, adware, worms, and botnets. The rise in malware attacks has increased the need to understand emerging threats and develop effective defenses and detection techniques. Rapid analysis of malware is essential because timely action and understanding of malware on a compromised system can help prevent significant financial losses.

Manually reverse-engineering malware is a labor-intensive process that is becoming increasingly complex as malware authors employ more sophisticated obfuscation techniques to evade detection and obscure their objectives. These techniques can range from simple methods, such as XOR or base64 encoding, to more advanced approaches, such as Mixed Boolean Arithmetic (MBA) or Virtual Machine-based obfuscation.

Given the increasing complexity of obfuscation techniques and the advancing capabilities of large language models, the development of tools to assist reverse engineers in efficiently deobfuscating malware is a natural next step.

Prior work on deobfuscation has relied on dynamic trace analysis [3], API call deobfuscation [4], and guided binary methods [5], [6]. Machine learning techniques have been applied to malware classification [7]–[10] and NLP-based detection [11], [12], with datasets such as BODMAS [13] enabling large-scale studies. These approaches address related problems, but assembly-level deobfuscation across multiple obfuscation types remains largely unaddressed.

A. Perseus

In this paper, I present Perseus, an automated malware deobfuscation tool that uses LoRA fine-tuning and is trained on pairs of obfuscated and clean assembly code. Perseus takes obfuscated x86_64 assembly as input and produces semantically equivalent clean assembly. With many obfuscation techniques to choose from, I decided to focus Perseus's capabilities on three that pose unique challenges with minimal overlap: Mixed Boolean Arithmetic, Control Flow Flattening, and Virtual Machine Obfuscation (which I refer to as Virtualization in this paper). Perseus is designed to assist the reverse engineer in understanding difficult or complex obfuscated code by providing deobfuscated assembly, which, in turn, reduces the time required to understand and respond to malware.

In the following paper, I will discuss Perseus's system design, including the data pipeline and model architecture. I will discuss how I chose the training data, the training and evaluation results, and, lastly, share results on how Perseus performed in a real-world CTF challenge from the Flare-On reverse engineering competition.

B. Obfuscation

Mixed Boolean Arithmetic, MBA, obfuscation combines arithmetic and boolean operations with semantically equivalent but complex expressions [14]. For example, something as trivial as $x+y$ might be rewritten as $(x \oplus y)+2(x \& y)$. While at runtime this produces identical results, a reverse engineer looking at the assembly for a function might have a harder time understanding the function's goal. I chose MBA because it provides expression-level obfuscation compared to Control Flow Flattening and Virtualization. This means MBA does not alter the control flow or program structure. The obfuscation

is confined to the individual expressions and assignments. The code may look syntactically normal, but the *meaning* of each expression has been obscured. Similarly, MBA can be used to fill binaries with junk expressions that hide the true functionality, making it much harder to determine malware's true intentions. For Perseus, I focused on expressions that are simplifiable and chose to leave handling junk expressions for future efforts.

Control Flow Flattening (CFF) dismantles a program's control flow and replaces it with a state machine representation. All function blocks are extracted, assigned numerical state identifiers, and placed in a large switch statement controlled by a "dispatcher". At runtime, state variables are updated after a block executes, and the dispatcher determines which logic block to execute next. The state transitions, with semantically identical logic, make the original structure unrecognizable. I chose CFF for its graph-level obfuscation, where the execution shape changes, in contrast to MBA's expression-level obfuscation and virtualization's full instruction replacement. However, unlike Virtualization and MBA, the instructions found within blocks are left unchanged and remain readable to a reverse engineer, which can give analysts a false sense of confidence and accessibility. CFF scales aggressively with program size: as program blocks grow, the state space grows too, making the program increasingly difficult to reconstruct.

Figure 1 shows a concrete example of Control Flow Flattening applied to `strend`, a simple string-traversal function. The 16-instruction clean function expands to 40 instructions. The state variable at `[rbp−8]` is initialized to state 2, and every block ends by writing the next state and jumping back to the dispatcher at `+0x96`. The computed `notrack jmp rax` at the center of the dispatcher is the most disorienting feature: there is no single branch target a reverse engineer can follow, only a jump table resolved at runtime.

Finally, Virtualization is a full-abstraction obfuscation technique. Unlike MBA and CFF, which reorganize or transform existing code, Virtualization replaces the execution model entirely. Native instructions are replaced with bytecode for a custom instruction set architecture (ISA), a custom proprietary set of virtual instructions, executed by an embedded interpreter with no connection to the host architecture. While the program functions as expected, there are no shared semantics between the native code (x86) and the custom ISA, so a reverse engineer must learn the ISA and how it functions before being able to provide meaningful contributions to reversing its functionality. Of the three techniques, Virtualization is the hardest to deobfuscate.

Together, these three techniques provide unique challenges for Perseus to identify patterns and deobfuscate: MBA through expression and semantic-level obfuscation, CFF by dismantling the execution flow into a state machine, and Virtualization by replacing native instructions with an unknown, proprietary instruction set.

C. Parameter-efficient fine-tuning

Training a foundation large language model (LLM) from scratch requires substantial computing resources and extensive training data, which is beyond the scope of this paper. Fortunately, domain-adapted models with billions of parameters and comprehensive knowledge of C and C++ source code, as well as corresponding x86 assembly, are already available. Nevertheless, these models still require additional fine-tuning to perform specialized tasks such as deobfuscating MBA, CFF, and Virtualization. Parameter-efficient fine-tuning (PEFT) is a technique that addresses this challenge.

Perseus uses Low-Rank Adaptation (LoRA), a parameter-efficient fine-tuning technique that keeps the base model weights fixed and introduces pairs of small, trainable low-rank adapter matrices into the model's attention layers [15]. Traditional fine-tuning of LLMs requires updating all model parameters. LoRA, however, updates only these smaller parameter subsets during training, reducing the number of trainable parameters while maintaining the model's original capacity.

LoRA primarily operates by using two hyperparameters: the rank r, which controls the dimensionality of the adapter matrices, and thus the number of parameters being trained, and the scaling factor α, which controls the influence the adapter matrices have on the base model [16].

