# Tady: A Neural Disassembler without Structural Constraint Violations

Siliang Qin, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Fengrui Yang and Hao Wang, *Tsinghua University;* Bolun Zhang, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Zeyu Gao and Chao Zhang, *Tsinghua University;* Kai Chen, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity25/presentation/qin

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

# Tady: A Neural Disassembler without Structural Constraint Violations

Siliang Qin[1,2], Fengrui Yang[3], Hao Wang[3], Bolun Zhang[1,2], Zeyu Gao [3], Chao Zhang [3]*, Kai Chen [1,2]*
[1] *Institute of Information Engineering, Chinese Academy of Sciences, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*
[3]*Tsinghua University, China*
*{qinsiliang, zhangbolun, chenkai}@iie.ac.cn*
*{yangfr23, hao-wang20, gaozy22}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn*

## Abstract

Disassembly is a crucial yet challenging step in binary analysis. While emerging neural disassemblers show promise for efficiency and accuracy, they frequently generate outputs violating fundamental structural constraints, which significantly compromise their practical usability. To address this critical problem, we regularize the disassembly solution space by formalizing and applying key structural constraints based on post-dominance relations. This approach systematically detects widespread errors in existing neural disassemblers' outputs. These errors often originate from models' limited context modeling and instruction-level decoding that neglect global structural integrity. We introduce Tady, a novel neural disassembler featuring an improved model architecture and a dedicated post-processing algorithm, specifically engineered to address these deficiencies. Comprehensive evaluations on diverse binaries demonstrate that Tady effectively eliminates structural constraint violations and functions with high efficiency, while maintaining instruction-level accuracy.

## 1 Introduction

Disassembly, the process of identifying instructions from raw byte sequences, forms the foundation of binary analysis [16, 31, 33]. Its accuracy is paramount, as numerous subsequent sophisticated analyses critically depend on its output. These include decompilation [6, 22, 37], data flow analysis [45, 47], program slicing [5], and binary code similarity detection [14, 35, 40, 41]. Errors introduced during disassembly can cascade, potentially invalidating entire analytical efforts.

Traditional disassembly algorithms contend with fundamental ambiguities inherent in binary code. These challenges include distinguishing code from data within the same memory space and precisely identifying instruction boundaries, especially in architectures featuring variable-length instructions. Industry-standard disassemblers such as IDA Pro, Ghidra, and Binary Ninja [13, 29, 38] employ sophisticated heuristics [48] to navigate these complexities. While often effective, their reliance on heuristics limits their adaptability and can lead to failures in corner cases [3, 8, 46] or obfuscations [24, 34].

Recent advancements in learning-based disassemblers [32, 43, 46] offer a promising data-driven paradigm, reducing dependence on handcrafted heuristics. These models, trained on extensive datasets of labeled binaries, first perform superset disassembly, which assumes all addresses as potential instruction starts and then categorizes them to find true code. They achieve high accuracy at the individual instruction level. However, a significant hurdle impedes their widespread adoption: **violation of fundamental constraints**. For example, a model might incorrectly identify a fall-through byte sequence after a valid instruction as non-code, leading to an output that represents an invalid execution trace. These violations render the disassembly practically unusable for downstream analyses, despite potentially high instruction-level metrics.

While individual instruction predictions might be locally accurate, a valid disassembly necessitates that choices for different byte sequences are consistent with each other. These disassembly choices exhibit complex interdependence, which can be broadly categorized into: *mutual exclusions*, where one disassembly choice invalidates another (e.g., overlapping instruction candidates cannot both be valid code), and *structural implications*, where the validity of one instruction choice necessitates or is predicated upon the validity of another due to inherent program structure (e.g., relationships dictated by control flow). Effectively modeling and enforcing the interdependence can significantly **regularize the solution space** of the inherently ill-posed disassembly problem, thereby enhancing the reliability of the results. Building on this principle, we present *Tady*, a novel approach designed to overcome key technical challenges in leveraging these structural constraints.

A primary challenge lies in **systematically characterizing and efficiently detecting violations of inter-instruction constraints.** While the concept of using interdependence to regularize disassembly is not new, prior approaches like Pdisasm [28] and D-Arm [44] typically relied on formulating explicit pairwise constraints between instruction candidates.

*Corresponding authors.

Although capable of capturing local relationships, this approach scales quadratically, becoming computationally intractable when applied to superset disassembly where the number of potential instructions can easily reach millions.

Our key insight is that the complex web of both mutual exclusion and structural implication constraints can be systematically and efficiently captured within the framework of **post-dominance relations** on a superset Control Flow Graph (CFG). In compiler theory, an instruction $A$ is said to post-dominate an instruction $B$ if every path from $B$ to any program exit point must pass through $A$. This structural property is pivotal: for instance, if $B$ is determined to be code, then $A$ must also be code, otherwise, executing $B$ would lead to executing data at $A$; conversely, if $A$ is not valid code, then $B$ cannot be code, as this would imply an execution path that eventually attempts to execute an invalid instruction at $A$.

Building upon this insight, *Tady* characterizes these global constraints based on post-dominance. It leverages the post-dominator tree (PDT, introduced in Section 2.2) derived from the superset CFG to detect violations of these structural constraints. This method allows for an efficient, linear-time algorithm to identify such inconsistencies, thereby avoiding the prohibitive cost of exhaustive pairwise comparisons.

Another significant challenge is **representing long-range interdependence and execution-aware context.** To satisfy mutual exclusion and structural implication constraints, a disassembler must understand relationships between instructions that can be distant, both in memory addresses and execution paths. Existing learning-based approaches often prove insufficient. Graph-based models like DeepDi [46], despite using message passing, typically have a limited receptive field. Sequence-based models like XDA [32] process bytes without inherent awareness of execution order semantics.

To tackle this, *Tady* utilizes a neural architecture that processes binaries by considering potential execution traces using a hybrid local-global attention mechanism. Specifically, it uses sliding window attention with special attention masks indicating execution reachability to capture local sequential information on the same trace, preventing focus on unrelated addresses. For global information, it adds a message-passing layer using an attention mechanism to pass information between jumps and calls, enabling the enforcement of long-range dependencies. This results in more contextually informed predictions aligned with CPU semantics.

The third challenge involves **enforcing consistency over probabilistic predictions.** Learning-based models inherently produce probabilistic outputs, assigning scores to potential instructions. However, a valid disassembly result must be a deterministic representation strictly adhering to the structural constraints. Simply thresholding instruction-level probabilities can lead to globally inconsistent results where individually plausible instructions collectively violate constraints.

*Tady* bridges this gap by incorporating a principled post-processing step. This framework leverages the previously characterized structural constraints (via the post-dominator tree) and employs a dynamic programming algorithm to prune inconsistencies from the model's probabilistic outputs. This reconciles neural scores with deterministic rules, ensuring a globally coherent and valid disassembly.

Our extensive evaluation on diverse datasets reveals a significant prevalence of structural constraint violations across disassembly tools. Neural disassemblers are particularly susceptible, exhibiting violations on most of the disassembly results. Surprisingly, even established rule-based disassemblers like IDA Pro and Ghidra are not immune, violating constraints on all binaries obfuscated with anti-disassembly techniques [24]. *Tady* significantly improves the consistency of disassemblers' outputs by completely eliminating all the structural constraint violations while maintaining high instruction-level accuracy.

Our primary contributions are as follows:

**Systematic Characterization of Structural Constraints.** We introduce a framework for characterizing structural constraints in disassembly and an efficient, PDT-based algorithm for detecting their violations without relying on the labels.

**Context-Aware Neural Disassembly Architecture.** We design a novel neural architecture that employs a hybrid attention mechanism, leading to more structurally informed disassembly results.

**Robust Enforcement of Deterministic Constraints.** We develop a principled post-processing technique to enforce the structural constraints. It applies to various disassemblers.

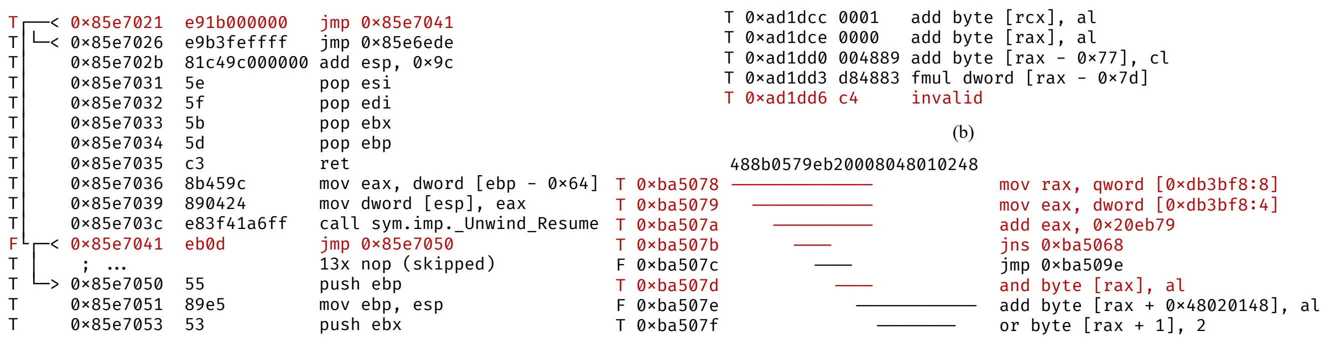
**Comprehensive Evaluation and Broader Impact.** Our extensive evaluation demonstrates that *Tady* maintains high accuracy while enforcing consistency. Furthermore, our error detection algorithm systematically locates many errors of widely-used disassemblers and disassembly datasets.

