Disassembling Obfuscated Executables with LLM

Huanyao Rong*, Yue Duan†, Hang Zhang*, XiaoFeng Wang*, Hongbo Chen*, Shengchen Duan†, Shen Wang†

*Indiana University Bloomington
†Singapore Management University

Abstract—Disassembly is a challenging task, particularly for obfuscated executables containing junk bytes, which is designed to induce disassembly errors. Existing solutions rely on heuristics or leverage machine learning techniques, but only achieve limited successes. Fundamentally, such obfuscation cannot be defeated without in-depth understanding of the binary executable's semantics, which is made possible by the emergence of large language models (LLMs). In this paper, we present DISASLLM, a novel LLM-driven dissembler to overcome the challenge in analyzing obfuscated executables. DISASLLM consists of two components: an LLM-based classifier that determines whether an instruction in an assembly code snippet is correctly decoded, and a disassembly strategy that leverages this model to disassemble obfuscated executables end-to-end. We evaluated DISASLLM on a set of heavily obfuscated executables, which is shown to significantly outperform other state-of-the-art disassembly solutions.

I. INTRODUCTION

Disassembly, a process of translating machine-readable bytes into human-readable assembly instructions, is a critical step of binary reverse engineering, enabling important downstream tasks such as decompilation [25], [18], binary rewriting [32], [42], [71] and vulnerability detection [63], [17]. One major challenge in disassembly is correct identification of instruction boundaries within a byte sequence: due to the mixed storage of program code and data, or variable-length instructions such as those of x86, the locations where instructions start are often hard to determine, leading to disassembly errors not only in the instructions at the boundaries but also in the follow-up ones. This challenge has been exploited by various obfuscation techniques, such as opaque predicate, call stack tampering, signal-based obfuscation [58], to confuse a disassembler, which utilize junk bytes to increase the difficulty in delimiting binary sequences. To address this problem, existing techniques leverage symbolic execution [19], [63], heuristics [54], [32], [51], [44], [69], or machine learning [57], [79], [43]. However, their effectiveness has been found to be rather limited.

A. Limitations of Existing Work

Symbolic execution-based approaches [63] remove junk binaries within infeasible paths of a program to help determine instruction start addresses through analyzing the program's control and data flows. Their applications to binary deobfuscation, however, are hindered by path- and state-explosions as well as the presence of hard-to-solve path conditions – weaknesses exploited by adversarial techniques [76]. A variation of the approaches, concolic execution [19], analyzes a single execution trace of a binary executable instead of its whole program, and therefore is more scalable, but only covers the portion of the binary sequence related to the execution trace.

Some prior research utilizes manually constructed, domain specific heuristics to decide the correct instruction boundaries [54], [32], [51], [44], [69]. These heuristics are usually based upon shallow syntactical features, such as unusual opcode of the instructions recovered from wrong start addresses. Such approaches tend to be less robust since they rely heavily on empirical observations and assumptions, which could be easily broken, either by circumvention or due to the situations they cannot fit into.

Recently, machine learning (ML) has shown promise in addressing this problem. An ML-based approach automatically learns the features of valid and invalid instruction sequences to identify correctly recovered instructions. For this purpose, an early work [43] relies on feature engineering and decision trees. These techniques, however, are found to be inadequate for comprehensive and accurate detection of instruction boundaries. A more recent solution – XDA [57] trains a transformer [70] to analyze individual bytes, predicting whether each of them is the start of an instruction. However, since these bytes are not decoded first, the model must learn the decoding by interpreting the raw bytes directly. This limits the model's capacity to learn the semantic meaning of the machine code, as it first has to learn the decoding process. Another state-of-the-art work is DeepDi [79], which employs superset disassembly [20] to decode all possible addresses, before instruction validity can be determined with a graph neural network (GNN) on the instruction flow graph. However, due to the limited semantic and contextual learning capability of GNN, it has been demonstrated that DeepDi is vulnerable to dedicated adversary obfuscation techniques, which drastically drop its F1 score from 91.64% to 16.67% [80]. A primary reason for their limitation is their restricted model size, which further limits their capability to understand the complex semantics of decoded assembly instructions. One might suggest increasing the model size and the training dataset; however, expanding their dataset to trillions of bytes and their parameter size to billions, similar to current state-of-the-art large language models, is simply impractical. Therefore, a transfer learning approach that fine-tunes the current large language model is a more promising solution by comparison. Finally, all aforementioned ML methods completely exclude the more traditional but widely-used and time-tested disassembly approaches (e.g., recursive disassembling algorithm), which unnecessarily lose some benefits. For example, traditional disassembly provides the insight that if one instruction is valid, its following instructions are also likely to be valid, while any overlapping instructions are likely to be invalid. However, these methods do not take advantage of this insight.

B. Our Method

Our observation is that an experienced human expert usually can effectively recognize instruction boundaries: disassembled instructions at incorrect offsets tend to deviate significantly from their contexts, either at the syntax level or at the semantic level, and can therefore be captured by human experts. Existing approaches, whether heuristic-based or ML-powered, are not designed to consider semantic information in valid instruction identification and have limited capability in doing so. The recent advancement of generative large language models (LLMs), however, brings unprecedented opportunities to find a much more effective solution, as LLMs are meant to predict missing tokens from their contexts, and have demonstrated a remarkable comprehension ability closer to that of human analysts than all existing techniques [70], [28], [13], [68], [24], [61]. One straightforward solution is thus to let LLM inspect the whole byte sequence and decide the instruction boundaries (similar to XDA [57]), or to enumerate all possible disassembly results with different offsets and then rely on LLM to pick the correct ones (similar to DeepDi [79]).

However, such a direct application of the LLM does not fully leverage its capabilities for the following reasons. (1) Presenting machine code in bytes or in superset disassembly does not convey their semantic information in the most effective form: the LLM may struggle to understand the raw bytes directly, or may become overwhelmed by the redundant information in superset disassembly. (2) Such a naive approach can be quite expensive: it requires inspecting many potential addresses, resulting in a large number of queries to an LLM, which incurs significant delays or monetary costs. (3) Current decoder LLMs are designed for text generation tasks. However, the task of identifying instruction validity is more akin to a classification task, which is generally handled by encoder models. (4) Current LLMs are not trained on a large corpus of valid and invalid assembly text, which limits their direct capability to identify such validity.

To address these problems, our insight is that disassembly errors due to incorrect instruction boundaries are not prevalent (e.g., due to the performance cost, usually only some important code snippets will be obfuscated). In a typical reverse engineering session, a traditional disassembler (though not resilient to obfuscation) can already recover most valid instructions, while human analysts only need to fix errors in limited regions. Therefore, our idea is to integrate the LLM into a traditional disassembler - such a hybrid system mimics the human expert's reverse engineering process automatically and efficiently. Based on this idea, we propose DISASLLM, a systematic disassembly approach resilient to obfuscation, built upon an LLM-based classifier capable of identifying the validity of decoded instructions. DISASLLM consists of two components: an LLM-based validity classifier that determines whether instructions are correctly decoded and a disassembly strategy that leverages this classifier to disassemble obfuscated executables end-to-end efficiently and effectively.

1) LLM-based Validity Classifier: To train this model, we fine-tuned Llama 3 8B [15] using the method proposed by LLM2Vec [21], which converts a decoder model into an encoder model for NLP understanding tasks, including the token classification we require. This method allows us to address the problem (3) mentioned previously.

To prepare the unsupervised text dataset for the fine-tuning, we first compiled source functions from AnghaBench [27] and collected all instruction boundary addresses as the ground truth. We then disassembled these compiled binaries according to the ground truth into assembly text, including validly decoded instructions and invalid ones in the comments. This allows the model to learn the patterns of both valid and invalid instructions. Additionally, to further train a classifier, we gather the supervised dataset by disassembling the compiled AnghaBench binaries using our disassembly strategy (introduced soon). However, we replace the classifier that should result from this training task with a classifier based on the ground truth of the instruction addresses, ensuring the classifier's results are always correct. The dataset is then obtained by recording all outcomes of these classifications. These training process allows us to solve problem (4).

2) Our Disassembly Strategy: Given an obfuscated binary executable, DISASLLM disassembles its code memory region (i.e., the .text section) using a combination of linear disassembly and recursive disassembly to obtain an initial result. Based on this result, DISASLLM then verifies the validity of each decoded instruction from the initial disassembly and tries to fix the incorrect disassembly if applicable. To do this both efficiently and effectively, instead of checking all decoded instructions simultaneously or each instruction individually, DISASLLM uses a pre-filter to examine N adjacent instructions at once to confirm the validity of instructions with very high or low probability values. Based on the pre-filter results, DISASLLM then checks the remaining instructions one by one. Such optimized design aims to address the problem (2) mentioned previously.

To best represent the semantic information for an LLM to check validity, in order to verify the validity of one or more instructions, DISASLLM extracts all related instructions in the disassembly result with a breadth-first search algorithm, such as those with reference relations. Additionally, since instruction fixing and checking are conducted interchangeably, related instructions that are already fixed can aid in identification. The method addresses the problem (1) of directly applying LLM.

3) Evaluation Results: We evaluated DISASLLM on a set of obfuscated executables with heavily inserted junk bytes. Compared to the state-of-the-art approach, DISASLLM achieved an approximate 40% improvement in detecting the first instruction after the junk byte region. We also present the relevant metrics of our classifier model, both when used in conjunction with the disassembly strategy and when used independently, demonstrating its effectiveness in identifying valid instructions.