This approach is well-suited to Perseus, where the objective is to adapt large pretrained code models with up to 22 billion parameters to a specialized domain of deobfuscation on a single GPU. LoRA makes this objective achievable, given time and computing limitations. Furthermore, Perseus incorporates QLoRA, which applies 4-bit quantization to the frozen base weights, reducing memory consumption and allowing fine-tuning of models that would otherwise exceed GPU VRAM.

II. System Design

A. Data Pipeline

The Perseus data pipeline transforms raw C code into function-level obfuscated assembly, with its deobfuscated, or clean, counterpart for supervised fine-tuning. The pipeline consists of five stages: source collection, obfuscation, compilation, disassembly, and training pair construction.

Source Collection. C source files are collected from designated repositories. For this project, I used the AnghaBench repository, a benchmark suite containing 1 million compilable programs mined from the largest public C repositories on GitHub [17]. Additionally, I added a small set of custom handwritten C samples that could be used as a proof of concept. Each candidate file is passed through a filter that requires a minimum number of lines of code, specified in the config.yaml, and at least one control flow construct (i.e., `if`, `for`, `while`, `switch`). Files that pass the filter are tested for compilation with GCC; those that fail are discarded.

Obfuscation. Tigress [18], an obfuscation framework developed by the University of Arizona, processes each file three times, once for each obfuscation type. The tool offers many obfuscation techniques, including MBA, CFF, and

```
; strend -- CFF obfuscated (40 instructions)
+0x0:  endbr64
+0x4:  push   rbp
+0x5:  mov    rbp, rsp
+0x8:  mov    qword ptr [rbp - 0x18], rdi
+0xc:  mov    qword ptr [rbp - 8], 2      ; state = 2 (initial)
+0x14: cmp    qword ptr [rbp - 8], 6
+0x19: ja     +0x95                        ; dispatcher: bounds check
+0x1b: mov    rax, qword ptr [rbp - 8]
+0x1f: lea    rdx, [rax*4]
+0x27: lea    rax, [rip + 0xe3b]           ; jump table base
+0x2e: mov    eax, dword ptr [rdx + rax]
+0x31: cdqe
+0x33: lea    rdx, [rip + 0xe2f]
+0x3a: add    rax, rdx
+0x3d: notrack jmp rax                    ; computed dispatch
+0x40: mov    rax, qword ptr [rbp - 0x18] ; case 2: return s
+0x44: jmp    +0x9b
+0x46: add    qword ptr [rbp - 0x18], 1   ; case 1: s++
+0x4b: mov    qword ptr [rbp - 8], 6      ; state = 6
+0x53: jmp    +0x96                        ; -> dispatcher
+0x55: mov    rax, qword ptr [rbp - 0x18] ; case 5/6: return s
+0x59: jmp    +0x9b
+0x5b: mov    rax, qword ptr [rbp - 0x18] ; case 0: check *s
+0x5f: movzx  eax, byte ptr [rax]
+0x62: test   al, al
+0x64: je     +0x70
+0x66: mov    qword ptr [rbp - 8], 1      ; state = 1 (advance)
+0x6e: jmp    +0x96                        ; -> dispatcher
+0x70: mov    qword ptr [rbp - 8], 3      ; state = 3 (null found)
+0x78: jmp    +0x96                        ; -> dispatcher
+0x7a: cmp    qword ptr [rbp - 0x18], 0   ; case 4: null-ptr check
+0x7f: jne    +0x8b
+0x81: mov    qword ptr [rbp - 8], 4
+0x89: jmp    +0x96
+0x8b: mov    qword ptr [rbp - 8], 6
+0x93: jmp    +0x96
+0x95: nop                                ; dead state
+0x96: jmp    +0x14                        ; loop back to dispatcher
+0x9b: pop    rbp
+0x9c: ret
```

```
; strend -- clean (16 instructions)
+0x0:  endbr64
+0x4:  push   rbp
+0x5:  mov    rbp, rsp
+0x8:  mov    qword ptr [rbp - 8], rdi    ; s = arg
+0xc:  cmp    qword ptr [rbp - 8], 0
+0x11: jne    +0x1e                        ; if (!s) return s
+0x13: mov    rax, qword ptr [rbp - 8]
+0x17: jmp    +0x2d
+0x19: add    qword ptr [rbp - 8], 1      ; s++
+0x1e: mov    rax, qword ptr [rbp - 8]
+0x22: movzx  eax, byte ptr [rax]
+0x25: test   al, al                       ; while (*s)
+0x27: jne    +0x19
+0x29: mov    rax, qword ptr [rbp - 8]    ; return s
+0x2d: pop    rbp
+0x2e: ret
```

Fig. 1: CFF obfuscation of `strend` (left) vs. clean assembly (right).

Virtualization, and provides straightforward CLI commands for automation. Once a candidate passes this stage, there are four variants: clean, MBA, CFF, and Virtualized files. Tigress applies `EncodeArithmetic` for MBA, `Flatten` for CFF, and `Virtualize` for virtualization, targeting all functions in the file.

Compilation. All four variants of each source file are compiled to x86_64 ELF binaries using GCC with optimizations disabled (`-O0`) to preserve the structure of the obfuscated output.

Disassembly. During disassembly, the function boundaries are extracted from the ELF symbol table. Each function is disassembled independently using `objdump`, producing per-function assembly files. CRT boilerplate functions are excluded at this stage (i.e., `_start`, `frame_dummy`, etc).

Training Pair Construction. The last stage takes each obfuscated function and pairs it with its corresponding clean disassembly, forming a supervised training example. Pairs are formatted as tuples of instruction/input/output, and serialized to a JSONL file. These are then split into training, validation, and test sets for the inference pipeline.

B. Model architecture

Perseus requires base models with strong performance on code-generation benchmarks and knowledge of C, C++, and x86 assembly. It was also important to select models with a range of parameter values to evaluate how model scale affects deobfuscation performance. With these constraints, Perseus uses Qwen 1.5B, Qwen 7B, DeepSeek 6.7B, and Codestral 22B. Qwen [19], DeepSeek [20], and Codestral [21] are code LLMs that achieve state-of-the-art performance across various benchmark tests. During fine-tuning, Perseus applies QLoRA with rank $r = 16$, scaling factor $\alpha = 32$, and a dropout of 0.1. The rank-to-alpha ratio of 1:2 is a standard starting configuration that allowed the adapters to influence the base model meaningfully without overriding its pretrained representations. The adapters are injected into seven attention and feed-forward projection layers: `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. These layers are consistent across all four models because they all share the same transformer architecture.