## 2 Structural Constraints

### 2.1 Constraint Violations

We identify three primary types of constraint violations. Each type is defined below, accompanied by an example identified within the labels provided by the x86-sok [31] dataset.

**Missing Post-Dominator (MPD).** This violation occurs when instruction $I_A$ identified as true code, but its post-dominator, instruction $I_B$, is missing from the disassembly. By definition of post-dominance, if $I_A$ is executed, then $I_B$ must also be executable. The incorrect labeling of $I_B$ thus constitutes a violation. Figure 1(a) illustrates an example from the binary `clang_m32_O0/ld.gold`. Here, the instruction at address `0x85e7021` is marked as true code. However, this unconditional jump instruction's target, `0x85e7041` (which is its post-dominator), is marked as non-code. This directly contradicts the post-dominance relationship implied by the jump instruction's semantics.



```
T┌──< 0×85e7021  e91b000000   jmp 0×85e7041
T│└─< 0×85e7026  e9b3feffff   jmp 0×85e6ede
T│    0×85e702b  81c49c000000 add esp, 0×9c
T│    0×85e7031  5e           pop esi
T│    0×85e7032  5f           pop edi
T│    0×85e7033  5b           pop ebx
T│    0×85e7034  5d           pop ebp
T│    0×85e7035  c3           ret
T│    0×85e7036  8b459c       mov eax, dword [ebp - 0×64]
T│    0×85e7039  890424       mov dword [esp], eax
T│    0×85e703c  e83f41a6ff   call sym.imp._Unwind_Resume
F└┌─< 0×85e7041  eb0d         jmp 0×85e7050
T │    ; ...                  13x nop (skipped)
T └─> 0×85e7050  55           push ebp
T     0×85e7051  89e5         mov ebp, esp
T     0×85e7053  53           push ebx
```

(a)

```
T 0×ad1dcc 0001   add byte [rcx], al
T 0×ad1dce 0000   add byte [rax], al
T 0×ad1dd0 004889 add byte [rax - 0×77], cl
T 0×ad1dd3 d84883 fmul dword [rax - 0×7d]
T 0×ad1dd6 c4     invalid
```

(b)

```
             488b0579eb20008048010248
T 0×ba5078 ──────────────                 mov rax, qword [0×db3bf8:8]
T 0×ba5079   ────────────                 mov eax, dword [0×db3bf8:4]
T 0×ba507a     ──────────                 add eax, 0×20eb79
T 0×ba507b       ────                     jns 0×ba5068
F 0×ba507c         ────                   jmp 0×ba509e
T 0×ba507d           ────                 and byte [rax], al
F 0×ba507e             ────────────       add byte [rax + 0×48020148], al
T 0×ba507f               ────────         or byte [rax + 1], 2
```

(c)

Figure 1: Example of Constraint Violations discovered in the labels provided by x86-Sok dataset. (a) Missing Post-Dominator. (b) Dead-End Sequence. (c) Overlapping Instructions.

**Dead-End Sequence (DES).** This violation arises when an instruction $I_A$ is identified as true code, yet an undecodable byte sequence $U$ post-dominates $I_A$. Since every path from $I_A$ to a program exit must pass through $U$, and $U$ itself is not valid code, execution of $I_A$ would lead to an invalid state. Figure 1(b) presents an example from the binary `clang_Of/libv8.so`. The instruction sequence starting from address `0xad1dcc` is labeled as code. This sequence, however, eventually leads to an undecodable byte sequence starting at `0xad1dd6`. This creates a dead-end, as execution cannot validly proceed through an undecodable address.

**Overlapping Instructions (OI).** This violation occurs when a disassembler proposes multiple distinct, valid instructions whose byte representations in memory overlap. The assumption that instructions do not overlap holds for most compiled code, with known exceptions typically limited to manually introduced lock prefixes via inline assembly or intentionally obfuscated code. A violation of this type usually indicates errors in the disassembly process. Figure 1(c) depicts an example from `clang_O3/libv8.so`. The upper portion shows the hexadecimal representation of raw bytes, with annotations indicating the span of candidate instructions. In this instance, the addresses `0xba5078`, `0xba5079`, `0xba507a`, `0xba507b`, and `0xba507d` are all marked as true code. However, their byte sequences overlap. Manual investigation reveals that only the instruction at `0xba5078` is the actual true code.

It is important to emphasize that a real ground-truth disassembly should not exhibit these violations, as they fundamentally contradict the post-dominance relationships inherent in control flow semantics and the standard structure of executable code. However, correct disassembly can be complex and involve many subtle cases. Consequently, the labels provided by many disassembly datasets may contain errors and might not represent the real ground-truth. To avoid confusion, we consistently refer to such dataset entries as `labels` rather than `ground-truth`. Furthermore, the presence of these violations in the labels implies errors. This observation can be leveraged to systematically locate errors and improve the quality of disassembly datasets.

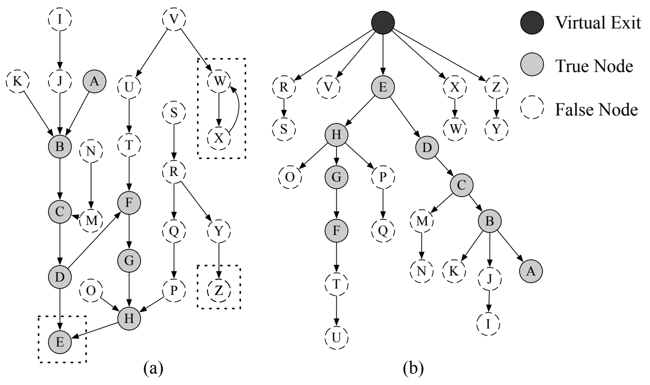


Figure 2: Example of a superset Control Flow Graph and its corresponding Post-Dominator Tree. (a) Superset Control Flow Graph. (b) Post-Dominator Tree.

## 2.2 Post-Dominator Tree

The post-dominance relation, crucial for the constraints discussed previously, is effectively characterized by the Post-Dominator Tree (PDT). In a PDT, nodes represent instructions from the Control Flow Graph (CFG), and the directed edges represent immediate post-dominance relationships.

A node $B$ is said to post-dominate node $A$ if every path from $A$ to an exit node in the CFG must pass through $B$. The immediate post-dominator of a node $A$ is its closest strict post-dominator. More formally, $B$ is the immediate post-dominator of $A$ if: (1) $B$ post-dominates $A$. (2) $B \neq A$. (3) There is no other node $P$ (where $P \neq A$ and $P \neq B$) such that $P$ post-dominates $A$ and $B$ post-dominates $P$.

Figure 2 provides an illustrative example of a CFG and its corresponding PDT. In the depicted CFG, node $A$ is post-dominated by nodes $B, C, D$, and $E$. Among these post-dominators, only node $B$ is the immediate post-dominator of $A$ according to the previous definition.

Constructing a PDT for a general CFG, particularly a superset CFG, which considers every address as potential instruction start and links all known control flow edges, presents unique challenges. Such CFGs may lack a single, common exit point and might not even form a connected graph. Therefore, a specialized construction process is necessary. Our PDT construction process is as follows.

**Handling Large-Scale Graphs with Weakly Connected Components (WCCs).** Superset CFGs can be exceptionally large, potentially comprising millions of nodes, making direct processing infeasible due to memory and computational constraints. To manage this scale, we first segment the input CFG into its WCCs. A WCC is a maximal subgraph where a path exists between any two nodes if edge directions are disregarded. Since WCCs are disjoint, they can be processed independently and sequentially. This modular approach significantly decreases the memory footprint and computational burden, allowing our analysis to scale to very large binaries. To restrict the size of each WCC, we do not connect the edges from the *call* instructions to their targets even if they can be determined statically. This approach restricts the sizes of the WCCs at function level, which is acceptable in practice. Otherwise, it is possible to find large WCCs consisting of almost all the functions in the binary.

**Establishing a Unified Exit Point for each WCC.** After decomposing the CFG into WCCs, each WCC is, by definition, connected (when viewed as undirected). However, a WCC may still possess multiple natural exit points, such as multiple `ret` instructions or program termination syscalls. To ensure that the post-dominance relation within each WCC forms a tree structure with a single root, we introduce a virtual exit node specific to that WCC. This virtual exit node becomes the sole successor for all nodes within the WCC that originally had an out-degree of zero.