4) Contributions: We have proposed DISASLLM and made the following contributions to the field of disassembly:

- We trained an LLM-based classifier model to identify the validity of instructions within a snippet of decoded assembly code by treating it as a token classification problem.

- Building upon the validity identification capability of the model, we propose a disassembly strategy that leverages the model to disassemble obfuscated binary executables.

- We evaluated DISASLLM on a benchmark based on OLLVM containing a set of obfuscated executables. The results show that it significantly outperforms other disassembler counterparts.

II. BACKGROUND

A. Binary Disassembly

Binary disassembly translates the raw bytes into human-readable assembly instructions, serving as the vital first step of binary reverse engineering. We discuss some common disassembly techniques.

Linear and recursive disassembly. Linear disassembly sequentially processes each instruction. After decoding one instruction, the disassembling continues from the next byte which is treated as the start of the subsequent instruction. This strategy exploits the continuity of compiler code generation. Recursive disassembly, on the other hand, processes the instructions based on observable control flows. For example, after decoding a jump instruction with a constant target address, the disassembler assumes that the target address starts a new instruction and continues disassembling from there. These two strategies could be combined to achieve the optimal results [54], where the code "holes" not disassembled by the recursive disassembly are handled by linear disassembly (e.g., used by Angr [63]).

However, these traditional approaches can be problematic when data or junk bytes present (e.g., at the next byte or the jump target address), whose decoding leads to invalid instructions. Even worse, the addresses of invalid instructions can overlap with the valid ones, causing the latter to be disregarded. Nonetheless, one interesting phenomenon of this traditional disassembly process is that even if one invalid starting address is disassembled, a correct starting address will eventually be encountered after several wrongly decoded instructions (e.g., imagine a 7-byte sequence containing 1 leading junk byte and 2 subsequent valid instructions, while it may be wrongly disassembled into 3 invalid instructions, starting from the 8th byte the disassembly will return to the normal). This phenomenon is referred to as the "self-repairing" [48], increasing the disassembly stability but also making the disassembly errors more stealthy.

Disassembly based-on valid instruction identification. To overcome the weakness of traditional disassembly, learning-based or heuristic-based strategies are proposed to decide the validity of the disassembled instruction sequences and the correct instruction boundaries, based on specific features. However, as described in §I-A, current approaches show limited effectiveness in defeating obfuscation.

B. Large Language Model

Based on vast amounts of training data, the large language model (LLM) [70] marks a significant advancement in natural language processing (NLP) and program code understanding [61], [24]. LLMs can be categorized into encoder models and decoder models. Encoder models primarily focus on understanding the text, while decoder models are mainly designed for text generation.

Token classification [11], which assigns labels to tokens in a text, is one task of encoder models. For example, named entity recognition labels a text entity with predefined categories such as person names, organizations, locations, and dates. This labeling takes into account both the entity and its contexts.

A recent work, LLM2Vec [21], shows that a decoder model can be converted into an encoder model through fine-tuning. The technique starts with enabling bidirectional attention of the transformer so that the preceding tokens can attend to the following tokens in the self-attention layer. However, since the generative decoder model is not trained with such bidirectional attention, the effectiveness may drop for such change. To address this, the LLM is fine-tuned in an unsupervised manner using a corpus of texts as the dataset, just like the pre-training step, but instead of training to predict the next token given the preceding tokens, LLM2Vec leverages the technique known as masked next token prediction (MNTP), which trains to predict the randomly masked tokens given the unmasked ones in a training token sequence, a training approach commonly used by encoder model [28]. With the fine-tuned model, the embedding vector of one or more tokens under the text context can be computed, which can then be used to train another classifier model for the token classification task. The encoder model enables more effective use in token classification tasks, as demonstrated in their results [21]. In the following sections, we will describe how to utilize this technique to construct our validity classifier model.

III. MOTIVATION

In this section, we exemplify the limitations of state-of-the-art disassemblers in detail and demonstrate how LLMs can help, which motivates our work.

A. Limitations of Existing Work

DeepDi [79] is the state-of-the-art disassembler resilient to obfuscation due to its superset disassembling and GNN-based valid instruction sequence identification. However, obfuscation designed to thwart disassembly can still drastically reduce its effectiveness. To show this, we compile a simple C program containing all C syntactic features and obfuscate the resulting binary by (1) flattening the control flow [45] with OLLVM [41], and (2) inserting the junk bytes between basic blocks. This obfuscation puts junk bytes in unreachable code regions, testing the disassembler's instruction boundary identification ability. DeepDi's performance on this binary is summarized in Table I. We define the false negative (FN) as "valid instructions not shown in the disassembly result", while false positive (FP) as "invalid instructions overlapping with the valid ones shown in the disassembly result", as non-overlapping invalid instructions do not affect the disassembly of valid ones. Precision, recall, and the F1 score are defined accordingly. The "All" row considers all instructions in the binary when computing the above metrics, while the "Junk" row only includes the first valid instructions following the inserted junk bytes in the statistical scope, because they are most heavily affected by the junk bytes, how well they can be disassembled directly reflects the disassembler's anti-obfuscation ability. As seen in Table I, though DeepDi achieves good metrics in the "All" row, its performance significantly reduces in the "Junk" row. This suggests that DeepDi can successfully disassemble

TABLE I: DeepDi on the small obfuscated binary.

	Precision	Recall	F1
All	0.94	0.92	0.93
Junk	0.57	0.47	0.52

Precision = TP / (TP + FP), Recall = TP / (TP + FN)
F1 = 2 * Precision * Recall / (Precision + Recall)

most valid instructions whose boundaries are not impacted by the junk bytes, but falls short when handling the obfuscated regions.

B. LLM's Potential

A key observation is that invalid instructions decoded at the wrong offsets exhibit a very different appearance compared to valid instructions within the context. Thus, an experienced human reverse engineer can usually discern invalid instructions from valid ones easily. Thanks to the self-repairing feature (§II-A), even with junk bytes, traditional disassembly algorithms can still recover many valid instructions, albeit mixed with invalid ones. Therefore, in principle, a human expert can inspect these disassembly results to identify and correct the invalid instructions. However, this process requires substantial manual effort and is time-consuming, especially for large binaries.

Our intuition is to replace human experts with an LLM, given its outstanding capabilities of comprehending semantic information in both natural languages [13] and programming languages [24], including assembly code [38]. The advancements of LLMs provide a unique opportunity to develop a system resembling the human-involved disassembling process, while fully automatic and efficient.

IV. PROBLEM STATEMENT

DISASLLM aims to disassemble executables obfuscated to thwart the disassembly process. For example, junk bytes are inserted in the code region of such an executable, causing the linear disassembly approach to decode invalid instructions that may even overlap with valid ones. Additionally, opaque predicates can cause a conditional jump, which would never actually occur, leading to the recursive disassembly to decode the unreachable branch target and potentially overlapping with valid instructions. To sum up, DISASLLM is robust enough to handle almost all types of obfuscation designed to defeat disassembly, such as opaque predicate, call stack tampering, signal-based obfuscation [58], control flow flattening [45], that are combined with junk byte insertion.

However, DISASLLM does not aim to address self-modifying code or packing. In these scenarios, an analyst should first apply other approaches [60], [22] to extract the recovered code bytes before using our system. Additionally, DISASLLM is focused solely on disassembling the executable rather than other downstream deobfuscation tasks (e.g., recovering the original program from VM-based obfuscation [78]).

V. THE DISASSEMBLY STRATEGY

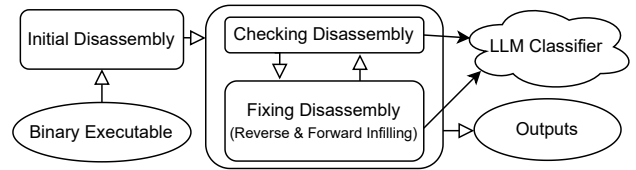Figure 1 illustrates the overview of DISASLLM. Given a binary executable, DISASLLM first performs an initial

Figure 1: Workflow of DISASLLM. The hollow arrow denotes the sequential procedure. The solid arrow represents the invocation with information returned.

disassembly on its code region (§V-A). Then, our system checks the validity of each decoded instruction (§V-B), during which it also attempts to fix disassembly errors when certain conditions are met (§V-C). Both procedures invoke the LLM-based classifier to check the validity of one or more instructions by representing the decoded instructions as text (§V-E). To enhance the efficiency of using LLM, DISASLLM employs a batching mechanism to exploit the parallel capabilities of GPUs (§V-D). Once all decoded instructions are checked and corrected, DISASLLM outputs the final disassembly results.

A. Initial Disassembly

Given the raw bytes of the code region of a binary executable, DISASLLM disassembles it using a combination of linear disassembly and recursive disassembly. The algorithm is essentially the same as recursive disassembly, but for each branch instruction, such as `jmp` and `ret`, the next instruction starting at its end address is also considered a candidate. Additionally, we employ a more aggressive recursive approach that tracks immediate numbers appearing in non-branch instructions, similar to [32].

The main reason for relying on the traditional approach instead of the more aggressive superset disassembly [20], which decodes every possible address, is two-fold. First, our linear plus recursive disassembly approach can produce sufficient results as a starting point more efficiently than super disassembly. This is due to the self-repairing feature of the traditional disassembly approach - even when an executable is heavily obfuscated with junk bytes inserted in the code region, our disassembly result still contains many validly decoded instructions. Second, as demonstrated later in Section §VII-C, the relevant instructions from traditional disassembly are sufficient to identify an instruction's validity correctly. In contrast, superset disassembly provides excessive redundant information that could interfere with performance.