C. Data Normalization

Raw disassembly cannot be ingested directly into a model. Absolute virtual addresses vary across compilation runs and reveal no structural information. Thus, Perseus applies two normalization passes: one during training and one at evaluation.

When preparing data for training, all instruction addresses are converted to function-relative offsets of the form `+0x{offset}`. For control-flow instructions (i.e. `call`, `jmp`, etc) operand addresses are resolved against a symbol map built from the binary's ELF symbol tables: `.symtab` and `.dynsym`. If the target falls within the function's boundaries, it is rewritten with a relative offset; otherwise, the raw address is preserved. This produces a position-independent representation that is consistent across samples compiled from different source files.

During evaluation, a second normalization pass is applied before computing metrics. DeepSeek-Coder was observed producing whitespace-collapsed output, omitting newlines and token spacing. This caused string comparisons to fail even

when the output assembly was semantically correct. Model output and ground-truth strings are therefore split on address boundaries, address prefixes are stripped, and all whitespace is removed before lowercasing, ensuring that formatting differences in model output no longer penalize semantically correct predictions.

D. Inference Pipeline

Before inference, each obfuscated assembly function is first packaged into a structured chat prompt using the model's native chat template. The prompt uses a system message and a user message. The system message is as follows:

> *You are a binary deobfuscation assistant. Given obfuscated x86-64 assembly code, you produce the equivalent clean, deobfuscated assembly. Preserve the function's semantics while removing obfuscation patterns such as MBA (mixed boolean-arithmetic), control flow flattening, and virtualization.*

The same system prompt is used for both fine-tuning and evaluation to ensure consistency between the training and deployment conditions. The user message appends the task instruction with the normalized obfuscated assembly. Generation uses low-temperature sampling ($T = 0.1$) and a maximum of 1024 new tokens, which keep the output deterministic and allow the model to recover from low-probability but correct token choices. Generated token sequences are decoded, and the input prompt is stripped, leaving only the model's predicted assembly.

III. Evaluation Metrics

In x86_64 assembly, each line is a single instruction consisting of an opcode and its operands; Perseus's predictions are analyzed line by line and compared to their deobfuscated counterpart. To measure performance during evaluation, Perseus uses two different metrics: Exact match and F1 score. During scoring, all instructions are weighted equally. Perseus does not assign higher importance to particular instructions such as branches or memory operations.

Exact Match. A prediction is an exact match if every instruction in the generated output matches its deobfuscated counterpart. Both blocks are normalized before comparison, so formatting variations do not affect the result. However, an exact match is a strict metric; a single incorrect or missing instruction counts as a failure. Exact match was most sensitive to instructions referencing memory offsets, where a single incorrect address caused the entire prediction to fail despite otherwise correct output.

F1 Score. Perseus uses F1 score, defined in Equation 1, where Precision and Recall are computed by positional alignment. Precision measures how many instructions Perseus deobfuscated correctly, and Recall measures how much of the expected instructions Perseus recovered.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{1}$$

the generated instruction at line i is compared against the expected instruction at line i. Deobfuscation is not as simple as binary classification; it involves assessing whether the model correctly recovers the original structure and obfuscated assembly functionality. It is important to know whether Perseus generated instructions that resemble the deobfuscated code but miss critical information (high precision, low recall) or recover some structure but generated incorrect and extra instructions (high recall, low precision). The F1 score captures both simultaneously.

A known scoring artifact for both metrics is RIP-relative addressing for accessing globals, string literals, or jump tables. Each instruction adds an offset from the instruction pointer, taking the form `[rip + 0xN]`. The offsets differ between the obfuscated and deobfuscated binaries even when the instruction is semantically correct. To avoid penalizing Perseus for this, a normalized variant (F1*) replaces all `[rip + 0xN]` operands with a placeholder before scoring. Results for both F1 and F1* are reported in Section IV.

IV. Experiments & Results

A. Proof of Concept

Prior to scaling training to the AnghaBench repository, a proof-of-concept test was conducted to validate that Perseus could learn deobfuscation using the data processing and ingestion approach on a small set of handwritten C samples. These samples included simple patterns such as arithmetic expressions, conditional branches, short loops, and combinations of these. Each sample was passed through the full data pipeline.

```
int fib(int n) {
    int a = 0, b = 1;
    for (int i = 0; i < n; i++) {
        int tmp = a + b;
        a = b;
        b = tmp;
    }
    return a;
}
```

```
int fib(int n) {
  int a = 0, b = 1, i = 0;
  while (((unsigned int)((i - n) ^
    ((i ^ n) & ((i - n) ^ i)))
    >> 31U) & 1) {
    int tmp = (a - ~b) - 1;
    a = b;
    b = tmp;
    i = ((i | 1) << 1) - (i ^ 1);
  }
  return a;
}
```

Fig. 2: Handwritten `fib` function (top) and its MBA-obfuscated equivalent (bottom) produced by Tigress.

Figure 3 shows the results of this test, where Qwen2.5-Coder-1.5B was used as the base model. The training results showed that the model overfitted after the fifth epoch, which was expected given the small sample size. These results confirmed that the pipeline functioned properly, the LoRA adapters were updating on meaningful signals, and the model

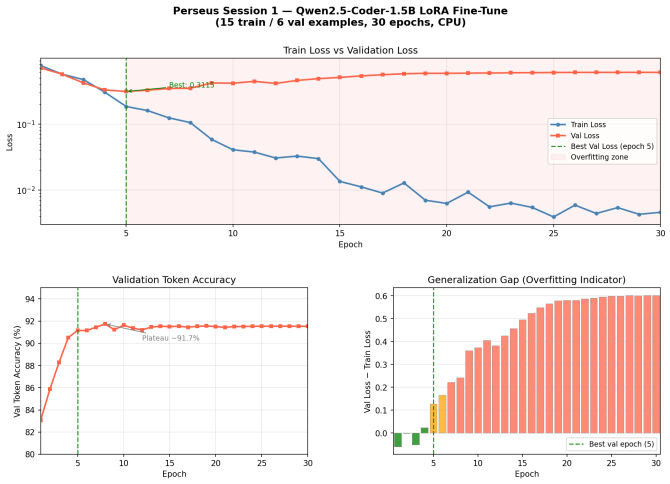

Fig. 3: Proof of Concept training session on handwritten samples.

could begin learning to deobfuscate real-world targets. This served as justification to scale to the AnghaBench dataset.