**Managing Cycles with Strongly Connected Components (SCCs).** A SCC is a maximal subgraph where there is a directed path from any node to any other node within that SCC. Infinite loops or complex cyclic structures within a WCC present a challenge for post-dominance analysis, as nodes within such cycles might not have a clear path to an external exit if all their successors are also within the cycle. We resolve this by identifying the SCCs of the WCC. The WCC can then be conceptually regarded as a condensation graph, where each SCC is collapsed into a single node; this condensation graph is inherently acyclic.

Instead of connecting all original zero out-degree nodes of the WCC directly to its virtual exit, we refine this for terminal SCCs. A terminal SCC is one that has no outgoing edges to nodes outside of itself within the WCC. It has an out-degree of zero in the condensation graph of the WCC. For each such terminal SCC, we connect designated representative nodes from within that SCC to the WCC's virtual exit node. For our purposes, these representative nodes are the control-flow transfer instructions within the respective SCCs that create the loops, which are the jumps. This ensures that nodes within terminal loops are properly anchored in the PDT. For instance, in Figure 2, the rectangle enclosing $E$ and $Z$ are single node terminal SCC, which are directly linked to the virtual exit. Node $W$ and $X$ form a terminal SCC representing an infinite loop, and the jump instruction $X$ is chosen as the representative node to link to the virtual exit.

**PDT Generation.** With each WCC preprocessed as described, augmented with its own virtual exit node and with its terminal SCCs appropriately linked to this exit, we then compute the immediate post-dominators for all nodes within that WCC. This is achieved using a standard algorithm, such as the Lengauer-Tarjan algorithm [19] on the reversed graph, since it is originally used to calculate immediate dominator instead of immediate post-dominator. The collection of these individual trees (one for each WCC, rooted at its virtual exit) forms the overall PDT for the input CFG.

It is important to note that our PDT construction does not require a fully reconstructed or perfectly complete CFG; it can operate effectively even with partial control-flow information, such as when the targets of some indirect jumps or calls remain unresolved.

## 2.3 Detection of Structural Violations

The PDT derived from a superset CFG provides a powerful structure for efficiently validating disassembly consistency. A correctly disassembled program, when represented as a PDT, exhibits specific structural properties. Primarily, all nodes representing actual instructions (true nodes) must form a connected subtree rooted at the virtual program exit. This implies the following two fundamental conditions.

**Path Integrity.** For any true node, the entire path from that node to the PDT root must consist exclusively of true nodes. Consequently, no true node should be post-dominated by a false node (decodable but classified as false) or an invalid node (a byte sequence that is not decodable).

**Non-Overlapping Assumption.** Adhering to the standard assumption that instructions do not overlap, a node in the PDT cannot post-dominate multiple true child nodes if those children represent non-control-flow (NCF) instructions. If it did, it would imply that multiple distinct NCF instructions fall-through to the same subsequent instruction, which is only possible if they overlap in memory.

Violations of these properties appear as identifiable patterns within the PDT. Figure 3 illustrates these patterns. Our detection algorithm, presented in Algorithm 1, systematically identifies these patterns through a single traversal of the PDT and returns three error sets.

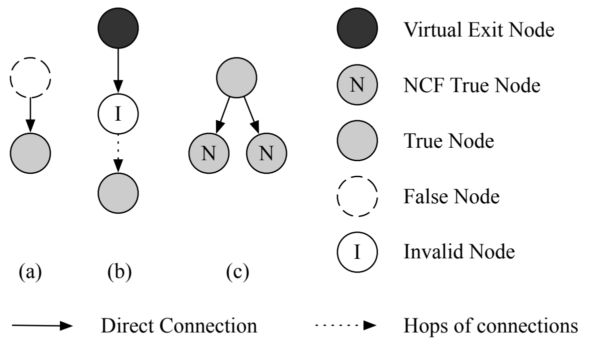


Figure 3: Patterns of violations. (a) Missing Post-Dominator. (b) Dead-End Sequence. (c) Overlapping Instructions.

**Missing Post-Dominator ($E_1$).** This violation occurs when a true node is post-dominated by a false node, as shown in Figure 3(a). The true node's path to the PDT root contains a false node, violating the *path integrity* property. The algorithm performs a BFS (lines 3, 8-26) to populate a *visitedSet* with all true nodes reachable from the root $r$ through paths of true nodes. After the BFS, any true node $v$ in the PDT that is not visited is added to $E_1$ (lines 27-31).

**Dead-End Sequence ($E_2$).** This occurs if a true instruction is post-dominated by an invalid node, as shown in Figure 3(b). The helper function (lines 4-7) is called for each NCF child $c$ of the root node (line 15). All such NCF nodes are invalid nodes, since otherwise they should be post-dominated by their subsequent instruction and cannot be immediate post-dominated by the virtual exit. This function performs a DFS from $c$ to collect all its true descendants to $E_2$.

**Overlapping Instructions ($E_3$).** This violation occurs if a node has multiple true NCF children, as shown in Figure 3(c). This implies that the instructions overlap with each other. During the BFS, if a node (that is not the root) has a true NCF child $c$, and the node has already visited a True NCF child, then $c$ is added to $E_3$ (lines 16-18).

By identifying these characteristic patterns, our approach detects violations of structural constraints from the PDT.

**Algorithm 1** Structural Violations Detection

```
1:  procedure ERROR DETECTION($PDT$, $r$)                    ▷ $r$ is the PDT root
2:      Initialize error sets $E_1 \leftarrow \emptyset$, $E_2 \leftarrow \emptyset$, $E_3 \leftarrow \emptyset$ and $visitedSet \leftarrow \emptyset$
3:      Start BFS from root $r$, add $r$ to $queue$ and $visitedSet$
4:      function DETECTDEADEND(node)
5:          Find all descendants of $node$ marked true via DFS
6:          Add them to $E_2$                                ▷ Dead-End Sequence
7:      end function
8:      while BFS $queue$ not empty do
9:          $node \leftarrow$ next node from $queue$
10:         if $node$ is True or is root then
11:             for each child $c$ of $node$ do
12:                 Add $c$ to $visitedSet$
13:                 if $c$ is non-control-flow then
14:                     if $node$ is root then
15:                         DetectDeadEnd($c$)
16:                     else if $c$ is True and $node$ visited True NCF child then
17:                         Add $c$ to $E_3$                 ▷ Overlapping Instructions
18:                     end if
19:                 else
20:                     if $c$ is True and $c \notin visitedSet$ then
21:                         Add $c$ to BFS $queue$
22:                     end if
23:                 end if
24:             end for
25:         end if
26:     end while
27:     for each node $v$ in $PDT$ do
28:         if $v \notin visitedSet$ then
29:             Add $v$ to $E_1$                             ▷ Missing Post-Dominator
30:         end if
31:     end for
32:     return $(E_1, E_2, E_3)$
33: end procedure
```

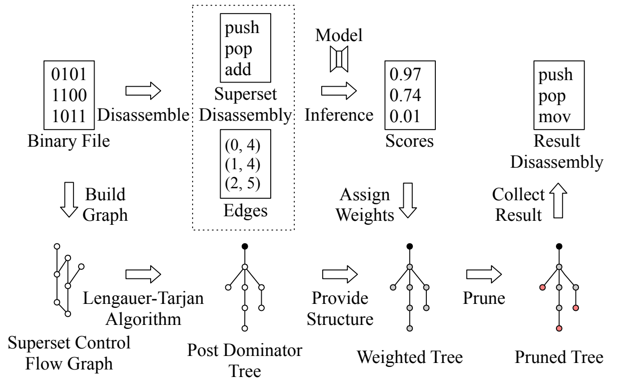


Figure 4: Overview of Tady.

## 3 Tady

Figure 4 shows our disassembler's workflow. We first conduct a superset disassembly over the executable section to extract instructions and their address connections. They are then fed into our model to assign scores to each address. Simultaneously, we construct a superset CFG connecting all instructions. The nodes are all instructions from the superset disassembly and the edges are the control flow edges, including fall-through, jump, conditional-jump and call.

We convert this CFG into a PDT with the steps described in Section 2.2. Each node is then assigned a weight based on the model's prediction, representing its likelihood of being a true instruction. After assigning the weights to PDT nodes, we apply our post-processing algorithm to regularize the solution. The inconsistent nodes are pruned and missing internal nodes are recalled. The nodes that remain on the tree after pruning form the final disassembly result.

In the following subsections, we first explain how our trace-aware model design incorporate necessary context to prevent constraint violations. Then, we introduce the algorithm for post-processing the output of model to enforce the result to satisfy the constraints.

## 3.1 Model Design

Satisfying the diverse constraints inherent in accurate instruction recognition, as detailed in Section 2, necessitates a model capable of processing both local and global contextual information. While non-overlapping assumptions primarily require local context regarding immediately adjacent instructions, constraints based on post-dominance demand a broader understanding of instruction interdependencies, often spanning considerable distances within the code.