After the initial disassembly, DISASLLM constructs a graph where the vertices represent basic blocks (sequences of decoded instructions), and the edges represent references between these blocks. We refer to this graph as the *disassembly graph*. The references include not only control flow relations but also references caused by immediate numbers due to a more aggressive recursive disassembly algorithm. Additionally, since DISASLLM also incorporates linear disassembly, the block resulting from the continued decoding of a non-conditional branch instruction creates a special edge. This edge connects the source block containing the branch instruction to the block starting at the end address of the branch instruction.

Only the first instruction of each block can be referenced. Therefore, during the disassembly process, if an instruction in the middle of a block is referenced, the block must be split into two blocks, with an edge connecting them. Such edges allow the breadth-first search algorithm to find related instructions when constructing the snippet for the classification.

B. Checking Disassembly Results

With the disassembly graph obtained from §V-A, DIS-ASLLM performs overlap minimization (described later in §V-E2). Following this, DISASLLM verifies the validity of each decoded instruction. Given a piece of decoded assembly instructions in text form, our task is to classify the validity of one or more instructions within this snippet. We consider this task a classical token classification problem, such as named entity recognition, which aims to classify which entity each word in a sentence belongs to. These tasks are highly similar in the sense that we can consider the instruction as the word and the validity as the entity.

We can consider two naive approaches to classify each instruction disassembled from §V-A. 1) We can represent all blocks in the entire disassembly graph as texts for the classification of each instruction. However, for large binaries consisting of thousands of instructions, the text will exceed the maximum token length supported by the model. 2) Instead of classifying all instructions individually, we can classify each instruction individually. This approach, in principle, can be more effective than the previous one because identifying a subsequent instruction can leverage the preceding validity results of other instructions, providing richer contextual information. However, this method can be too slow for large binaries.

Therefore, we propose a trade-off between these two methods. Firstly, DISASLLM checks $N = 16$ multiple adjacent instructions simultaneously to conduct a prefilter on the disassembled instructions. We obtain the probability of each instruction being valid by obtaining the sigmoid [33] value from the classifier. In the prefilter stage, DISASLLM empirically sets the validity of instructions with a probability $p > 0.95$ as valid and $p < 0.05$ as invalid. According to our evaluation in §VII-B, these instructions already constitute most of the total number of instructions in the disassembly graph. Additionally, DISASLLM deletes the blocks in which all instructions are invalid, removing most invalid overlapped blocks. After block deletion, overlap minimization is further performed to reduce the overlapped regions in the new disassembly graph. After the prefilter stage, we check the remaining instructions one by one. This allows DISASLLM to exploit the validity information of any related instructions classified during the prefilter stage.

To examine the validity of instructions, presenting only the instructions themselves is insufficient; their related instructions are also required as input to provide contextual information. These related instructions are obtained by conducting a breadth-first search starting from the instructions to be examined. To create the graph for the BFS task, we consider the disassembly graph to be a graph with instructions as vertices. In this graph, an instruction within a block is connected to the next instruction in the block, and the edges that connect blocks are now used to connect the instructions. When performing BFS, DISASLLM traverses both forward

and reverse edges, ensuring that given an instruction I, both the instructions referenced by I and the instructions that reference I are traversed. The BFS stops when the number of related instructions reaches a limit, which we have empirically set to 32, as this should provide sufficient contextual information for identifying validity. DISASLLM then extracts the blocks containing instructions visited by BFS. These blocks can either be the original blocks from the disassembly graph if all instructions within the blocks are visited or split blocks if only part of the instructions are visited. These blocks are represented as texts for token classification to check validity.

C. Fixing Disassembly Results

Note that not all authentic instructions are decoded in the disassembly graph; instead, these instructions can overlap with invalid ones in the graph. Therefore, DISASLLM must correct the disassembly results obtained from §V-A rather than merely classifying the validity of each instruction.

To achieve this, during the validity checking described above, DISASLLM keeps track of the validity of the memory regions. A valid region is defined as a region occupied by instructions already identified as valid. In contrast, an invalid region is a region that does not contain any parts of valid or unidentified instructions (i.e., it only contains invalid instructions). Other regions are considered unidentified. When an invalid region is produced between two valid regions, DISASLLM attempts to find the correct disassembly within the invalid region. All invalid instructions within this region are removed from the disassembly graph. DISASLLM first attempts reverse infilling to find the valid instruction sequence that ends at the start address of the valid region. Next, DISASLLM conducts forward infilling to find any missing blocks in the invalid region.

1) Reverse Infilling: The re-synchronization from invalid instructions in the invalid region to valid instructions in the valid region may be attributed to the self-repairing feature of the disassembly process mentioned earlier. This also suggests that there might be a hidden instruction sequence ending at the start address of the valid region. The reverse infilling process aims to find this sequence.

To find all possible instructions that end at a particular address a, we must decode all possible addresses from $a - n$ to $a - 1$, where $n = 15$ is the maximum instruction length in the x86 architecture, and keep only the instructions that end at a. Addresses that have already been identified as invalid instructions are not included. The process can be continued recursively on the start addresses of these newly found instructions. The results can be considered as a tree, where the root node is a, and each child node represents an address decoded into an instruction ending at the address in its parent node. We denote this process as reverse disassembly.

Therefore, given the start address of a valid region, DIS-ASLLM conducts a BFS on this tree to obtain a maximum of N instructions, where $N = 16$ is the maximum number of instructions identified at once in the prefilter step. DISASLLM then obtains the related instructions for these N instructions using the previously mentioned method, gets the probability of each instruction being valid, and retains only those with $p > 0.95$, similar to the process in the prefilter step. By

identifying a valid instruction sequence before the valid region, the start address of the valid region can be extended, and this process is continued recursively at the new start address.

Once no new valid instruction is identified, the reverse infilling switches to single instruction mode. In this mode, instead of performing reverse disassembly recursively, DISAS-LLM only finds all instructions ending at address a, where a is the start address of the latest valid region. The validity of these instructions is classified similarly to obtain probability values. If none of these values exceed 0.5, DISASLLM stops the reverse infilling process. Otherwise, it selects the instruction with the highest probability as the valid one and recursively continues the single instruction mode with the new start address of the valid region.

2) Forward Infilling: After completing reverse infilling on an invalid region between two valid ones, DISASLLM begins forward infilling. This step aims to find small valid blocks within the invalid region that are too small to be resynchronized with the number of instructions required by the self-repairing (e.g., containing fewer than three instructions).

To start forward infilling, DISASLLM disassembles one block at the first non-invalid address in the invalid region and identifies the validity of each instruction in the block. The process also works in two stages: a prefilter that confirms the validity of highly certain instructions and a single instruction checker that examines the remaining instructions one by one, ensuring all instructions are identified. If any valid instructions are detected, the process continues from the end of the last valid instruction; otherwise, it continues from the next non-invalid address. The process stops when the end of the invalid region is reached.

3) Example: To illustrate the effect of disassembly fixing, we use a simplified example snippet extracted from a real obfuscated binary, as shown in Listing 1. On the left, we present the initial disassembly results that have already been checked using the method described in §V-B. The instructions in the green frame are identified as valid, designating the corresponding area as a valid region. Conversely, the red frame marks the invalid region containing invalid instructions. Since the invalid region lies between two valid regions, DISASLLM attempts to fix the disassembly within the invalid region. The right part illustrates the same regions after the instructions in the invalid region have been fixed, assuming the classifier model functions correctly. The lower blue rectangle marks the instruction sequence obtained from reverse infilling, while the upper blue rectangle marks the short block recovered from forward infilling.

D. Batching

To further improve the efficiency of DISASLLM, we exploit the parallel capabilities of GPUs by using the batching mechanism supported by the transformer library [73]. For each of the processes described above, classification tasks are not executed immediately. Instead, they are pushed into a queue with the data required to post-process the results after classification. When the number of items in the queue reaches M, the maximum batch size, DISASLLM pops out the batch for classification and then post-processes the result probabilities using the corresponding data.

Listing 1 An example of disassembly fixing.

```
mov DWORD PTR [rbp-0x8], eax
mov DWORD PTR [rbp-0x4], 0x8
jmp 0x401007

mov edi, eax
rex.RB cld
(bad)
add BYTE PTR [rax], al
add bl, ch
(bad)
lea edi, ds:0x402000

mov al, 0x0
call 0x401000
```

```
mov DWORD PTR [rbp-0x8], eax
mov DWORD PTR [rbp-0x4], 0x8
jmp 0x401007

db 0x89
mov DWORD PTR [rbp-0x4], 0x17
jmp 0x401007

db 0xA9
lea rdi, ds:0x402000

mov al, 0x0
call 0x401000
```

E. Representing Blocks of Instructions

Given (split) blocks extracted from the disassembly graph using the previously mentioned method, we must represent them as text to leverage LLM for the code understanding task. The blocks should first be sorted in ascending order according to their start addresses so that when presented as text, they also follow the same order. Representing a block is a straightforward process of displaying all its instructions in Intel syntax. Additionally, for each instruction with extra information, a comment is appended after the instruction. For example, if the instruction is already known to be valid or invalid, a comment ; valid or ; invalid can be appended. However, unlike assembly code typically written by programmers, these blocks have some special features that need to be tackled.