B. Model comparison

To evaluate how model scale affects deobufscation performance, all four models were trained on 5,000 samples from AnghaBench using identical hyperparameters: 12 epochs, early stopping with the top 3 checkpoints saved, and the same LoRA configurations. A total of 5,000 samples were chosen to balance training cost against coverage, as hardware and time constraints limited the number of training samples for larger models.[1]

Table 1 summarizes the performance on the held-out test set; samples Perseus had not seen during training. F1 is the weighted average across all three obfuscation types. F1* is the RIP normalized F1 score, where all `[rip + 0xN]` operands are replaced with a placeholder before scoring. This resolved the exact-match and line-level issues described in Section III.

Table 1: Model F1 scores on 5k AnghaBench samples.

Model	Exact Match	F1	F1*
Qwen2.5-Coder-1.5B	0.0%	0.261	0.290
Qwen2.5-Coder-7B	0.0%	0.275	0.314
DeepSeek-Coder-6.7B	0.0%	0.279	0.309
Codestral-22B	0.0%	**0.344**	**0.388**

Exact match is 0% across all models, as expected given the data scale. An exact match requires every instruction to be reproduced, and the functions Perseus trains on range from 10 to 30+ instructions. A single incorrect register or missed instruction counts as a failure. With 5,000 training samples, Perseus does not yet have sufficient exposure to fully generalize to functions it has never seen.

The F1 score is therefore the primary metric. Codestral-22B leads at an F1 score of `0.344`, with the three smaller models clustering between 0.26 and 0.28. Interestingly, the performance gap is not proportional to the parameter count. The 1.5B, 7B, and 6.7B models perform nearly identically, suggesting that the jump in parameter count from 7B to 22B is more significant than from 1.5B to 7B for this task. Virtualization significantly lowers the weighted average across all models, a trend examined in detail in Section IV-C.

Whether these scores represent success or failure depends on the intended use case. If the goal was to perform autonomous deobfuscation with no human in the loop, an average F1 of 0.344 would be inadequate. However, Perseus is designed as an analyst assistance tool, not a replacement for human judgment. The question is not whether Perseus is perfect, but whether Perseus can reduce the time and cognitive load required to understand the obfuscated code.

For example, manual deobfuscation of MBA-obfuscated assembly requires an analyst to reason through complex bitwise arithmetic instructions or dynamically debug the obfuscated assembly; both processes can take minutes to hours per function. Perseus can process a function in seconds. An MBA F1 of 0.510 means that, on average, over half the obfuscated instructions are correctly resolved before an analyst opens a disassembler. Even partially deobfuscating a function narrows the search space. The fault tolerance for an assistance tool is high, as every instruction Perseus resolves correctly is one fewer that the analyst must derive manually.

Consider Perseus's output for `strchr` function under MBA obfuscation (Codestral-22B, F1=0.426). By the metric, less than half the instructions match. Figure 4 shows the obfuscated input Perseus received alongside its generated output. The obfuscated version expands two simple equality checks into dense bitwise sequences of 10 to 12 instructions, with each using MBA identities. This grows the function from 21 instructions to 42 obfuscated instructions. Perseus collapses this back to 26 instructions.

Despite the score, the analyst reading Perseus's output would immediately see: args added to the stack, a byte fetched from an address each iteration, null-termination checks, character comparisons, pointer increments, and a loop back to the beginning. This is `strchr`, a string character search. This would be identifiable in seconds. An analyst manually analyzing the two MBA sequences would eventually reach the same conclusion, but would take considerably longer. This shows that even if Perseus's output isn't perfect, it can still be useful.

C. Obfuscation type breakdown

Table 2: F1 scores by obfuscation type (* = RIP normalized).

Model	MBA	MBA*	CFF	CFF*	Virt	Virt*
Qwen2.5-Coder-1.5B	0.447	0.463	0.302	0.382	0.061	0.061
Qwen2.5-Coder-7B	0.454	0.523	0.326	0.378	0.073	0.074
DeepSeek-Coder-6.7B	0.471	0.493	0.313	0.389	0.079	0.079
Codestral-22B	**0.510**	**0.539**	**0.505**	**0.619**	0.066	0.066

MBA. Of all three obfuscation types, Mixed Boolean Arithmetic is the most learnable obfuscation type across all models. This observation is unsurprising given the nature of

[1]Codestral-22B was trained on a cloud-hosted H100 PCIe 80GB GPU via RunPod. Checkpointing was vital to avoid losing progress across sessions.

```
; MBA: *p == c  (10 instructions)
+0xf:  mov   eax, dword ptr [rbp-0x1c]
+0x12: mov   byte ptr [rbp-0x1], al
+0x15: mov   rax, qword ptr [rbp-0x18]
+0x19: movzx eax, byte ptr [rax]
+0x1c: movsx edx, al
+0x1f: movsx eax, byte ptr [rbp-0x1]
+0x23: mov   ecx, edx
+0x25: sub   ecx, eax       ; c - *p
+0x27: movsx edx, byte ptr [rbp-0x1]
+0x2b: mov   rax, qword ptr [rbp-0x18]
+0x2f: movzx eax, byte ptr [rax]
+0x32: movsx eax, al
+0x35: sub   edx, eax       ; *p - c
+0x37: mov   eax, ecx
+0x39: or    eax, edx       ; (c-*p)|(p*-c)
+0x3b: test  eax, eax
+0x3d: js    +0x45          ; not equal?
; MBA: *p == '\0'  (9 instructions)
+0x45: movzx eax, byte ptr [rax]
+0x49: movsx eax, al
+0x4c: not   eax            ; ~*p
+0x51: mov   edx, eax
+0x53: movzx eax, byte ptr [rax]
+0x57: movsx eax, al
+0x5d: sub   eax, 0x1       ; *p - 1
+0x60: and   eax, edx       ; ~*p & (*p-1)
+0x62: test  eax, eax
+0x64: jns   +0x6d          ; not null?
```

```
; Perseus output (26 instructions)
+0x0:  endbr64
+0x4:  push  rbp
+0x5:  mov   rbp, rsp
+0x8:  mov   qword ptr [rbp - 8], rdi
+0xc:  mov   dword ptr [rbp - 0xc], esi
+0xf:  jmp   +0x29
+0x11: mov   rax, qword ptr [rbp - 8]
+0x15: movzx eax, byte ptr [rax]
+0x18: test  al, al           ; null check
+0x1a: je    +0x23
+0x1c: mov   eax, 0
+0x21: jmp   +0x45
+0x23: add   qword ptr [rbp - 8], 1  ; ++p
+0x28: nop
+0x29: mov   rax, qword ptr [rbp - 8]
+0x2d: movzx eax, byte ptr [rax]
+0x30: movsx eax, al
+0x33: sub   eax, dword ptr [rbp - 0xc]
+0x36: test  eax, eax         ; *p == c?
+0x38: je    +0x11
+0x3a: mov   rax, qword ptr [rbp - 8]
+0x3e: movzx eax, byte ptr [rax]
+0x41: test  al, al
+0x43: jne   +0x23
+0x45: pop   rbp
+0x46: ret
```