Previous approaches exhibit limitations in addressing these multifaceted requirements. DeepDi [46], for instance, utilizes a Graph Neural Network (GNN) to propagate information along the execution trace. However, the GNN's receptive field, constrained by its layer-wise message passing, struggles to capture long-range dependencies critical for post-dominance constraints, where related instructions can be many hops apart. Conversely, XDA [32] processes raw byte sequences without explicitly leveraging control flow semantics. This makes it difficult for the model to effectively reason about control-flow-dependent constraints.

Our model introduces a novel architecture specifically designed to overcome these challenges. It integrates two key mechanisms. First, to capture fine-grained local context, we employ a masked local attention mechanism. This involves a transformer architecture equipped with a sliding window attention mechanism, notably augmented by an attention mask. This mask filters out instructions that, despite being within the sliding window, are not relevant to the task, thereby ensuring that local attention focuses only on semantically relevant preceding and succeeding instructions.

Second, to address the limitations of purely local attention and capture distant dependencies, we incorporate a global message passing layer. This layer facilitates information exchange between instruction blocks that may be spatially distant in the static code but are connected through control flow.

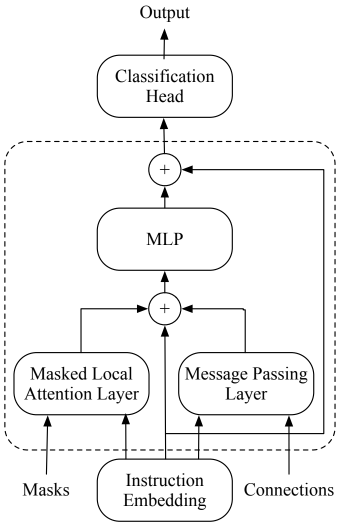


Figure 5: Overview of the Model Design.

This allows the model to effectively learn relationships critical for satisfying constraints like post-dominance, which often involve instructions far apart in the disassembly but proximate on potential execution paths.

As illustrated in Figure 5, our model processes input as follows: First, an RNN layer generates initial instruction-level embeddings from the input instruction sequence. These embeddings then pass through a series of transformer blocks. Each transformer block uniquely combines the masked local attention mechanism (for precise local context) and the global message-passing layer (for capturing long-range, control-flow-aware dependencies). Finally, a classification head processes the refined hidden states from the transformer blocks, outputting a score for each potential instruction start address, indicating its likelihood of being a true instruction.

### 3.1.1 Instruction Embedding

Instruction-level embeddings are generated with efficiency as a primary consideration, given that disassembly is a performance-critical task. We have empirically found that using raw bytes directly, rather than more complex tokenization over printable assembly, provides lightweight yet sufficiently effective features for this purpose.

Initially, the model is provided with the raw byte sequence of the input code and the decoded length of each instruction. Within the model, the specific byte sequence for each individual instruction is recovered using this information.

These bytes are then converted into numerical IDs. To preserve positional information within an instruction, the ID for a byte at offset i within that instruction is augmented by adding $256i$. Given that the maximum instruction length for x86-64 is 15 bytes, this scheme results in $256 \times 15 = 3840$ unique possible IDs. These numerical IDs are then passed through an embedding layer to create dense vector representations for each byte. This sequence of byte embeddings for each instruction is then processed by an RNN layer to compute the final instruction-level embedding.

### 3.1.2 Masked Sliding Window Attention

We introduce Masked Sliding Window Attention (MSWA) as a mechanism for capturing local context information. This approach constrains each instruction to attend only to a rele-

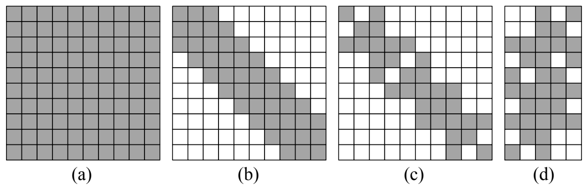


Figure 6: Sliding Window Attention with an Example Reachability Mask. (a) Full Attention. (b) Sliding Window Attention. (c) Masked Sliding Window Attention. (d) Flattened Mask.

vant subset of instructions within a sliding window, defined by a specific attention mask, as shown in Figure 6(c).

Furthermore, this masking strategy can be applied heterogeneously within a single multi-head attention layer: different attention heads can utilize distinct masks, allowing them to specialize in capturing different types of relationships simultaneously. For instance, an execution-order context is captured using a reachability mask, which limits attention to instructions within the same execution trace. Another application is an overlapping relation context, where the mask allows an instruction to attend to all other instructions it overlaps with.

Within each sliding window, as shown in Figure 6(b), an instruction's attention is restricted based on the applied mask. For computational efficiency, the sliding window pattern is typically flattened, as depicted in Figure 6(d), requiring much less computation comparing to the full attention as shown in Figure 6(a). The masked attention computation for a window of instructions is defined as:

$$\text{Attention}(Q, K_s, V_s) = \text{softmax}\left(\frac{Q(K_s)^T}{\sqrt{d_k}} + M_t\right) V_s$$

where Q, K, and V are the query, key, and value matrices respectively. The subscript $s$ indicates that keys and values are selected within the sliding window. The mask $M_t$ applies large negative values to disallowed attention pairs, effectively ensuring near-zero attention weights between instructions that do not satisfy the specific contextual relationship $t$.

Two variants of attention mask are used in our model. The **reachability mask** is derived from the connections information parsed from the instructions' semantics. To determine reachability between instructions, we follow each instruction's execution in parallel until it reaches maximum steps and collect reachable instructions along the way. The collection stops when encountering conditional jumps for simplicity. The **overlapping mask** is determined by identifying all instructions within the window that overlap with the current instruction, which can be calculated based on the given input length information as mentioned in Section 3.1.1.

Our MSWA framework offers significant advantages by enabling focused attention based on defined local contexts. It allows direct attention between all instructions on the same execution path within a window, rather than requiring multiple message-passing iterations as in GNNs. For instance, in a sequence $A \to B \to C$, the reachability mask allows C to directly attend to both A and B in a single layer.

The applied mask provides attention that is more focused compared to naive sequence models by filtering out instructions irrelevant to the specific local context being modeled, ensuring attention is concentrated on valid or pertinent relationships. The ability to assign different masks to different attention heads further enhances this focused attention, allowing the model to simultaneously learn diverse contextual features within a single layer.

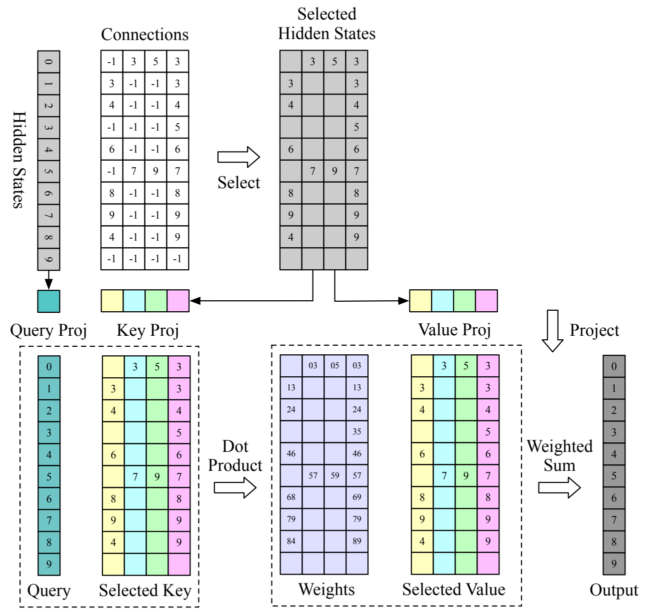


Figure 7: Global Message Passing With Selective Attention.

### 3.1.3 Global Message Passing

To effectively capture the global structure of a program, we introduce a **global message passing mechanism**. This mechanism leverages selective attention across various types of instruction relationships. For each instruction node $i$, we categorize its connections as follows:

**Must Transfer** ($M_i$): The unique successor instruction that is guaranteed to execute immediately after instruction $i$.

**May Transfer** ($T_i$): Both the potential jump target and the fall-through instruction of a conditional branch instruction.

**Next** ($N_i$): The instruction that sequentially follows instruction $i$ in the program's memory layout.

The total number of these connections for any given node $i$ is $C = |M_i| + |T_i| + |N_i| = 4$. These connections define an adjacency structure, represented as a tensor of shape $(B, L, C)$, where $B$ is the batch size and $L$ is the sequence length.

As illustrated in Figure 7, for each instruction $i$, represented by its hidden state $h_i$, we compute attention scores over its neighboring instructions using type-specific projections:

$$Q_i = W_q h_i$$

$$K_{j,t} = W_k^t h_j \quad \text{for } j \in \{M_i \cup T_i \cup N_i\}$$

$$V_{j,t} = W_v^t h_j \quad \text{for } j \in \{M_i \cup T_i \cup N_i\}$$

Here, $t$ denotes the specific connection type ($M$, $T$, or $N$). $W_q$, $W_k^t$, and $W_v^t$ are learnable projection matrices.