1) Handling Gap: The extracted blocks can be noncontinuous; in other words, in the sorted list of blocks B, the start address of block $B[i + 1]$ is larger than that of its preceding block $B[i]$. In such cases, we must indicate this gap in the text to inform the LLM. To achieve this, we mark each continuous region's start and end addresses consisting of one or more adjacent blocks preceding and following the instructions. Examples of the text representation of these regions can be found in Figure 2. The start address is marked like a label, while the end address is marked in a comment. For example, the region 0x401000−0x401006 consists of two adjacent blocks with ranges 0x401000−0x401004 and 0x401004−0x401006. Moreover, if a block in the middle of the region is referenced by any instruction in the list of blocks (e.g., jmp), its start address should also be marked.

2) Handling Overlap: Another possible cause is that blocks can overlap with each other: in the sorted list of blocks B, the start address of $B[i + 1]$ is smaller than the end address of $B[i]$. This can frequently occur if the binary is obfuscated, such as when an opaque predicate causes disassembly desynchronization [69] (as shown in Listing 2). To present such overlap more explicitly, we use a special technique to convey the overlap information in the text. Firstly, DISASLLM groups all overlapping blocks in a transitive relation. In other words, two blocks that do not overlap are still in the same group if they both overlap with another common block. This problem can be considered a variant of the problem of merging overlapping intervals [7]. We describe the details of this algorithm in §V-F. The strategy described in §V-E1 represents groups with a single block. For groups with more than one block (i.e., those with overlaps), DISASLLM wraps them using a special

Listing 2 An example of overlap minimization.

```
0x401000:              0x401000:
cmp eax, eax           cmp eax, eax
je 0x401007            je 0x401007
; 0x401004             nop
                       int3
<<<<<<<               ; 0x401006
0x401004:
nop                    <<<<<<<
int3                   0x401006:
call 0x4011c3          call 0x4011c3
add BYTE PTR [rdi+0x1], bh   add BYTE PTR [rdi+0x1], bh
; 0x401011             ; 0x401011
=======               =======
0x401007:              0x401007:
mov eax, 0x1           mov eax, 0x1
mov edi, 0x1           mov edi, 0x1
; 0x401011             ; 0x401011
>>>>>>>               >>>>>>>

0x401011:              0x401011:
movabs rsi, 0x402000   movabs rsi, 0x402000
mov edx, 0xe           mov edx, 0xe
syscall                syscall
; 0x401022             ; 0x401022
```

notation similar to the git conflict marker [10]. An example of such representation is shown in the left part of Listing 2, where the block with the range 0x401004−0x401011 and the block with the range 0x401007−0x401011 overlap with each other and are wrapped with markers.

However, simply using such a representation does not effectively demonstrate the overlap. For example, if block b_1 overlaps with block b_2, but only a few of the ending instructions in b_1 overlap with b_2, while b_1 is a large block. In this case, using the conflict marker to wrap the two whole blocks does not convey the overlap in the most representative manner. Instead, we should *minimize the overlap region* as much as possible by splitting the overlapped blocks. To achieve this, we apply the same algorithm for grouping overlapping intervals described in §V-F to all instructions in the overlapped group (i.e., multiple transitive overlapped blocks), treating each instruction as an interval. This results in a list of groups consisting of possibly overlapped instructions. For all groups consisting of multiple instructions (i.e., where the overlap exists), we split the basic block at both the start and end addresses of the group to minimize the overlapped regions.

F. Group Overlapping Intervals

Problem statement. The input of the algorithm is a list of intervals $I = \{(a_i, b_i) \mid i = 0, 1, 2, \ldots, n-1\}$, where n is the number of intervals. In our scenario, the interval is the start address and the end address of either an instruction or a block. We consider two intervals (a_i, b_i) and (a_j, b_j) as overlapped if $a_i \leq a_j < b_i$ or $a_j \leq a_i < b_j$. Our definition of overlap is different from others [7] in the sense that we do not consider $a_j = b_i$ or $a_i = b_j$ as overlap, because adjacent instructions and blocks are not overlapped ones in our scenario. The task is to group all overlapped intervals in I, where the overlap relation is transitive, so two intervals that do not overlap with each other should be in the same group if they both overlap with another interval.

Algorithm. To achieve this, we firstly sort I according to a_i

in ascending order, and iterate over the sorted list. We always maintain b', the maximum b value of the last group. If the a of the next interval is smaller than b', we add it to the last group and update b'; otherwise, create a new group that only contains this interval. The details are shown in Algorithm 1.

Algorithm 1 Group Overlapping Intervals

Require: $I = \{(a_i, b_i) \mid i = 0, 1, 2, \ldots, n-1\}$
 1: sort I according to a in ascending order
 2: $b' := I[0].b$
 3: $g := \{I[0]\}$
 4: $G := \{\}$
 5: **for** $i = 1$ **to** $n-1$ **do**
 6:   **if** $I[i].a < b'$ **then**
 7:     g.append($I[i]$)
 8:     $b' := max(b', I[i].b)$
 9:   **else**
 10:     G.append(g)
 11:     $b' := I[i].b$
 12:     $g := \{I[i]\}$
 13:   **end if**
 14: **end for**
 15: G.append(g)
 16: **return** G

VI. THE LLM-BASED VALIDITY CLASSIFIER

In this section, we outline the process of fine-tuning Llama 3 [15] to transform it into an encoder model capable of identifying instruction validity through token classification, utilizing techniques proposed in LLM2Vec [21]. First, we describe the forward propagation process, explaining how the probability outputs for validity are generated from the input text snippet of decoded assembly instructions (§VI-A). Next, we describe the process of fine-tuning the decoder transformer model to develop an encoder model specialized in identifying instruction validity. This encoder model can compute the embedding vector of an instruction within the context of a snippet (§VI-B). Finally, we explain the training process for the last linear classifier layer, which generates the probability values for instruction validity (§VI-C).

A. Forward Propagation

We utilize the forward propagation method proposed by LLM2Vec [21], which is generally the same as the forward propagation used in the encoder model for token classification, with slight variations.

First, given a snippet of decoded assembly text with the instructions to be checked separated (i.e., in the form of "words"), the text is tokenized using the fast tokenizer [53]. This allows us to determine which "word" each token belongs to. Additionally, a special token <s> is added at the beginning to mark the start of the sequence. These k tokens are then input into the transformer model [70], so that the last hidden layer outputs k vectors matching each of the input tokens. For a token at index i, its embedding vector in the last hidden layer is denoted as e_i.

The embedding of an instruction constituted by n tokens at indexes $\{i_0, i_0 + 1, ..., i_0 + n - 1\}$ is the mean value of the

corresponding output vectors in the last hidden layer, computed as follows:

$$e = \frac{\sum_{i=i_0}^{i_0+n-1} e_{i-1}}{n}$$

It is important to note that e_{i-1} is used instead of e_i here: instead of using the output vector of each token, the output vector of the previous token of each token is used.

Finally, the embedding vector of the instruction is input into a linear classifier with an input size equal to the dimension of e and an output size of 1. The sigmoid function [33] is then applied to the output value to calculate the final probability indicating the validity.

B. MNTP Training

Similar to LLM2Vec, we fine-tuned the model using masked next token prediction (MNTP). However, instead of using texts from English Wikipedia, the model is trained on assembly code gathered from compiled binaries. To generate a large dataset of binaries, we used AnghaBench [27]. This benchmark consists of a substantial number of individually-compilable functions extracted from open-source repositories. We compiled these functions into binaries at different optimization levels. Additionally, because normal compilation generally does not insert non-code bytes into the code region, we also compiled the functions with junk byte insertion between basic blocks, using the same obfuscation approach described in §III-A. The ground truth instruction addresses were also gathered by modifying the source code of the `clang` assembler [46], [6].

For each generated binary, we disassembled it according to the ground truth addresses to obtain all valid instructions. Additionally, we decoded at incorrect addresses to obtain invalid instructions, placing these invalid instructions into comments. This included addresses of non-code bytes as well as addresses within the middle of valid instructions. An example snippet of the assembly text used for training is shown in Listing 3. Invalid instructions are decoded at all non-code byte addresses, and one random offset in the middle of each multibyte valid instruction is selected to decode an additional invalid instruction. We do not select all possible offsets within the valid instructions because this would produce too many invalid instructions, making the dataset excessively large and the instructions unbalanced. The validity of each decoded instruction is also marked at the end of the line with a comment (i.e., `;  valid` and `;  invalid`). Besides, each address referenced by any instruction in the text is labeled with its hexadecimal address value (e.g., `0x29f3:`).

This form of the dataset aims to enable the model not only to learn to be an encoder but also to become more familiar with both valid and invalid assembly code. According to Llama's reports on how the pretrained dataset is gathered [67], [68], [61], the dataset includes only publicly available code and does not contain a large amount of assembly code generated by the compiler, let alone invalidly decoded instructions in the executable. Therefore, this fine-tuning aims to address and complement this deficiency.

Listing 3 Part of the text in the MNTP dataset.

```
.byte 0xc9 ; leave  ; invalid
.byte 0x88 ; mov bl, ch ; invalid
jmp 0x29f3 ; valid
; offset 1: add ch, bl ; invalid
jmp 0x29f3 ; valid
; offset 1: add byte ptr [rax - 0x75], cl ; invalid

0x29f3:
mov rsp, qword ptr [rbp - 0x98] ; valid
; offset 2: movsd dword ptr [rdi], dword ptr [rsi] ; invalid
```

The training settings are generally the same as those in the original work of LLM2Vec [21]: the MNTP fine-tuning is performed using LoRA [34] with $r = 16$ and a masked probability of 0.2. Additionally, we increased the total number of training steps from 1,000 to 100,000 to allow for a longer training period. This is necessary because our dataset contains much more tokens than the English Wikipedia dataset in total.