Fig. 4: `strchr` MBA obfuscated input (left, key sections) vs. Perseus output (right).

this obfuscation type, which replaces arithmetic and bitwise operations with more complex expressions. LLMs pretrained on C and assembly retain enough arithmetic reasoning to partially invert them. The scores range from 0.447 to 0.510, with consistent improvement under RIP normalization.

CFF. Control flow flattening is where model size has the greatest impact. Without the RIP normalization changes, Qwen and DeepSeek plateau around an F1 score of 0.31, while Codestral-22B reaches a score of 0.505. Codestral shows a 60% improvement and has the largest gap in model size across the three obfuscation types. With the RIP normalization adjustments, Codestral-22B reaches 0.619, indicating that a significant portion of errors involve position-dependent operands rather than structural misunderstanding. The performance gap is likely due to Tigress `Flatten`, which restructures a function's branching skeleton into a state machine. Reconstructing the original control flow requires reasoning about the entire function at once rather than instruction by instruction. Reasoning across an entire function's structure at once favors larger models, which explains why Codestral-22B outperforms smaller models.

A concrete example is `isp_pad_buffer_type`, where Codestral-22B achieved an F1 of 0.852. Figure 5 shows Perseus's output alongside the expected clean assembly. The model correctly recovered the function's branching structure and instruction sequence; the only deviations were a single incorrect jump offset (`jmp +0x4b` vs `jmp +0x51`) and three RIP relative address mismatches. Neither error was structural; the control flow skeleton, comparisons, and return paths were all correct. This is consistent with the RIP normalization improvement seen in Table 2: the gap between raw CFF F1 (0.505) and normalized CFF F1 (0.619) can be explained by position-dependent operand errors, not by failures to understand the state machine.

Virtualization. Virtualization deobfuscation fails across all models, with F1 scores between 0.061 and 0.079 and their RIP normalized scores being identical. The root cause is the input length. Tigress `Virtualize` replaces native x86_64 instructions with a custom bytecode interpreter, expanding functions by a factor of 3.5–7.9× as shown in Table 3. Clean functions in the test set ranged from 16 to 109 instructions, while their virtualized counterparts averaged over 600 and peaked at over 1,000. This far exceeded what the model could meaningfully process as flat instruction sequences with a 4,096-token context window. Perseus's predictions, regardless of the clean function's length, consistently generated approximately 48 instructions, producing a fixed-length approximation rather than true deobfuscation.

Table 3: Instruction expansion under Tigress `Virtualize`.

Sample	Clean (instrs)	Virtualized (instrs)
glxewInfo	71	273
layout	164	568
tiler	111	604
username	81	478
dev	86	502
separations	88	539
hash	102	673
get_1	88	625
psm	91	722
poll	155	940

A sliding window approach was considered to handle the length problem. However, because virtualization-obfuscated code has a fetch-decode-dispatch structure, slicing this into windows would destroy the structural context that makes the dispatcher identifiable. The model would only see fragments of handler code with no view of how they connect, which is the information needed to reconstruct and deobfuscate.

The logic for a Control Flow Graph (CFG) builder was implemented in `src/feature_selection.py`, which includes basic block extraction, edge classification, and graph-level features. However, this was not added to the training pipeline due to time constraints, but it represents the natural next direction. Providing Perseus with a CFG representation alongside raw disassembly would give it structural context about how instruction blocks connect. This information is

```
; Perseus output (27 instr, F1=0.852)
+0x0:  endbr64
+0x4:  push   rbp
+0x5:  mov    rbp, rsp
+0x8:  mov    qword ptr [rbp - 8], rdi
+0xc:  mov    dword ptr [rbp - 0xc], esi
+0xf:  mov    rax, qword ptr [rbp - 8]
+0x13: mov    eax, dword ptr [rax]
+0x15: cmp    dword ptr [rbp - 0xc], eax
+0x18: jl     +0x21
+0x1a: mov    eax, 0
+0x1f: jmp    +0x4b       ; <-- DIFF (+0x51 expected)
+0x21: mov    rax, qword ptr [rbp - 8]
+0x25: mov    rdx, qword ptr [rax + 8]
+0x29: mov    eax, dword ptr [rbp - 0xc]
+0x2c: cdqe
+0x2e: shl    rax, 2
+0x32: add    rax, rdx
+0x35: mov    edx, dword ptr [rax]
+0x37: mov    eax, [rip+0x2eb2]  ; <-- RIP
+0x3d: and    eax, edx
+0x3f: test   eax, eax
+0x41: je     +0x4b
+0x43: mov    eax, [rip+0x2e9e]  ; <-- RIP
+0x49: jmp    +0x51
+0x4b: mov    eax, [rip+0x2e9a]  ; <-- RIP
+0x51: pop    rbp
+0x52: ret
```

```
; Expected clean output (27 instructions)
+0x0:  endbr64
+0x4:  push   rbp
+0x5:  mov    rbp, rsp
+0x8:  mov    qword ptr [rbp - 8], rdi
+0xc:  mov    dword ptr [rbp - 0xc], esi
+0xf:  mov    rax, qword ptr [rbp - 8]
+0x13: mov    eax, dword ptr [rax]
+0x15: cmp    dword ptr [rbp - 0xc], eax
+0x18: jl     +0x21
+0x1a: mov    eax, 0
+0x1f: jmp    +0x51
+0x21: mov    rax, qword ptr [rbp - 8]
+0x25: mov    rdx, qword ptr [rax + 8]
+0x29: mov    eax, dword ptr [rbp - 0xc]
+0x2c: cdqe
+0x2e: shl    rax, 2
+0x32: add    rax, rdx
+0x35: mov    edx, dword ptr [rax]
+0x37: mov    eax, [rip+0x2eae]
+0x3d: and    eax, edx
+0x3f: test   eax, eax
+0x41: je     +0x4b
+0x43: mov    eax, [rip+0x2eaa]
+0x49: jmp    +0x51
+0x4b: mov    eax, [rip+0x2e9e]
+0x51: pop    rbp
+0x52: ret
```