The attention weights $\alpha_{i,j,t}$ between instruction $i$ and a neighbor $j$ of type $t$ are computed using a softmax function:

$$\alpha_{i,j,t} = \text{softmax}\left(\frac{Q_i K_{j,t}^T}{\sqrt{d_k}}\right)$$

where $d_k$ is the dimension of the key vectors, used for scaling.

The updated hidden state $h'_i$ for instruction $i$ is calculated as a weighted sum of the value vectors from its neighbors:

$$h'_i = \sum_{t \in \{M,T,N\}} \sum_{j \in \text{type } t \text{ neighbors of } i} \alpha_{i,j,t} V_{j,t}$$

Some instructions lack certain connection types. For example, an unconditional jump has no *May Transfer* edges. We employ masked attention for them, setting the corresponding attention weights $\alpha_{i,j,t}$ to zero for non-existent connections, effectively preventing information flow through those paths.

This selective attention mechanism facilitates efficient information propagation across the program's control flow graph. It is effective for relaying information between distant instructions connected via jumps or calls. This contributes to maintaining global consistency in the result.

## 3.2 Pruning Algorithm

While our model's rich feature set helps satisfy constraints, it cannot guarantee full compliance with the hard constraints of valid disassembly. Post-processing is therefore essential to ensure validity of the final output. Since these constraints are fundamentally tied to post-dominance relation, we develop a post-processing algorithm based on PDT. As illustrated in Figure 2, our goal is to find a subset of nodes in the PDT that satisfies the properties discussed in Section 2.3, while maximizing the sum of their confidence scores. This can be viewed as finding the most probable self-consistent disassembly solution. We formulate this as a dynamic programming problem where we seek to find a maximum-weighted subtree rooted at the original PDT root. The problem naturally decomposes into sub-problems of finding optimal subtrees rooted at each child node. By solving these sub-problems recursively and combining their solutions, we obtain the globally optimal pruned tree. Our pruning algorithm consists of two phases.

**Weight Propagation.** The Weight Propagation phase, detailed in Algorithm 2, aggregates instruction-level confidence scores upward through the tree to compute subtree weights. This prevents naive greedy pruning that might discard entire subtrees due to a single negative-weighted node, even when the subtree contains many high-confidence instructions.

As shown in Algorithm 2, we perform a post-order depth-first traversal where each node's weight is updated only after processing all its children. The key insight is that we selectively aggregate only positive weights from children (lines 6-8), ensuring the pruning decisions maximize the overall confidence of the retained instructions. The final weight of each node combines its own score with the aggregated child weights, clamped to be non-negative (line 10).

**Result Collection.** The Result Collection phase, implemented in Algorithm 3, constructs the final pruned tree through a breadth-first traversal. Starting from the root, we build the pruned tree by selectively including nodes based on their propagated weights and types.

**Algorithm 2** Weight Propagation in Post-Dominator Tree

```
 1: procedure PROPAGATEWEIGHTS($tree, root$)
 2:     function POSTORDERVISIT($node$)
 3:         $childW \leftarrow 0$
 4:         for $child \in$ Children($node$) do
 5:             POSTORDERVISIT($child$)
 6:             if $weight[child] > 0$ then
 7:                 $childW \leftarrow childW + weight[child]$
 8:             end if
 9:         end for
10:         $weight[node] \leftarrow \max(0, weight[node] + childW)$
11:     end function
12:     POSTORDERVISIT($root$)
13: end procedure
```

**Algorithm 3** Pruning Post-Dominator Tree

```
 1: procedure PRUNETREE($tree, root$)
 2:     $prunedTree \leftarrow \emptyset$
 3:     $Q \leftarrow$ new Queue()
 4:     $Q$.push($root$)
 5:     while $Q$ is not empty do
 6:         $node \leftarrow Q$.pop()
 7:         if $weight[node] > 0$ or $node = root$ then
 8:             AddNode($prunedTree, node$)
 9:             $maxFT \leftarrow (0, \text{null})$
10:             for $child \in$ Children($node$) do
11:                 if not IsControlFlow($child$) then
12:                     if $weight[child] > maxFT.w$ then
13:                         $maxFT \leftarrow (weight[child], child)$
14:                     end if
15:                 else if $weight[child] > 0$ then
16:                     AddNode($prunedTree, child$)
17:                     AddEdge($prunedTree, node, child$)
18:                     $Q$.push($child$)
19:                 end if
20:             end for
21:             if $maxFT.n \neq$ null then
22:                 AddNode($prunedTree, maxFT.n$)
23:                 AddEdge($prunedTree, node, maxFT.n$)
24:                 $Q$.push($maxFT.n$)
25:             end if
26:         end if
27:     end while
28:     return $prunedTree$
29: end procedure
```

The algorithm maintains two key invariants: (1) Control-flow instructions are included only if they have positive weights (lines 15-19). (2) For non-control-flow children of each node, only the highest-weighted child is retained (lines 11-15, 21-25). This selective inclusion, combined with the breadth-first traversal order, ensures the resulting tree maintains all constraints while pruning away low-confidence or inconsistent instructions. The final pruned tree, returned on line 28, represents a valid disassembly solution that maximizes confidence scores while satisfying all structural constraints.

## 4 Evaluation

In this section, we evaluate four key research questions:
**RQ1 (Prevalence of Inconsistency).** What is the frequency and severity of structural constraint violations in disassemblers' results and datasets' labels?
**RQ2 (Disassembly Performance).** How well does our neural disassembler perform compared to other approaches?
**RQ3 (Efficiency).** How efficient are our model, the error detection and the pruning algorithms?
**RQ4 (Ablation Study).** How effectively do the aspects of our model design contribute to the overall performance?

### 4.1 Implementation and Setup

**Implementation.** For instruction decoding and the extraction of associated metadata, including byte length, control flow characteristics, and successor instructions, we utilize the MCDisassembler from the LLVM compiler infrastructure [17]. To optimize for processing efficiency, we directly use the internal representation of instructions rather than their printable, objdump-style disassembly output. The error detection and pruning algorithms are implemented in C++ and leverage the Boost Graph Library. To facilitate integration with Python-based workflows, these C++ implementations are equipped with Python bindings that expose results as NumPy arrays. Neural network models are implemented and trained using Flax [12], which operates on top of JAX [4]. For operational deployment, trained models are exported to the TensorFlow [27] SavedModel format. Serving is then managed by TensorFlow-Serving [30], with a Python client responsible for data preprocessing and gRPC communication with the serving endpoint. All experiments were conducted on a PC equipped with an Intel Core i9-12900K CPU, a single NVIDIA RTX A6000 Ada generation GPU, 64GB of RAM, and a 1TB NVMe SSD.
**Deep Learning Model Settings.** For the optimization of our deep learning model, we utilize the AdamW optimizer [26] with a learning rate of $1 \times 10^{-3}$. To address the issue of class imbalance in the training data, we employ Focal loss [23], a variant of Binary Cross-Entropy loss [9] designed to prioritize difficult examples. The specific hyper-parameters for the Focal loss are set to $\alpha = 0.8$ and $\gamma = 4.0$. Our model architecture consists of two transformer layers incorporating our custom attention mechanism. The hidden size of the transformer layers is 16, and the intermediate size is 32. We use a sliding window approach with a window size of 64 on both sides. We use four attention heads for our MSWA, three of them use reachability mask and the other one uses overlapping mask. The input sequences are chunked into pieces of length 8192; chunks shorter than this maximum are padded with zeros to ensure consistent input dimensions.
**Datasets.** Our experiments utilize several public benchmarks. For training, we primarily use Pangine [20], a dataset of 879 binaries compiled with clang (3.8, 6.0), gcc (5.4, 7.0), icc (19.1.1.219), and msvc-cl (19.26.28806) across various optimization levels (O0-O3, Os, Ofast) for x86 and x86-64 architectures. Its labels are derived from intermediate compilation results. The model is trained on this dataset for one epoch with a 9:1 train-validation split. The trained model is evaluated on Assemblage [25] (large-scale open-source Windows programs from GitHub, labels from PDB files) and x86-sok [31] (Linux binaries, including complex ones like openssl and mysqld, with labels from modified clang 6.0.0 and gcc 8.1). We excluded bap-corpora [2] as its components are covered by x86-sok. Additionally, we created rw, a real-world dataset mirroring DeepDi's [46] protocol, using the latest versions of their selected projects and x86-sok's modified compilers for labels and binaries. Acknowledging limitations in label generation (e.g., incomplete coverage from external linking, ambiguity of padding bytes), we follow ddisasm-WIS [7] by marking addresses without labels as `ignore`, excluding them from training and evaluation. All experiments focus on the `.text` section.