C. Classifier Training

After MNTP fine-tuning, we can compute the embedding of an instruction within a snippet of decoded assembly instructions. Using this embedding, we can then train a linear classifier to identify the validity of the instruction. We follow the practice of LLM2Vec [21] by training only the parameters of the linear classifier while keeping the decoder transformer model frozen. However, since we only have two labels, one neuron with a sigmoid function [33] is sufficient, instead of using softmax with two neurons in the last layer.

The supervised dataset used to train the classifier consists of entries where each text is split into "words" with corresponding labels, similar to most datasets used in token classification problems, such as [62]. In our context, the text is an assembly code snippet, and the assembly instructions with their validity labels are separated as "words". A simplified example of such an entry in the dataset is shown in Listing 4. Each region separated by horizontal lines specifies a "word" in the text, with the corresponding label on the right-hand side. The labels 0 and 1 represent valid and invalid instructions, respectively, while the label -100 indicates "words" that are not considered when computing the loss function.

Listing 4 An example snippet in supervised dataset.

`0x8279c:` `xchg esp, eax`	-100
`sub bh, bl`	0
`jnp 0x82768`	0
`0x827a1:`	-100
`and al, 0x24`	0
`or byte ptr [rax], al`	0
`add byte ptr [rax], al`	0
`jmp 0x827e8`	1
`0x827aa:`	-100
`loop 0x827f4`	0
`mov eax, dword ptr [rsp + 0x10]`	0
`bswap rax`	1

`0x827b3:`
`mov qword ptr [rsp + 0x38], rax`
`mov rax, qword ptr [rsp + 0x28]`
	-100

To obtain this dataset, a supervised dataset is gathered to be specialized for the task required by the disassembly strategy of DISASLLM. To achieve this, we use the same disassembly strategy described in §V on the same binaries used in MNTP training, but replace the validity classifier with the ground truth, ensuring it always provides the correct validity. Each time the classifier is invoked, the queried assembly snippet is split so that each instruction being checked becomes a single word. These checked instructions are labeled with the ground truth validity value: `0` for invalid and `1` for valid. For other texts, the label is marked as `-100` to inform the trainer that the word is not considered in the loss function. These include not only non-instruction text such as address labels or overlap markers, but also instructions not checked by the classifier (e.g., `xchg esp, eax` in Listing 4). Such split texts, along with the corresponding labels, are recorded as an entry in the dataset. In this way, we can construct the most appropriate behavior for the classifier to learn. When applied to all binaries of AnghaBench, this approach produces an excessive number of dataset rows required by the linear classifier learning. Therefore, we randomly select a subset of the results.

VII. EVALUATION

Implementation. We implement DISASLLM in 2,000+ lines of Python code, along with 200+ lines of Python code to gather the training data. We use Capstone [12] as the underlying disassembler engine that decodes a byte sequence into an instruction. To implement the disassembly method described in §V, we wrap the disassembly graph in a Python class, exposing an efficient interface for basic block insertion, modification, and deletion. To implement the batching mechanism (§V-D), instead of sharing one queue for classification tasks required by different components, DISASLLM only shares the same queue for classification tasks that have same form of post-processing. For example, the classification tasks requested by the prefilter step are in a different queue from those requested by the reverse infilling. Additionally, DISASLLM uses a Python abstract interface to represent the classifier primitive, which can either be the LLM-based validity classifier used in real disassembly tasks or one that checks the validity according to the ground truth directly, which is used to gather training data as described in §VI-C.

Settings. We ran all of our experiments on an Ubuntu 22.04.4 LTS system equipped with an NVIDIA H100 GPU and an AMD EPYC 9124 16-Core Processor.

Research questions. In the evaluation part, we explore the following research questions through experiments.

- **RQ1**: How effective is DISASLLM to disassemble obfuscated executables compared to other disassemblers?

- **RQ2**: How accurate is the LLM-based validity classifier when it is used by the different components of DISASLLM described in §V?

- **RQ3**: How effective is our fine-tuned model for identifying valid instructions when it is used alone?

- **RQ4**: How fast is DISASLLM for the disassembly task?

A. Comparison with Other Disassemblers (RQ1)

Benchmark. We use the same set of real-world open-source programs as used by DeepDi [79], which contains a wide range of targets ranging from cryptography libraries (`openssl`) to SQL database engines (`sqlite3`). Each program is compiled and obfuscated in the same way as described in §III-A (e.g., OLLVM-based [41] control flow flattening [45] and junk bytes insertion between flattened blocks). To ensure the insertion of junk bytes, we add a bogus control flow (i.e., control flow that never executes) from the `SwitchInst` of the dispatcher block to all inserted junk bytes. This obfuscation scheme ensures that junk bytes are inserted before almost all basic blocks, while the control flow information is stripped by the flattening. Therefore, disassemblers' anti-obfuscation ability can be thoroughly tested. To obtain the groundtruth, we modify the LLVM assembler [46] to collect the starting addresses of all valid instructions.

Baseline disassemblers. We firstly choose DeepDi [79] as the state-of-the-art disassembly approach based on machine learning. We also choose Angr [63] because it is the best industrial disassembler according to the measurement of [54]. However, we only use its `CFGFast` feature instead of `CFGEmulated` that leverages the symbolic execution, because it throws an exception when applied on our obfuscated binaries. We also chose Ddisasm [32] as a heuristic-based baseline. There are also some disassemblers [19], [69] aiming to tackle obfuscation. We did not include them because the obfuscations they handle is different from the one used in this benchmark. Moreover, according to the maintainer of BINSEC, BB-DSE [19] is unavailable now. Besides, we did not include XDA [57] because its fine-tuned disassembler model is not released, while according to the experiment of DeepDi, they demonstrate similar results, so DeepDi alone is already representative enough for the machine learning approaches.

Results. The results are shown in Table II, which show that DISASLLM outperforms other state-of-the-art disassemblers. Angr and Ddisasm are even unable to correctly disassemble most of the instructions. DeepDi, the second-best disassembler, achieves a decent score on decoding all instructions but fails to decode the first instructions following the junk byte regions. Moreover, we find that DISASLLM achieves higher precision than the recall in general, which means some correct instructions are not decoded. We try to explain this results using intermediate metrics in §VII-B.

B. Intermediate Metrics of Classifier Model (RQ2)

In this part, we demonstrate the effectiveness of the LLM-based model when it is used by the different components of DISASLLM. We consider three DISASLLM components that utilizes the classifier: disassembly checking, reverse infilling and forward infilling. Each of these components can be further separated into two components: prefilter component, which checks multiple instructions at once, and single component, which checks each instruction one by one (except for reverse infilling, because there can be multiple instructions ending at the same address). Therefore, we have $3 \times 2 = 6$ components in total. The details about these components have been described previously in §V-B and §V-C.

TABLE II: The results of comparing DISASLLM to other disassembler counterparts.

Disassembler	Type	Metrics	curl	diffutils	GMP	ImageMagick	libmicrohttpd	libtomcrypt	OpenSSL	PuTTy	SQLite	zlib
DisasLLM	All	Precision	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**	**0.99**	**0.99**	**0.99**	**0.99**	**0.98**
		Recall	0.92	**0.94**	**0.95**	0.92	**0.94**	0.94	0.91	0.93	0.93	**0.94**
		F1	**0.95**	**0.96**	**0.96**	**0.95**	**0.96**	0.96	0.95	0.96	**0.96**	**0.96**
	Junk	Precision	**0.94**	**0.92**	**0.90**	**0.89**	**0.92**	**0.91**	**0.94**	**0.93**	**0.94**	**0.92**
		Recall	**0.91**	**0.89**	**0.89**	**0.86**	**0.91**	**0.89**	**0.87**	**0.89**	**0.90**	**0.91**
		F1	**0.92**	**0.91**	**0.89**	**0.87**	**0.91**	**0.90**	**0.90**	**0.91**	**0.92**	**0.92**
DeepDi	All	Precision	0.95	0.96	0.96	0.96	0.96	0.97	0.96	0.97	0.96	0.96
		Recall	**0.93**	0.93	0.92	**0.94**	0.93	**0.95**	**0.94**	**0.96**	**0.94**	0.93
		F1	0.94	0.94	0.94	0.95	0.94	**0.96**	**0.95**	**0.96**	0.95	0.95
	Junk	Precision	0.60	0.62	0.65	0.57	0.59	0.67	0.63	0.65	0.61	0.58
		Recall	0.51	0.51	0.48	0.52	0.47	0.52	0.55	0.58	0.52	0.47
		F1	0.55	0.56	0.55	0.54	0.52	0.59	0.59	0.61	0.56	0.52
Angr	All	Precision	0.68	0.67	0.66	0.75	0.70	0.75	0.73	0.78	0.72	0.71
		Recall	0.74	0.71	0.77	0.80	0.73	0.84	0.79	0.82	0.75	0.74
		F1	0.71	0.69	0.71	0.77	0.71	0.80	0.76	0.80	0.73	0.72
	Junk	Precision	0.05	0.05	0.04	0.04	0.05	0.04	0.07	0.10	0.06	0.05
		Recall	0.10	0.10	0.09	0.09	0.09	0.09	0.14	0.18	0.11	0.09
		F1	0.07	0.07	0.05	0.06	0.06	0.06	0.10	0.13	0.08	0.06
DDisasm	All	Precision	0.84	0.86	0.81	0.85	0.84	0.82	0.87	0.92	0.87	0.86
		Recall	0.23	0.21	0.19	0.12	0.18	0.22	0.33	0.41	0.26	0.17
		F1	0.37	0.34	0.31	0.22	0.30	0.35	0.48	0.56	0.40	0.29
	Junk	Precision	0.25	0.29	0.17	0.27	0.26	0.16	0.31	0.42	0.32	0.31
		Recall	0.08	0.08	0.05	0.05	0.07	0.08	0.14	0.18	0.10	0.07
		F1	0.12	0.12	0.08	0.09	0.11	0.10	0.19	0.25	0.15	0.12

Therefore, for each of the components, when it uses the LLM-based classifier model, we record the addresses of the instructions being checked and the result probability values, along with the annotation of the component that requests the classification. Based on the ground truth valid instruction addresses, we can then use these records to compute the metrics about the validity identification of each of the components. To be specific, we count the number of false positives, false negatives, true positives and true negatives. The results are presented in Table III.