Fig. 5: Perseus output (left) vs. expected (right) for `isp_pad_buffer_type` CFF (F1=0.852).

needed to recognize the dispatcher loop and reconstruct the original branching logic before generating clean assembly output.

New research confirms this direction. Pushan [22], published in March 2026, demonstrates that CFG-based approaches can successfully address virtualization deobfuscation, but instead of recovering clean x86_64 assembly, it decompiles to C pseudocode. Pushan also uses Tigress for evaluation and builds its approach around CFG recovery. Their results show that 988 of 1,000 Tigress-generated binaries were successfully deobfuscated. Though Perseus's attempt was unsuccessful, the shared insight between Perseus and Pushan is the same: the structure lives in the graph, not the linear trace.

D. Flare-On CTF Challenge

To evaluate Perseus on real-world obfuscated code outside the training distribution, it was applied to a function extracted from a Flare-On CTF binary, `FlareAuthenticator.exe`. Flare-On is Mandiant's annual reverse engineering capture-the-flag competition, widely considered one of the most technically demanding CTFs in the security community, and one of the few that consists solely of reverse engineering challenges. Challenges vary from malicious PDF files, Python games, complex cryptography, and, more importantly for Perseus, obfuscated binaries. They are custom-crafted challenges with purposeful obfuscation, making them a meaningful test of generalization.

The challenge chosen for this test is challenge 8 from Flare-On 12. The binary, `FlareAuthenticator.exe`, is a Windows application that prompts the user to enter a 25-character password. The user must enter it correctly to get the flag. Unfortunately, Perseus was trained on ELF files, not PE files, but in this scenario, we only need to target a single function within the binary. `FlareAuthenticator.exe` has many various obfuscation techniques, some of which Perseus was not trained on. However, the target function at RVA `0x176c2` is a key function for solving for the password

FlareAuthenticator MBA — Base vs Fine-tuned: Property Comparison
S = accumulated sum, P = derived product, Ground truth: [rax+0x78] = S + P

Model	Mode	Lines Out	Output Format	Simplified (< 18)?	Semantically Correct?
Qwen 7B	Zero-shot Base	7	Prose + Code	Yes	Partial
Qwen 7B	Perseus Fine-tuned	5	Clean ASM	Yes	Yes
Codestral 22B	Zero-shot Base	9	Prose + Code	Yes	No
Codestral 22B	Perseus Fine-tuned	5	Clean ASM	Yes	Yes

Fig. 6: Flare-On CTF results

and consists of 18 MBA-obfuscated instructions. The sequence encodes a simple memory update that loads a value, adds a second operand, and stores the result. To a human analyst, the 18-instruction sequence reveals no obvious arithmetic intent without manual simplification or dynamic analysis, which can be difficult for this particular binary.

Of all four fine-tuned models evaluated, only Qwen2.5-7B and Codestral-22B were successful. To compare the improvements from fine-tuning, the two base models were evaluated. Figure 6 summarizes the results.

The base model failures are instructive. Both Qwen2.5-7B base and Codestral-22B base respond with markdown prose and step-by-step explanations of the obfuscation, with embedded code blocks, rather than producing clean assembly output. The models understand that deobfuscation is being requested, but have not learned the output format or the algebraic reasoning required to simplify the sequence. Fine-tuning corrects both. Figure 7 shows the base model failure for Codestral-22B; the fine-tuned output is shown in Figure 8.

The other two fine-tuned models, Qwen2.5-1.5B and DeepSeek-Coder-6.7B, echo the obfuscated input unchanged while DeepSeek produces a whitespace-collapsed version. Both Qwen2.5-7B and Codestral-22B correctly reduce the 18-instruction sequence to 5 clean instructions.

```
; Codestral-22B BASE -- raw model output
The given assembly function 'accumulate_final_value' is
obfuscated using Mixed Boolean-Arithmetic (MBA) technique.
Here's the deobfuscated version of the function:

'''assembly
0x000176c2:  mov  rax, qword ptr [rbp + 0x678]
0x000176c9:  mov  r9,  qword ptr [rbp + 0x348]
0x000176d0:  mov  rdx, qword ptr [rax + 0x78]
0x000176d4:  mov  rcx, rdx
0x000176d7:  xor  rcx, r9          ; WRONG: rdx XOR r9
0x000176da:  add  rcx, rdx         ; + rdx
0x000176dd:  inc  rcx              ; + 1
0x000176e0:  or   rcx, rdx         ; | rdx  (not rdx+r9)
0x000176e3:  and  rdx, r9
0x000176e6:  sub  rcx, rdx
0x000176e9:  add  rcx, rdx
0x000176ec:  mov  qword ptr [rax + 0x78], rcx
'''
The deobfuscated code performs the following operations:
1. Loads [rbp+0x678] into rax ... [explanation continues]
```