To assess robustness against obfuscation, we use OLLVM-14.0 [15] (LLVM IR level), Tigress [1] (source code level), and binobf [24] (binary-level anti-disassembly). For binobf, we use the 11 SPEC CPU 2000 binaries from the obf-benchmark [24] dataset. For OLLVM and Tigress, we use the quarks dataset [36], which includes variously configured obfuscated binaries. There is a obfuscation ratio controlling how many functions are obfuscated within the binary, and we choose the subset of the dataset where this ratio is 100%, indicating all the functions are obfuscated. Details of the quarks dataset's composition can be found in its documents.

Our final evaluation set includes 81,875 binaries from Assemblage, 3,997 from x86-sok, 879 from Pangine, 526 from rw, 1,298 from quarks, and 11 from obf-benchmark. To manage evaluation time, datasets with over 1,000 binaries are randomly sampled (1,000 binaries, seed 0); smaller datasets are used entirely. For rule-based disassemblers, a 60-second timeout per binary is enforced; binaries causing failures or timeouts are excluded from that disassembler's results.
**Baselines.** We compare our approach against several state-of-the-art rule-based and neural disassemblers. For rule-based systems, we include *IDA Pro (v9.1)* [13], a widely adopted commercial tool, from which we programmatically extracted instruction addresses using its SDK (idalib). We also evaluated *Ghidra (v11.3.2)* [29], a popular open-source suite, using pyghidra to dump instruction addresses. Additionally, we benchmarked against *ddisasm (v1.9.0, commit 11d6ba92)* [8], a research disassembler based on Datalog; this version has already incorporated improvements post-dating their Weighted Interval Scheduling (WIS) work [7], and we parsed its exported GTIRB files. Our comparison with neural network-based disassemblers includes two key systems. We evaluated *XDA* [32] by fine-tuning the pre-trained model provided by its authors on the instruction boundary classification task for 30

Table 1: Comparison of Disassembly Tools Across Datasets and Error Metrics (F: File Error Rate, B: Errors per 1MB).

| Dataset | Err | Labels F | Labels B | IDA F | IDA B | Ghidra F | Ghidra B | Ddisasm F | Ddisasm B | DeepDi F | DeepDi B | XDA F | XDA B | Tady F | Tady B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pangine | M | 0.03 | 0.6 | 0.01 | 0.0 | 0.01 | 0.1 | 0.00 | 0.0 | 0.94 | 60.9 | 1.00 | 7455.8 | 0.88 | 22.7 |
| | D | 0.00 | 0.0 | 0.01 | 0.0 | 0.01 | 0.1 | 0.00 | 0.0 | 0.60 | 11.2 | 0.99 | 483.3 | 0.00 | 0.0 |
| | O | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.88 | 58.0 | 1.00 | 591.2 | 0.76 | 24.1 |
| | N | 0.07 | 6.5 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.41 | 3.1 | 0.89 | 426.8 | 0.87 | 210.0 |
| | T | 0.10 | 7.1 | 0.02 | 0.1 | 0.01 | 0.1 | 0.00 | 0.0 | 0.97 | 133.3 | 1.00 | 8957.1 | 0.99 | 256.9 |
| Assemb | M | 0.08 | 30.5 | 0.02 | 0.3 | 0.00 | 0.0 | 0.00 | 0.4 | 0.46 | 30.6 | 1.00 | 8251.1 | 0.77 | 70.5 |
| | D | 0.00 | 0.0 | 0.01 | 0.1 | 0.01 | 0.3 | 0.02 | 0.5 | 0.16 | 6.1 | 0.83 | 181.6 | 0.00 | 0.0 |
| | O | 0.00 | 0.1 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.28 | 21.3 | 0.75 | 186.7 | 0.87 | 177.7 |
| | N | 0.22 | 9.4 | 0.02 | 0.3 | 0.00 | 0.0 | 0.01 | 0.2 | 0.07 | 1.2 | 0.50 | 82.2 | 0.46 | 49.3 |
| | T | 0.26 | 39.9 | 0.04 | 0.7 | 0.01 | 0.3 | 0.03 | 1.1 | 0.54 | 59.3 | 1.00 | 8701.5 | 0.99 | 297.5 |
| X86-Sok | M | 0.25 | 102.2 | 0.01 | 0.0 | 0.00 | 0.0 | 0.00 | 0.1 | 0.80 | 106.0 | 1.00 | 7414.6 | 0.70 | 51.5 |
| | D | 0.00 | 0.1 | 0.00 | 0.0 | 0.00 | 0.0 | 0.04 | 0.4 | 0.34 | 33.1 | 0.96 | 334.6 | 0.00 | 0.0 |
| | O | 0.00 | 0.1 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.66 | 72.9 | 0.88 | 447.0 | 0.49 | 44.6 |
| | N | 0.78 | 538.5 | 0.00 | 0.0 | 0.00 | 0.0 | 0.01 | 0.1 | 0.22 | 4.0 | 0.85 | 612.9 | 0.76 | 250.9 |
| | T | 0.82 | 640.9 | 0.01 | 0.0 | 0.00 | 0.0 | 0.05 | 0.5 | 0.90 | 216.0 | 1.00 | 8809.1 | 0.95 | 347.0 |
| RW | M | 0.27 | 65.8 | 0.00 | 0.0 | 0.02 | 0.1 | 0.00 | 0.0 | 0.83 | 345.6 | 0.99 | 6903.4 | 0.81 | 73.4 |
| | D | 0.01 | 0.8 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.55 | 33.8 | 0.86 | 550.3 | 0.00 | 0.0 |
| | O | 0.01 | 6.2 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.76 | 81.9 | 0.87 | 213.1 | 0.61 | 45.7 |
| | N | 0.67 | 487.2 | 0.00 | 0.0 | 0.00 | 0.0 | 0.02 | 0.1 | 0.31 | 4.1 | 0.83 | 484.9 | 0.82 | 228.4 |
| | T | 0.70 | 560.1 | 0.00 | 0.0 | 0.02 | 0.1 | 0.02 | 0.1 | 0.88 | 465.4 | 0.99 | 8151.7 | 0.95 | 347.5 |
| Obf-Ben | M | 1.00 | 12918.9 | 1.00 | 17.2 | 1.00 | 456.4 | 1.00 | 41.1 | 1.00 | 3879.5 | 1.00 | 23003.2 | 1.00 | 2487.0 |
| | D | 1.00 | 2129.7 | 1.00 | 50.7 | 1.00 | 488.8 | 1.00 | 186.6 | 1.00 | 330.8 | 1.00 | 2643.0 | 0.00 | 0.0 |
| | O | 0.73 | 2.3 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 1.00 | 1074.1 | 1.00 | 1241.8 | 1.00 | 430.3 |
| | N | 0.00 | 0.0 | 0.00 | 0.0 | 0.64 | 0.8 | 0.80 | 1.4 | 1.00 | 6.2 | 1.00 | 72.9 | 1.00 | 3.8 |
| | T | 1.00 | 15050.9 | 1.00 | 67.9 | 1.00 | 946.0 | 1.00 | 229.0 | 1.00 | 5290.6 | 1.00 | 26960.8 | 1.00 | 2921.1 |
| Quarks | M | 0.29 | 2.8 | 0.30 | 2.6 | 0.30 | 4.9 | 0.21 | 0.9 | 1.00 | 489.6 | 1.00 | 3448.9 | 0.96 | 332.0 |
| | D | 0.13 | 1.1 | 0.13 | 1.3 | 0.16 | 1.4 | 0.05 | 0.2 | 0.72 | 35.2 | 1.00 | 1753.9 | 0.00 | 0.0 |
| | O | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.00 | 0.0 | 0.98 | 134.8 | 1.00 | 339.1 | 0.86 | 104.9 |
| | N | 0.03 | 0.1 | 0.03 | 0.1 | 0.02 | 0.2 | 0.00 | 0.0 | 0.48 | 4.4 | 0.79 | 815.8 | 1.00 | 317.1 |
| | T | 0.32 | 4.0 | 0.33 | 4.0 | 0.33 | 6.5 | 0.25 | 1.1 | 1.00 | 663.9 | 1.00 | 6357.7 | 1.00 | 753.9 |

epochs, adhering to their original paper's setup. We also included *DeepDi* [46], for which we utilized the official model due to the absence of a publicly available implementation.

**Evaluation Metrics.** To evaluate the performance of our approach, we employ several metrics to assess both instruction-level accuracy and the frequency of constraint violations. For characterizing instruction-level accuracy, we utilize standard metrics: *Precision*, *Recall*, and the *F1 score*. These are calculated based on the reference labels and provide a comprehensive measure of how accurately individual instructions are identified. To quantify the frequency of constraint violations, we use two distinct metrics: *File-Level Error Rate*: This metric represents the proportion of files that contain at least one constraint violation relative to the total number of files analyzed. It provides an overview of how widespread errors are across the disassembly results. *Byte-Normalized Error Rate*: To account for the varying sizes of binary files, this metric characterizes errors by dividing the total number of detected errors by the total number of bytes in the code section of all analyzed files. This normalization allows for a more fine-grained comparison of error propensity, irrespective of file size. To make the number more readable, we report the average errors per 1MB$(1024 \times 1024)$, instead of per byte.