Firstly, we can find that during the process of checking the decoded instructions in the initial disassembly results, the prefilter step is already capable of identifying most of the instructions, with a very high accuracy. By contrast, the remaining instructions that are then checked one by one are harder to correctly classify, which aligns with our assumptions. Additionally, false negatives are generally higher than the false positives, especially in the prefilter step. We believe this also explains the metrics in Table II where the recall values of DISASLLM are lower than the precision values.

During the process of fixing the disassembly, the scenarios are almost similar for both reverse infilling and forward infilling: the prefilter (i.e., checking multiple decoded instructions as fixing candidates at once) handles most of the instructions correctly, while the single-instruction step performs worse. Moreover, there are many true negatives in the prefilter of forward infilling, because most of the decoded instructions in the invalid regions are invalid ones, and they are successfully identified when multiple instructions are checked simultaneously.

These intermediate results suggest that the model indeed works with high accuracy when used by the different components of DISASLLM. However, since there are still considerable false positives and false negatives, the model capability can be further improved by better fine-tuning.

C. Capability of Fine-tuned Model (RQ3)

In this part, we conduct an ablation study investigating the capability of our fine-tuned model when it is used alone, and compare its capability with some other LLM baselines. To be specific, we utilize a naive approach for disassembly: we use LLM model to check the validity of instruction decoded at each address in the code region, and output all addresses that are considered as valid.

However, LLM does not obtain enough information if we only provide the decoded instruction at the address to be checked. Therefore, we also provide the surrounding instructions of start addresses within the range $a - 50$ and $a + 50$, where a is the address to be checked. We utilize two approaches to provide the surrounding instructions: 1) We decode all addresses within the range (i.e., superset disassembly) and provide all of them to LLM. 2) We conduct linear disassembly starting from $a - 50$ and a to get the instructions from both, and show the possible overlap using the format we mentioned in §V-E2. Note that these two approaches both ensure the instruction to be checked at a is decoded.

To enable decoder LLM model to conduct such task, we leveraged the prompt engineering. In the prompt, we introduce the task of identifying the validity of the instruction by informing LLM how instruction can be invalid, the format of the provided assembly snippet, and how the snippet is disassembled (i.e., superset or linear). Finally, we provide the disassembled snippet and ask if a particular instruction is valid, and the response is limited to generate only one character: 'Y' for valid or 'N' for invalid.

We chose Codestral 22B [3] and Llama3 8B [15] as the baseline models for this task, because Codestral is the current state-of-the-art open-weight code model, and Llama3 8B is the model that DISASLLM is fine-tuned on. We also tested Meta LLM Compiler [26], the state-of-the-art model that has been trained on assembly text. However, the model fails to

TABLE III: The metrics of LLM-based validity classifier when being used by the components of DISASLLM.

Component		Metrics	curl	diffutils	GMP	ImageMagick	libmicrohttpd	libtomcrypt	PuTTy	SQLite	zlib
Checking	Prefilter	FP	835	178	917	5440	262	544	567	1251	114
		FN	8190	968	2894	36672	1596	4393	7542	14358	653
		TP	206261	29409	130508	858529	51269	145085	164896	341318	21703
		TN	250456	36194	150463	800430	54954	117969	140301	372861	22050
	Single	FP	3266	530	2540	15554	749	1897	2137	4890	309
		FN	5138	527	1659	22804	895	4524	4667	9335	552
		TP	20832	2630	8345	116998	5445	12656	18127	40394	2098
		TN	2746	362	1586	9934	550	1295	1362	3940	199
Reverse	Prefilter	FP	502	103	355	1793	155	246	424	879	65
		FN	634	132	56	1965	192	98	427	863	34
		TP	41177	6101	20114	120717	9175	17877	20152	62408	3738
		TN	75268	10484	36607	255438	16249	36731	36490	110532	6969
	Single	FP	357	52	232	1380	91	166	246	750	40
		FN	222	67	52	963	58	105	164	403	12
		TP	2423	350	713	14648	596	2540	1710	4003	194
		TN	586	72	287	2896	112	284	413	1024	52
Forward	Prefilter	FP	4165	597	1367	10025	928	1528	3345	7341	381
		FN	7541	719	1357	24308	1062	2006	3602	10357	327
		TP	20796	3073	14380	61932	4697	8103	9632	29355	1889
		TN	236686	30890	149741	836289	49908	111068	120236	335712	19462
	Single	FP	514	75	330	1840	108	285	357	774	45
		FN	926	164	226	3015	163	483	610	1932	63
		TP	1131	227	293	3764	198	465	650	2035	71
		TN	1551	209	811	5166	318	731	897	2384	145

TABLE IV: Execution speed of DISASLLM.

	curl	diffutils	GMP	ImageMagick	libmicrohttpd	libtomcrypt	OpenSSL	PuTTy	SQLite	zlib
Time (seconds)	11588.09	1595.89	6450.46	44274.76	2490.57	5625.90	75514.68	6508.85	18290.14	990.12
Size (bytes)	1616997	227299	1040736	6605067	372941	1052911	8343974	1118449	2531104	155647
Speed (bytes/second)	139.54	142.43	161.34	149.18	149.74	187.15	110.49	171.84	138.39	157.20

follow our instruction of answering only 'Y' or 'N', so we exclude it from our results. We did not include any online proprietary models, since the task requires so many tokens that the costs would be too high. Additionally, we also include DeepDi, because it also provides API that identifies if an offset is the start of a valid instruction given the byte sequence. Note that the DeepDi here works differently than the DeepDi in Table table II, where it checks all addresses at once, while the DeepDi only checks one address for each classification. For fine-tuned model of DISASLLM, we only tested the linear disassembly, because our model is not trained on the dataset of superset disassembly, which clearly cannot give decent results. Since checking all possible addresses can be too slow for LLM, we only consider the first page of the code region (i.e., 4096 bytes) of each obfuscated binary.

The results are presented in Table table V with the same format as Table table II. We mark the subscript L for linear disassembly, and subscript S for superset disassembly. Firstly, we can find that DISASLLM- where only the LLM-based validity classifier alone is used to disassemble the binary, produces worse results than the whole system of DISASLLM presneted in Table table II. This shows that our disassembly strategy is not only more efficient but also more effective. However, our model is still better than DeepDi, whose results are not very different from those in table II. Additionally, we can find that directly applying the decoder LLM on such classification task does not yield good results: they almost identify all instructions as valid, so precision is too low while the recall is pretty high. Besides, leveraging superset disassembly does not generate

better results than the linear disassembly, which suggests that the superset disassembly indeed provides too much redundant information that is not required by the validity identification.

D. Efficiency of Disassembly (RQ4)

Finally, we measure the efficiency of DISASLLM. We record the execution time of DISASLLM on each executable generated in §VII-A and calculate the average processing speed in "bytes per second". The batch size M was set to 32 in this experiment. The results are presented in Table IV.

According to the results, DISASLLM disassembles the binary in roughly 100+ bytes per second, meaning that an executable containing millions of bytes (MBs in size) will need hours to be disassembled. Although this is slower than some other disassemblers like DeepDi (while still faster than expensive approaches like symbolic execution-based ones), DISASLLM has the advantage of effectively handling the obfuscation and producing more accurate results. We thus consider it a valuable and practical tradeoff. We leave the performance optimization as a future work.

VIII. DISCUSSION

Efficiency optimization. Although already optimized using prefiltering and batching, when disassembling large binary with millions of bytes, DISASLLM still needs to take many hours to accomplish the task. To address this issue, one may suggest not to check all instructions, but instead, if one instruction is identified as valid, the instructions that must

TABLE V: The results of using models directly on the disassembly.