Fig. 7: Codestral-22B base model output on the Flare-On MBA sequence.

```
; Obfuscated input (18 instructions)
0x176c2:  mov  rax, [rbp + 0x678]
0x176c9:  mov  r9,  [rbp + 0x348]
0x176d0:  mov  rdx, [rax + 0x78]
0x176d4:  mov  rcx, r9
0x176d7:  not  rcx
0x176da:  mov  r8,  rdx
0x176dd:  not  r8
0x176e0:  or   r8,  rcx
0x176e3:  mov  rcx, rdx
0x176e6:  add  rcx, r9
0x176e9:  lea  r8,  [r8 + rcx + 1]
0x176ee:  or   rdx, r9
0x176f1:  sub  rcx, rdx
0x176f4:  mov  rdx, rcx
0x176f7:  or   rdx, r8
0x176fa:  and  rcx, r8
0x176fd:  add  rcx, rdx
0x17700:  mov  [rax + 0x78], rcx
```

```
; Qwen2.5-7B output (5 instructions)
0x176c2:  mov  rax, [rbp + 0x678]
0x176c9:  mov  r9,  [rbp + 0x348]
0x176d0:  mov  rdx, [rax + 0x78]
0x176d4:  add  rdx, r9
0x176d7:  mov  [rax + 0x78], rdx

; Codestral-22B output (5 instructions)
0x176c2:  mov  rax, [rbp + 0x678]
0x176c9:  mov  rdx, [rbp + 0x348]
0x176d0:  mov  rcx, [rax + 0x78]
0x176d4:  add  rcx, rdx
0x176d7:  mov  [rax + 0x78], rcx
```

Fig. 8: Flare-On MBA CTF function: obfuscated input (top) and fine-tuned model outputs (bottom).

Both outputs correctly capture the function's original intent: `[rax + 0x78] += r9`. The register allocation is the only difference between the two models: Qwen2.5-7B uses `rdx` while Codestral-22B uses `rcx`. That said, both are correct and can be interpreted by an analyst. Neither model was trained on data from this challenge. The result demonstrates that fine-tuning MBA patterns from AnghaBench samples transfers to real-world obfuscation, providing evidence that Perseus generalizes beyond its training configuration.

V. Limitations

Though Perseus demonstrates meaningful performance in deobfuscating MBA and CFF, several limitations should be noted.

Virtualization. As discussed in Section IV-C, the most significant weakness of Perseus is Virtualization-obfuscation, which largely failed across all models. The core issue is that Tigress `Virtualize` expands functions by 3.5–7.9× the number of instructions, producing inputs that exceed the model's token length and context window. Perseus would need to take a slightly different approach, implementing the unused CFG code to recover the obfuscated binary structure before attempting to recover the original assembly instructions.

Tigress. All training data is generated by the Tigress obfuscation tool. Real-world malware commonly uses different toolchains such as OLLVM, Themida, VMProtect, and custom packers. The output of these tools may differ significantly from Tigress-generated code. The Flare-On result provided some evidence that MBA patterns transfer across obfuscators, but broader validation is needed across other toolchains.

Evaluation Metric. Line-level F1 score measures structural similarity, but not necessarily semantic equivalence. Two assembly sequences can be semantically equivalent but differ in register allocation or instruction ordering, which may result in a lower F1 score. A thorough evaluation process could include compiling outputs and comparing their execution traces.

Failure Scenarios. Perseus degrades significantly when obfuscation is surrounded by complex structure. For example, in the MBA obfuscated `print_tabs` function found in Figure 9, Perseus produced a wrong stack frame, lost an argument register, and failed to deobfuscate the MBA obfuscated loop counter; resulting in an F1 score of 0.13. This suggests the model has not learned to handle MBA that appears inside bodies of logic or functions with many local variables.

Context Window. Perseus was fine-tuned with a maximum sequence length of 4,096 tokens. Very long functions, particularly virtualized ones, exceeded this limit and are truncated before inference, thus potentially losing vital information. No segmentation or sliding-window strategy was implemented because those techniques would lose meaningful context required for deobfuscation.

Single-function scope. Perseus processes one function at a time with no visibility into the surrounding code or overall binary functionality. Obfuscation patterns that encode information across functions cannot be resolved from a single function's disassembly.

VI. Future Work

Perseus's results on MBA and CFF suggest that automated deobfuscation using fine-tuning is a viable approach, but it is far from a solved problem. Several directions could advance Perseus toward a practical, analyst-facing tool.

Semantic Evaluation Metric. Though the F1 score was decent at measuring performance on the surface, more information could have been gathered about Perseus's performance by compiling outputs and analyzing execution compared to the original. A more thorough evaluation approach may provide more insight into whether Perseus is producing semantically correct assembly that may differ structurally. This distinction

```
; print_tabs -- EXPECTED (key lines)
+0x8:  sub    rsp, 0x30               ; correct frame
+0xc:  mov    qword ptr [rbp - 0x18], rdi
+0x10: mov    qword ptr [rbp - 0x20], rsi
+0x14: mov    qword ptr [rbp - 0x28], rdx  ; rdx at -0x28
+0x18: mov    qword ptr [rbp - 0x30], rcx  ; rcx at -0x30
       ...
+0xa8: add    qword ptr [rbp - 8], 1   ; i++ (1 instruction)
```

```
; print_tabs -- GENERATED (key failures)
+0x8:  sub    rsp, 0x20          ; WRONG: should be 0x30
+0xc:  mov    qword ptr [rbp - 0x18], rdi
+0x10: mov    qword ptr [rbp - 0x20], rsi
+0x14: mov    qword ptr [rbp - 0x10], rdx
+0x18: mov    qword ptr [rbp - 0x10], rcx ; WRONG: aliases rdx slot
       ...
+0xb1: add    rax, rax           ; MBA not deobfuscated:
+0xb4: or     rax, 2             ;   i++ expressed as
+0xb8: mov    rdx, rax           ;   9-instruction MBA
+0xbb: mov    rax, qword ptr [rbp - 8]
+0xbf: xor    rax, 1
+0xc3: mov    rcx, rax
+0xc6: mov    rax, rdx
+0xc9: sub    rax, rcx
+0xcc: mov    qword ptr [rbp - 8], rax
+0xd0: jmp    +0x104             ; truncated
```

Fig. 9: `print_tabs` MBA failure (F1=0.130, Codestral-22B). Expected output (left) vs. Perseus output (right).

matters for an analyst assistance tool: correct output across different registers is still useful, but incorrect output that looks structurally correct is not.

More Training Data. Perseus was only trained on 5,000 samples pulled from AnghaBench across all models and showed decent results. The next step would be to test whether Perseus's performance improves when trained on larger amounts of data, as well as data from different sources, such as real-world malware samples.