## 4.2 Violations of Constraints

This section evaluates the prevalence of constraint violations in dataset labels and disassembler outputs. Since our pruning algorithm eliminates all of the violations, we report the error rate before pruning. While our error detection algorithm can operate without boolean labels, we implemented a score-based approach to align with our pruning algorithm. For

Table 2: Errors Detected in Dataset Labels.

| Stats | Pangine | Assemb | X86-Sok | RW | Obf-Ben | Quarks |
|---|---|---|---|---|---|---|
| M | 391 | 294,670 | 135,156 | 14,801 | 160,287 | 2,825 |
| D | 0 | 351 | 85 | 185 | 26,424 | 1,122 |
| O | 12 | 689 | 192 | 1,397 | 28 | 4 |
| N | 4,423 | 90,760 | 712,429 | 109,577 | 0 | 133 |
| T | 4,826 | 386,470 | 847,862 | 125,960 | 186,739 | 4,084 |
| Files | 879 | 81,875 | 3,997 | 526 | 11 | 1,298 |

rule-based disassemblers and dataset labels, we assign a score of 1 to addresses labeled as instructions (true) and $-1$ to those labeled as non-instructions (false). For binary classification models, including ours and DeepDi, we use their output logits as scores. For XDA, a multi-class classification model, we use the probability of an address being an instruction start and apply an inverse sigmoid function to derive the score.

Table 1 presents the evaluation results for constraint violations, where **M** denotes *Missing Post-Dominator* (MPD), **D** signifies *Dead-End Sequence* (DES), and **O** represents *Overlapping Instructions* (OI). A significant portion of detected MPD errors involve instructions post-dominated by a `NOP` instruction not recognized as code. These instances often indicate missing padding bytes within a basic block or mark a virtual exit and are effectively dead code. We list these as **N** (NOP), separate them from other, more critical, MPD violations. **T** denotes total of all errors. The detailed statistics for error counts in dataset labels are shown in Table 2.

Several key conclusions emerge from these results:

**All dataset labels contain constraint violations.** This finding aligns with previous research [7], where errors were manually identified. Our approach systematically locates these violations without requiring manual intervention or another referencing tool's results. The errors often have common patterns, such as missing `NOP`s within basic blocks or missing jump targets, indicating false negatives. The errors not belonging to any such patterns usually indicate false positives. These findings can help identify bugs in the dataset construction scripts, enhancing the quality of the resulting datasets. Tady has more `NOP` related errors but fewer errors of other types than DeepDi. After investigation, we found that its training set Pangine wrongly labels many `NOP`s within basic blocks as false, resulting in the wrong behavior of the model. This shows the necessity of high-quality training sets.

**Rule-based disassemblers (IDA, Ghidra, ddisasm) still exhibit errors.** Although these tools are designed to enforce constraints, leading to relatively better performance, violations were identified in their outputs. Manual investigation verified these as genuine disassembler errors. This finding indicates that they might be tricked into invalid intermediate states without effective rollback mechanisms, particularly when encountering obfuscated code.

**Neural disassemblers struggle to enforce consistency through architectural design alone.** For instance, DeepDi does not prevent all overlapping instructions though utilizing overlapping edges in its design. Our model, Tady, successfully prevents DES but does not fully mitigate MPD and OI issues, this highlights the importance of a rule based regularization upon neural disassemblers' outputs.

**Obfuscation significantly increases consistency violations.** Code from the Obf-Benchmark and Quarks datasets caused all disassemblers to generate substantially more violations compared to other datasets. This indicates that the obfuscation techniques effectively confused the disassemblers, leading them to incomplete or incorrect intermediate states. This observation offers insight into why even rule-based disassemblers produce constraint violations.

## 4.3 Accuracy Evaluation

In this section, we evaluate the instruction-level performance of various disassemblers using labels provided by the datasets. While these dataset labels may contain errors, meaning absolute performance cannot be precisely determined, the relative performance metrics remain valuable for comparative analysis of the disassemblers.

We evaluated the disassemblers on both common and obfuscated binaries. For this experiment, in addition to the previously introduced *Tady* model (trained on the Pangine dataset), we constructed another model, named *TadyA*, which was trained for one epoch on the training portion of a composite dataset created from all available datasets to simulate a train-validation split. Specifically, this composite dataset was generated by sampling 2 files from obf-benchmark and 100 files from each of the other five datasets. We then performed a 9:1 train-validation split on this mixed dataset.

**Overall Performance.** As shown in Table 3, neural disassemblers, despite potentially lower precision, usually outperform rule-based disassemblers in terms of recall. Our *TadyA* model, trained on a small portion of the combined datasets, generalized well to the remaining data, often yielding the best performance among all evaluated disassemblers, achieving the best performance on three of the six benchmarks and performs competently on others. Furthermore, our *Tady* model, trained exclusively on the Pangine dataset, demonstrated robust performance on other datasets featuring unseen compilers and binaries, consistently outperforming its direct competitor, DeepDi, on all Linux benchmarks.

**Pruning.** In most cases, the pruning algorithm has a positive impact on the F1 score. This is particularly evident for XDA, which assigns appropriate weights in general but does not explicitly considers the interdependence between instructions. For other disassemblers, either internal nodes are recalled as positive or the successors of negative-weighted nodes are pruned. How the F1 score changes depends on the quality of the assigned weights. Since the purpose of the pruning algorithm is to enforce constraints, an improvement in accuracy is not guaranteed, though often observed.

**Anti-Obfuscation.** Notably, despite no prior training on ob-

Table 3: Comparison of Precision, Recall, and F1 Before and After Pruning for Various Datasets and Disassemblers.

| Dataset | Metric | State | IDA | Ghidra | Ddisasm | DeepDi | XDA | TadyA | Tady |
|---|---|---|---|---|---|---|---|---|---|
| Pangine | P | B | 1.0000 | **1.0000** | 1.0000 | 0.9985 | 0.9613 | 0.9988 | 0.9996 |
| | | A | **1.0000** | 0.9999 | 1.0000 | 0.9991 | 0.9893 | 0.9994 | 0.9999 |
| | R | B | 0.9884 | 0.9332 | 0.9996 | 0.9995 | 0.9630 | 0.9994 | **0.9999** |
| | | A | 0.9884 | 0.9332 | 0.9996 | 0.9998 | 0.9906 | 0.9990 | **0.9999** |
| | F1 | B | 0.9942 | 0.9654 | **0.9998** | 0.9990 | 0.9621 | 0.9991 | 0.9997 |
| | | A | 0.9942 | 0.9654 | 0.9998 | 0.9995 | 0.9900 | 0.9992 | **0.9999** |
| Assemb | P | B | 0.9959 | **0.9999** | 0.9996 | 0.9990 | 0.9747 | 0.9965 | 0.9980 |
| | | A | 0.9962 | **0.9999** | 0.9996 | 0.9997 | 0.9897 | 0.9979 | 0.9992 |
| | R | B | 0.9859 | 0.9559 | 0.9915 | 0.9954 | 0.9252 | **0.9956** | 0.9935 |
| | | A | 0.9864 | 0.9562 | 0.9916 | **0.9956** | 0.9679 | 0.9953 | 0.9939 |
| | F1 | B | 0.9909 | 0.9774 | 0.9955 | **0.9972** | 0.9493 | 0.9960 | 0.9957 |
| | | A | 0.9912 | 0.9776 | 0.9956 | **0.9976** | 0.9787 | 0.9966 | 0.9966 |
| X86-Sok | P | B | **0.9955** | 0.9950 | 0.9889 | 0.9704 | 0.9575 | 0.9907 | 0.9792 |
| | | A | **0.9954** | 0.9944 | 0.9890 | 0.9716 | 0.9795 | 0.9909 | 0.9791 |
| | R | B | 0.9860 | 0.9773 | **0.9999** | 0.9994 | 0.9584 | 0.9996 | 0.9989 |
| | | A | 0.9859 | 0.9788 | **0.9999** | 0.9985 | 0.9918 | 0.9996 | 0.9991 |
| | F1 | B | 0.9908 | 0.9861 | 0.9944 | 0.9847 | 0.9580 | **0.9951** | 0.9890 |
| | | A | 0.9906 | 0.9865 | 0.9944 | 0.9849 | 0.9856 | **0.9953** | 0.9890 |
| RW | P | B | **0.9963** | 0.9958 | 0.9856 | 0.9767 | 0.9531 | 0.9925 | 0.9843 |
| | | A | **0.9963** | 0.9958 | 0.9856 | 0.9786 | 0.9776 | 0.9925 | 0.9848 |
| | R | B | 0.9908 | 0.9227 | **0.9998** | 0.9981 | 0.9533 | 0.9995 | 0.9906 |
| | | A | 0.9908 | 0.9228 | **0.9998** | 0.9838 | 0.9911 | 0.9996 | 0.9906 |
| | F1 | B | 0.9936 | 0.9578 | 0.9926 | 0.9873 | 0.9532 | **0.9960** | 0.9874 |
| | | A | 0.9935 | 0.9579 | 0.9926 | 0.9812 | 0.9843 | **0.9961** | 0.9877 |
| Obf-Ben | P | B | 0.8452 | 0.6828 | 0.6945 | 0.8643 | 0.6641 | **0.9237** | 0.9040 |
| | | A | 0.8674 | 0.7106 | 0.7064 | 0.8702 | 0.8053 | **0.9008** | 0.8934 |
| | R | B | 0.0912 | 0.4465 | 0.1550 | 0.9317 | 0.8005 | **0.9783** | 0.9624 |
| | | A | 0.0894 | 0.4324 | 0.1514 | 0.8930 | 0.8221 | **0.9278** | 0.9197 |
| | F1 | B | 0.1646 | 0.5399 | 0.2534 | 0.8967 | 0.7260 | **0.9502** | 0.9323 |
| | | A | 0.1620 | 0.5376 | 0.2494 | 0.8815 | 0.8136 | **0.9141** | 0.9064 |
| Quarks | P | B | **1.0000** | 0.9982 | 0.9984 | 0.9948 | 0.9436 | 0.9975 | 0.9952 |
| | | A | **1.0000** | 0.9990 | 0.9983 | 0.9971 | 0.9849 | 0.9978 | 0.9966 |
| | R | B | **1.0000** | 0.9191 | 0.9982 | 0.9972 | 0.9700 | 0.9994 | 0.9958 |
| | | A | **0.9999** | 0.9433 | 0.9981 | 0.9880 | 0.9946 | 0.9995 | 0.9974 |
| | F1 | B | **1.0000** | 0.9570 | 0.9983 | 0.9960 | 0.9566 | 0.9984 | 0.9955 |
| | | A | **0.9999** | 0.9704 | 0.9982 | 0.9926 | 0.9897 | 0.9986 | 0.9970 |