Disassembler	Type	Metrics	curl	diffutils	GMP	ImageMagick	libmicrohttpd	libtomcrypt	OpenSSL	PuTTy	SQLite	zlib
DISASLLM-	All	Precision	0.90	0.88	0.88	0.93	0.93	0.92	0.92	0.91	0.92	0.92
		Recall	0.92	0.91	0.91	0.89	0.94	0.93	0.87	0.91	0.94	0.97
		F1	0.91	0.90	0.89	0.91	**0.94**	0.92	0.89	0.91	0.93	0.94
	Junk	Precision	**0.83**	**0.75**	**0.85**	**0.87**	**0.92**	**0.79**	**0.81**	**0.86**	**0.90**	**0.83**
		Recall	**1.00**	0.92	**1.00**	0.95	0.94	0.95	0.87	0.92	0.95	0.96
		F1	**0.91**	**0.83**	**0.92**	**0.91**	**0.93**	**0.86**	**0.84**	**0.89**	**0.93**	**0.89**
DeepDi	All	Precision	**0.94**	**0.97**	**0.96**	**0.96**	**0.96**	**0.98**	**0.96**	**0.94**	**0.96**	**0.97**
		Recall	0.91	0.91	0.95	0.95	0.91	0.93	0.94	0.95	0.97	0.97
		F1	**0.93**	**0.94**	**0.96**	**0.95**	0.93	**0.95**	**0.95**	**0.95**	**0.97**	**0.97**
	Junk	Precision	0.55	0.68	0.68	0.66	0.60	0.65	0.68	0.59	0.73	0.63
		Recall	0.49	0.48	0.66	0.63	0.45	0.44	0.61	0.64	0.79	0.61
		F1	0.52	0.56	0.67	0.64	0.52	0.53	0.64	0.61	0.76	0.62
Codestral$_S$	All	Precision	0.28	0.25	0.25	0.25	0.24	0.22	0.27	0.27	0.29	0.25
		Recall	0.98	0.99	**0.99**	**0.99**	0.99	**1.00**	**1.00**	**1.00**	**0.99**	0.99
		F1	0.44	0.39	0.40	0.40	0.39	0.36	0.43	0.42	0.45	0.40
	Junk	Precision	0.22	0.18	0.19	0.21	0.18	0.15	0.21	0.22	0.22	0.18
		Recall	1.00	**1.00**	1.00	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
		F1	0.36	0.31	0.31	0.35	0.31	0.26	0.35	0.37	0.36	0.31
Codestral$_L$	All	Precision	0.28	0.26	0.25	0.29	0.29	0.26	0.28	0.28	0.32	0.31
		Recall	**0.99**	**1.00**	0.99	0.99	**1.00**	1.00	0.99	1.00	0.98	**1.00**
		F1	0.43	0.41	0.39	0.45	0.45	0.41	0.44	0.44	0.49	0.47
	Junk	Precision	0.19	0.18	0.18	0.22	0.20	0.14	0.21	0.20	0.24	0.21
		Recall	1.00	1.00	0.99	0.99	0.99	0.99	1.00	0.99	0.99	1.00
		F1	0.32	0.31	0.30	0.37	0.33	0.25	0.35	0.33	0.39	0.35
Llama$_S$	All	Precision	0.30	0.27	0.26	0.32	0.33	0.29	0.33	0.31	0.34	0.34
		Recall	0.92	0.95	0.93	0.85	0.90	0.94	0.85	0.90	0.85	0.93
		F1	0.45	0.43	0.41	0.47	0.49	0.44	0.47	0.46	0.49	0.49
	Junk	Precision	0.24	0.25	0.21	0.28	0.27	0.18	0.30	0.27	0.34	0.27
		Recall	0.90	0.95	0.89	0.81	0.86	0.91	0.87	0.87	0.79	0.93
		F1	0.38	0.39	0.33	0.42	0.42	0.30	0.44	0.41	0.48	0.42
Llama$_L$	All	Precision	0.27	0.25	0.24	0.28	0.29	0.25	0.27	0.27	0.31	0.30
		Recall	0.97	0.99	0.97	0.93	0.98	0.93	0.94	0.97	0.94	0.97
		F1	0.42	0.40	0.39	0.43	0.45	0.39	0.42	0.42	0.46	0.45
	Junk	Precision	0.20	0.20	0.18	0.23	0.21	0.15	0.21	0.20	0.25	0.22
		Recall	0.96	0.99	0.96	0.94	0.99	0.94	0.96	0.96	0.95	0.97
		F1	0.33	0.33	0.30	0.37	0.35	0.26	0.35	0.32	0.39	0.36

follow its control flow should also be valid. However, the problem of this claim is that the inserted junk bytes can appear like valid instructions: while decoding them does not affect the disassembly results since they do not overlap with the real valid instructions, the instructions following their control flow may cause such overlap. Thus, it is still necessary to check all instructions. Another potential approach is to use a smaller model, such as CodeBERT [31], for prefiltering, so that validity of each instruction that is obvious even from the perspective of a small language model can be confirmed first. We leave this direction as the potential future work.

Valid overlapping instructions. In general, even in the obfuscated binary, two overlapping instructions cannot be both valid. An exception case, also handled by CoDisasm [22], involves a `jmp` instruction jumping into itself. For example, consider the disassembly of the byte sequence `eb ff c9`, where `eb ff` is decoded to `jmp +1` with target address pointing to `ff c9`, which is decoded to `dec ecx`. Since the two overlapping instructions are both executed at runtime, they are both valid. In principle, DISASLLM is capable of handling this case if such pattern occurs in the initial disassembly graph, as long as the model has learned this pattern, because our disassembly method does not intentionally resolve all overlapped blocks. However, if such pattern occurs in the invalid region which is going to be fixed, DISASLLM is not able to disassemble this pattern correctly because we assume no overlap in the invalid region when it is being fixed.

Another more extreme case involves hiding all real instructions into immediate numbers of another set of valid instructions [36]. Since such obfuscation technique aims to defeat the self-repairing feature of the disassembly, current design of DISASLLM cannot handle this scenario. A potential future extension that may allow DISASLLM to handle this involves training another classifier to identify if immediate number of a valid instruction may hide another instruction, and if the classifier says yes, DISASLLM can start an extra disassembly starting from the immediate number of the instruction and check if they are valid.

Combining with traditional disassembly techniques. Currently, our initial disassembly process only leverages the most basic traditional disassembly approach, which does not include resolving target addresses of an indirect branch, currently handled by symbolic execution [63] or heuristic-based approach [54]. These techniques can also be applied upon our initial disassembly algorithm to get a more comprehensive initial disassembly graph.

IX. RELATED WORK

A. Disassembly and Deobfuscation

Disassembly of normal binary. Commonly-used industrial disassemblers include IDA [5], Ghidra [4], Angr [63], Binary

Ninja [2], Radare2 [9], BAP [1] and objdump [8]. According to investigations of previous works [54], [16], these disassemblers generally leverage linear or/and recursive disassembly, along with additional algorithms and heuristics to improve the effectiveness. DISASLLM is similar to them in the sense that the algorithm is our LLM-based method. Datalog disassembly [32] leverages datalog, a declarative logic programming language, to implement the algorithm and heuristics for binary disassembly, such as the heuristic to resolve overlapping blocks. Probabilistic disassembly [51] computes probability for each instruction to be valid based on some heuristics known as hints. By contrast, DISASLLM also computes such probability but DISASLLM relies on LLM-based approach instead of any heuristic. The disassembler of SAFER [59] also leverages a probability-based approach combined with static analysis using a prioritized error correction algorithm. XDA [57] pre-trained an encoder transformer with dataset of byte sequences in the compiled binaries, using masked token prediction as the training objective, which was then fine-tuned for instruction address identification. DISASLLM also leverages a similar procedure, but it was trained on decoded instructions instead of raw bytes, which contain clearer semantic information for the model to learn. DeepDi [79] disassembles all possible addresses of the code region to construct an instruction flow graph for identifying the validity of each instruction using a graph neuron network. However, as we have demonstrated, its resilience to obfuscated code is limited.

Disassembly of obfuscated binary. Even if disassembling normal binary is already a challenging task, some previous works also aimed to disassemble obfuscated binary. An early work [44] proposes a method to statically disassemble obfuscated binary, which mainly relies on statistical and heuristic methods to exclude invalid instructions and find valid instructions. Another early work [43] leverages the decision tree, through feature engineering on the decoded instructions, to detect invalid disassembly caused by the obfuscation. The opaque predicates detective [69] applies a series of heuristics to identify the unreachable branch of an opaque predicate that causes desynchronization of the disassembly. CoDisasm [22] aims to extract and disassemble self-modifying obfuscated binary. Their disassembly approach relies on the dynamic trace and is resilient to obfuscation technique containing overlapping instructions that are both valid. Backward-Bounded DSE [19] utilizes the dynamic trace for concolic execution in order to decide infeasible paths, which provide complementary information for their sparse disassembly method aiming to be resilient to opaque predicate and call stack tampering. However, these two works above both heavily depend on dynamic trace, so uncovered paths that in general constitute most parts of the program cannot benefit much from these methods, because achieving a high program coverage is a problem known to be hard. DISASLLM, on the other hand, not only performs well without any execution trace, but also can benefit from execution trace by providing them as priorly known valid instruction addresses.

Other deobfuscation techniques. Besides defeating obfuscation causing disassembly desynchronization, other obfuscation techniques and approaches to deobfuscate them also exist. A generic deobfuscation approach [78] leverages taint propagation and dependency analysis on the execution trace of an obfuscated executable to pinpoint instructions with semantic significance. Another work [77] proposes methods to improve the effectiveness of symbolic execution on the obfuscated code. LOOP [52] also utilizes concolic execution on the execution trace to identify the opaque predicates in the trace. All of these works similarly require a (symbolic) execution trace of the obfuscated program, unlike DISASLLM.

B. LLM for Program Analysis

Recent advancement of the large language model has contributed significantly to the field of program analysis, including fuzzing [50], [29], [14], [74], static bug detection [23], [47], [64], [65], [72] and code repair [75], [40], [30], [55], [39].