CFG Integration. Perseus implements a CFG builder in `src/feature_selection.py`, but it is not incorporated into the training pipeline. Providing Perseus with graph-level information, in addition to raw assembly, would reveal how blocks are connected [22]. This is the next path to take that may enable Perseus to successfully deobfuscate virtualization-obfuscated samples.

Other Obfuscators. Extending training data to use commonly found obfuscators such as OLLVM, Themida, or VMProtect would test whether this approach generalizes beyond Tigress and improve performance on real-world malware.

Ghidra/IDA plugin. This tool was designed to be used alongside an analyst's workflow, its natural direction would be to develop a plugin for the common tools used by a reverse engineer, such as IDA, Ghidra, or Binary Ninja. A function could be deobfuscated without leaving the disassembler.

More Obfuscation Techniques. Due to scope limitations, Perseus focused on MBA, CFF, and Virtualization. However, there are plenty of other techniques that malware authors use to evade detection or delay reverse engineers from analyzing their software, such as dummy code injection, opaque predicates, or string encryption [23].

VII. Conclusion

As our world grows increasingly interconnected, so does the number of cybercriminals looking to exploit online weaknesses to deploy malware and compromise victims. As malware obfuscation grows more complex, the need for a tool to assist a reverse engineer rises. Perseus is a tool that assists analysts by automating malware deobfuscation through LoRA fine-tuning. With the Codestral-22B model, Perseus resulted in F1 scores of 0.510 for MBA and 0.505 for CFF obfuscations. More importantly, the Flare-On result demonstrates that Perseus generalizes beyond its training data. Base models produced markdown prose and failed to simplify the sequence, while fine-tuned models correctly collapsed 18 MBA-obfuscated instructions from a real CTF binary into 5 clean instructions, with no training on that binary or the obfuscator used. Though the results are promising, key limitations prevent Perseus from being an analyst-ready tool, including the inability to deobfuscate virtualization-obfuscated binaries and the lack of support for commonly used obfuscators used by active malware. Perseus is a step toward a practical, analyst-facing deobfuscation tool that demonstrates the approach is viable and provides a clear path forward. The source code is available at https://github.com/aromans/Perseus.

References

[1] Huntress, "Malware statistics," accessed 2024. [Online]. Available: https://www.huntress.com/malware-guide/malware-statistics

[2] WatchGuard Technologies, "Internet security report Q2 2025," 2025, accessed 2025. [Online]. Available: https://www.watchguard.com/wgrd-resource-center/security-report-q2-2025

[3] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 674–691.

[4] V. Kotov and M. Wojnowicz, "Towards generic deobfuscation of Windows API calls," in *Proceedings of the Workshop on Binary Analysis Research (BAR) 2018*, 2018.

[5] Y. Zhao, Z. Tang, G. Ye, X. Gong, and D. Fang, "Input-output example-guided data deobfuscation on binary," *Security and Communication Networks*, vol. 2021, p. Article 4646048, 2021.

[6] R. Tofighi-Shirazi, I. M. Asăvoae, P. Elbaz-Vincent, and T.-H. Le, "Defeating opaque predicates statically through machine learning and binary analysis," in *Proceedings of the 3rd ACM Workshop on Software Protection (SPRO '19)*. Association for Computing Machinery, 2019, pp. 1–12.

[7] M. Al Kadri, M. Nassar, and H. Safa, "Transfer learning for malware multi-classification," in *Proceedings of the 23rd International Database Applications & Engineering Symposium (IDEAS '19)*. Association for Computing Machinery, 2019, pp. 1–7.

[8] B. Bokolo, R. Jinad, and Q. Liu, "A comparison study to detect malware using deep learning and machine learning techniques," in *2023 6th International Conference on Big Data and Artificial Intelligence (BDAI)*. IEEE, 2023, pp. 1–6.

[9] V. Saini, R. Gupta, and N. Soni, "OpCode-based malware classification using machine learning and deep learning techniques," *arXiv preprint arXiv:2504.13408*, 2025.

[10] K. Aryal, M. Gupta, and M. Abdelsalam, "A survey on adversarial attacks for malware analysis," *arXiv preprint arXiv:2111.08223*, 2021.

[11] H. Liu, S. McCune, Q. D. Tran, F. Di Troia, and Y. Park, "Context-aware natural language processing for malware detection," in *2025 Silicon Valley Cybersecurity Conference (SVCC)*. IEEE, 2025, pp. 1–9.

[12] Q. D. Tran, J. Lim, and F. Di Troia, "An advanced malware detection system based on NLP to generate genetic markers," in *2024 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2024, pp. 1–6.
[13] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "BODMAS: An open dataset for learning based temporal analysis of PE malware." [Online]. Available: https://whyisyoung.github.io/BODMAS/
[14] Plzin, "Mixed Boolean-arithmetic," 2024, accessed 2024. [Online]. Available: https://plzin.github.io/posts/mba
[15] 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," *arXiv preprint arXiv:2106.09685*, 2021.
[16] T. Dettmers, A. Pagnoni, A. Fanous, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," *arXiv preprint arXiv:2305.14314*, 2023.
[17] B. C. F. da Silva *et al.*, "AnghaBench: A suite with one million compilable C benchmarks," 2021. [Online]. Available: https://github.com/brenocfg/AnghaBench
[18] University of Arizona, "The tigress C diversifier/obfuscator," accessed 2024. [Online]. Available: https://tigress.wtf/
[19] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu *et al.*, "Qwen2.5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.
[20] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi *et al.*, "DeepSeek-Coder: When the large language model meets programming — the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.
[21] Mistral AI, "Codestral: Hello, world!" 2024, accessed 2024. [Online]. Available: https://mistral.ai/news/codestral
[22] A. Sudhir, Z. L. Basque, W. Gibbs, A. P. Bajaj, P. S. Singaria, M. Zakocs, J. Hu, M. Schloegel, T. Bao, A. Doupe, Y. Shoshitaishvili, and R. Wang, "Pushan: Trace-free deobfuscation of virtualization-obfuscated binaries," *arXiv preprint arXiv:2603.18355*, 2026.
[23] PreEmptive Solutions, "Code obfuscation techniques," 2024, accessed 2024. [Online]. Available: https://www.preemptive.com/blog/code-obfuscation-techniques/