Application on reverse engineering. A pioneering measurement study [56], by prompting to ask LLM questions about the decompiled code, firstly investigates the ability of LLM to help reverse engineering. DeGPT [35] follows such direction by designing a prompt system to optimize the output from the decompiler, along with a checker that ensures the optimized output from LLM has the same semantic as the original decompiled code. LLM4Decompile [66] fine-tunes the LLM to aid the decompilation task. The model supports both decompilation from assembly to C code directly and refinement of the decompiled code from Ghidra to produce better decompiled code. Nova [38] utilizes hierarchical attention and contrastive learning to improve LLM's capability on understanding and generating the assembly code. BinaryAI [37] leverages LLM for the task of software composition analysis by computing the embeddings of functions in the binary and matching them with source functions of the third-party libraries. LATTE [49] improves the binary taint analysis with LLM by automating the manual components of taint analysis, using static analysis and prompt engineering. Although similar to DISASLLM, these works all try to improve reverse engineering by the usage of LLM, they contribute to the different aspects of the reverse engineering. By contrast, to the best of our knowledge, DISASLLM is the first work that leverages LLM for the disassembly task.

X. CONCLUSION

In this work, we proposed DISASLLM, an end-to-end system for disassembling obfuscated binary executables by effectively leveraging the capability of large language model. DISASLLM consists of a fine-tuned LLM-based validity classifier aiming to identify if decoded instructions are correct, and a disassembly strategy that utilizes such classifier to disassemble the obfuscated executable in an end-to-end manner. According to our evaluation, the approach is effective on the disassembly task compared to other state-of-the-art disassemblers, showing the potential of our method. We believe that our research has opened a new direction in leveraging LLM on the disassembly task.

REFERENCES

[1] "Binary analysis platform," https://github.com/BinaryAnalysisPlatform/bap.

[2] "Binary ninja," https://binary.ninja/.

[3] "Codestral: Hello, world!" https://mistral.ai/news/codestral/.

[4] "ghidra," https://github.com/NationalSecurityAgency/ghidra.

[5] "Ida pro - hex rays - state-of-the-art binary code analysis solutions," https://hex-rays.com/ida-pro/.

[6] "llvm-project," https://github.com/llvm/llvm-project.

[7] "Merge overlapping intervals," https://www.geeksforgeeks.org/merging-intervals/.

[8] "objdump(1) - linux man page," https://linux.die.net/man/1/objdump.

[9] "Radare2: Libre reversing framework for unix geeks," https://github.com/radareorg/radare2.

[10] "Resolving a merge conflict using the command line," https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line.

[11] "Token classification," https://huggingface.co/tasks/token-classification.

[12] "The ultimate disassembly framework – capstone – the ultimate disassembler," https://www.capstone-engine.org/.

[13] O. J. Achiam and S. A. et al., "Gpt-4 technical report," 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257532815

[14] J. Ackerman and G. V. Cybenko, "Large language models for fuzzing parsers (registered report)," *Proceedings of the 2nd International Fuzzing Workshop*, 2023.

[15] AI@Meta, "Llama 3 model card," 2024. [Online]. Available: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[16] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *USENIX Security Symposium*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:4657306

[17] X. Bai, L. Xing, M. Zheng, and F. Qu, "idea: Static analysis on the security of apple kernel drivers," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1185–1202.

[18] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *International Conference on Verification, Model Checking and Abstract Interpretation*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:6789794

[19] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded dse: Targeting infeasibility questions on obfuscated codes," *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 633–651, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:6208323

[20] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Network and Distributed System Security Symposium*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:29156207

[21] P. BehnamGhader, V. Adlakha, M. Mosbach, D. Bahdanau, N. Chapados, and S. Reddy, "Llm2vec: Large language models are secretly powerful text encoders," *ArXiv*, vol. abs/2404.05961, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269009682

[22] G. Bonfante, J. M. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:6096804

[23] H. Chen, Y. Zhang, X. Han, H. Rong, Y. Zhang, T. Mao, H. Zhang, X. Wang, L. Xing, and X. Chen, "Witheredleaf: Finding entity-inconsistency bugs with llms," *ArXiv*, vol. abs/2405.01668, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269588287

[24] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235755472

[25] C. G. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, 1995. [Online]. Available: https://api.semanticscholar.org/CorpusID:8229401

[26] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Meta large language model compiler: Foundation models of compiler optimization," 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:270924331

[27] A. F. da Silva, J. N. Rocha, B. C. F. Guimarães, and F. M. Q. Pereira, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 378–390, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:229576527

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:52967399

[29] J. Eom, S. Jeong, and T. Kwon, "Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation," *ArXiv*, vol. abs/2402.12222, 2024.

[30] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1469–1481, 2022.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *ArXiv*, vol. abs/2002.08155, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211171605

[32] A. Flores-Montoya and E. M. Schulte, "Datalog disassembly," in *USENIX Security Symposium*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:182952674

[33] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International Work-Conference on Artificial and Natural Neural Networks*, 1995. [Online]. Available: https://api.semanticscholar.org/CorpusID:2828079

[34] J. E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *ArXiv*, vol. abs/2106.09685, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235458009

[35] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with llm," *Proceedings 2024 Network and Distributed System Security Symposium*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267622140

[36] C. Jamthagen, P. Lantz, and M. Hell, "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries," *2013 Workshop on Anti-malware Testing Research*, pp. 1–9, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:25234080

[37] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Binaryai: Binary software composition analysis via intelligent binary source code matching," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 2771–2783, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267068832

[38] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, and X. Zhang, "Nova: Generative language models for assembly code with hierarchical attention and contrastive learning," 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:265445138

[39] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[40] H. Joshi, J. P. Cambronero, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, "Repair is nearly generation: Multilingual program repair with llms," in *AAAI Conference on Artificial Intelligence*, 2022.

[41] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm – software protection for the masses," *2015 IEEE/ACM 1st International Workshop on Software Protection*, pp. 3–9, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:307592

[42] H.-G. Kim, S. Kim, J. Lee, K. Jee, and S. K. Cha,

"Reassembly is hard: A reflection on challenges and strategies," in *USENIX Security Symposium*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:254103322

[43] N. Krishnamoorthy, S. K. Debray, and K. Fligg, "Static detection of disassembly errors," *2009 16th Working Conference on Reverse Engineering*, pp. 259–268, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:2763656

[44] C. Krügel, W. K. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX Security Symposium*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:6458864

[45] T. László and A. Kiss, "Obfuscating c++ programs via control flow flattening," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:5061467

[46] C. Lattner and V. S. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:978769

[47] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 474 – 499, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268375318

[48] C. Linn and S. K. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Conference on Computer and Communications Security*, 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:8864003

[49] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Li, and L. Sun, "Harnessing the power of llm to support binary taint analysis," *ArXiv*, vol. abs/2310.08275, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:263909501

[50] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[51] K. A. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1187–1198, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:85521444

[52] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:9509889

[53] A. Moi and N. Patry, "HuggingFace's Tokenizers," Apr. 2023. [Online]. Available: https://github.com/huggingface/tokenizers

[54] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 833–851, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220831521

[55] H. A. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2339–2356, 2021.

[56] H. A. Pearce, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, and B. Dolan-Gavitt, "Pop quiz! can a large language model help with reverse engineering?" *ArXiv*, vol. abs/2202.01142, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246472967

[57] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *ArXiv*, vol. abs/2010.00770, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:222125324

[58] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *USENIX Security Symposium*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:1563249

[59] S. Priyadarshan, H. Nguyen, and R. C. Sekar, "Accurate disassembly of complex binaries without use of compiler metadata," *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260903852

[60] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 289–300, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:8870772

[61] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. P. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. D'efossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *ArXiv*, vol. abs/2308.12950, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261100919

[62] E. T. K. Sang and F. D. Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," in *Conference on Computational Natural Language Learning*, 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:2470716

[63] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:3337994

[64] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, M. Shi, and Y. Liu, "Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning," *ArXiv*, vol. abs/2401.16185, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267311934

[65] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 2048–2060, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260682978

[66] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," *ArXiv*, vol. abs/2403.05286, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268297213

[67] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," *ArXiv*, vol. abs/2302.13971, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257219404

[68] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. M. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. S. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. M. Kloumann, A. V. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *ArXiv*, vol. abs/2307.09288, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259950998

[69] Y.-J. Tung and I. G. Harris, "A heuristic approach to detect opaque predicates that disrupt static disassembly," *Proceedings 2020 Workshop on Binary Analysis Research*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:215232596

[70] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Neural Information Processing Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13756489

[71] S. Wang, P. Wang, and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 236–247, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:15802894

[72] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and detecting misuses of deep learning

apis," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 2482–2493, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:265105013

[73] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:268093756

[74] C. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2023.

[75] C. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494, 2023.

[76] H. Xu, Y. Zhou, Y. Kang, F. Tu, and M. R. Lyu, "Manufacturing resilient bi-opaque predicates against symbolic execution," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 666–677, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:50774681

[77] B. Yadegari and S. K. Debray, "Symbolic execution of obfuscated code," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:6651074

[78] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. K. Debray, "A generic approach to automatic deobfuscation of executable code," *2015 IEEE Symposium on Security and Privacy*, pp. 674–691, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:16936908

[79] S. Yu, Y. Qu, X. Hu, and H. Yin, "Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *USENIX Security Symposium*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:244398600

[80] Z. Zhang, G. Tao, G. Shen, S. An, Q. Xu, Y. Liu, Y. Ye, Y. Wu, and X. Zhang, "Pelican: Exploiting backdoors of naturally trained deep learning models in binary code analysis," in *USENIX Security Symposium*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259267066