fuscated data, *Tady* exhibited significant robustness against anti-disassembly obfuscators in the Obf-Benchmark dataset. It outperformed rule-based disassemblers by a large margin; for instance, ddisasm and IDA identified only a few instructions. Ghidra, while achieving better results than IDA Pro and ddisasm, required over an hour to disassemble these binaries. This finding aligns well with DeepDi's experiments [46].

**Indirect Jumps.** The robustness of our disassembler against indirect jumps was demonstrated with its performance against obfuscators. The Tigress obfuscator, employed in the Quarks dataset, implements several techniques that introduce complex indirect jumps, including *virtualization*, *Control Flow Flattening* (CFF), *mix1*, and *mix2*. To confirm that these obfuscations indeed introduce indirect jumps, we compared the ratio of indirect jumps in binaries before and after applying obfuscation. Our analysis revealed that indirect jumps were at least five times more frequent post-obfuscation. Specifically, CFF applied at the O0 optimization level increased the ratio of indirect jumps by a factor of 17.32. *Tady*'s accuracy remained high in these scenarios, achieving F1 scores above 0.999, which demonstrates its robustness against indirect jumps.

## 4.4 Efficiency Evaluation

This section evaluates the efficiency of our proposed disassembler, *Tady*, including the model and the subsequent pruning algorithm. Our evaluation consists of time/memory consumption, and a detailed breakdown of computational costs.

To create a representative dataset, we first sorted binaries in x86-sok by their code section size. We then sampled binaries by selecting from exponentially increasing size intervals, choosing up to three samples from each interval where available. The disassembly script, previously employed in our analyses, was run in headless mode. We recorded the execution time for each disassembler to facilitate a comparative efficiency analysis. Rule-based disassemblers were benchmarked on a single Core i9-12900K CPU, while neural network-based disassemblers utilized an NVIDIA A6000 Ada GPU.

**Disassembly Efficiency.** Figure 8 shows the relationship between code section size and disassembly time for various disassemblers. The results show that *Tady* achieves the second-fastest performance, surpassed only by DeepDi. Notably, *Tady* is significantly faster than the widely-used rule-based disassembler, IDA Pro.

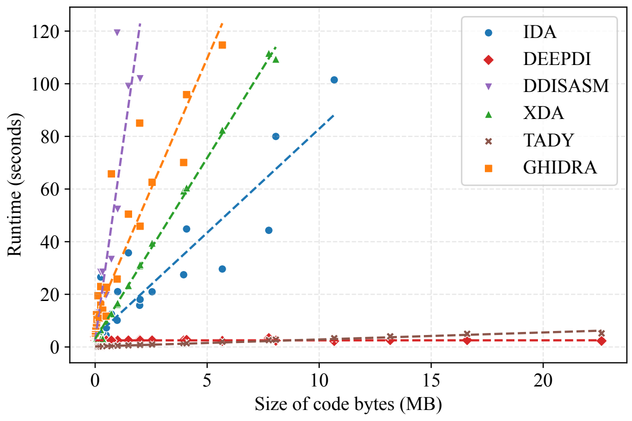


Figure 8: Run time comparison of the disassemblers.

To further quantify *Tady*'s efficiency and analyze its time consumption components, we conducted a rigorous benchmark. This involved recording a detailed breakdown of the time consumed by each disassembly step. These steps include the *Preprocessing* phase, which involves superset disassembly executed on a single CPU, and the *Model Inference* phase, which is performed on the GPU with the batch size and sequence length empirically set to 32 and 8192, respectively.

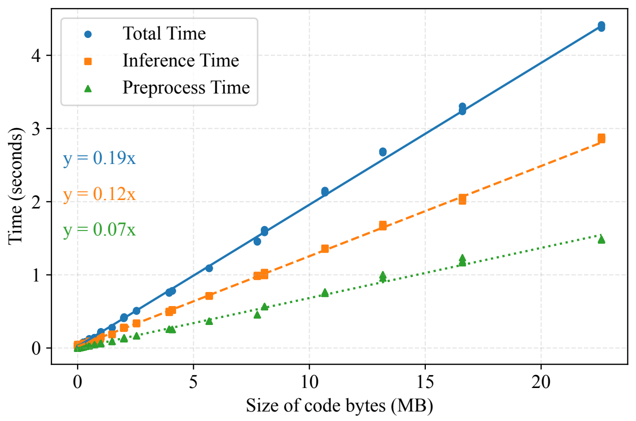


Figure 9: Time Consumption of Disassemble runtime.

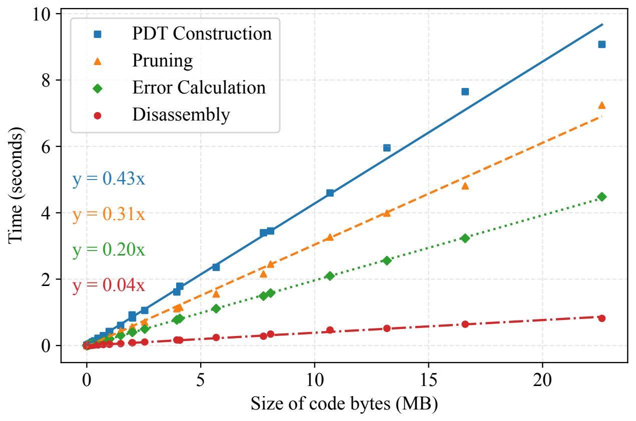


Figure 10: Time Consumption of the Algorithms.

As depicted in Figure 9, *Tady*'s total throughput is 5.26 MB/s. The CPU-bound preprocessing step accounts for a significant portion (37%) of the total time. Despite this, *Tady*'s overall performance remains considerably faster than IDA Pro, highlighting its substantial efficiency and practical applicability for large-scale binary analysis.

Beyond the disassembly phase including the preprocessing and model inference, we also evaluated the time and memory efficiency of our error detection and pruning algorithm.

**Time Consumption.** Figure 10 shows how the algorithms' execution time correlates with the binaries' code section size. The processing time exhibits a linear relationship with code size, which aligns perfectly with our theoretical analysis predicting linear time complexity for these algorithms in Appendix A. The pruning components are also efficient, achieving throughputs of 2.33 MB/s for the PDT construction process and 3.23 MB/s for the pruning algorithm itself. Even when factoring in the time for these post-processing steps, *Tady* maintains its high overall speed.

**Memory Usage.** Figure 11 shows the linear relationship between peak memory usage and the size of the code section. Processing the PDT at the WCC level effectively reduces memory requirements for tree construction. Since most WCCs are concentrated within a comparatively small size range, memory usage is primarily dominated by storing edges. Consequently, the memory needed to process individual WCCs is comparatively small. This finding underscores the effectiveness of our approach in managing WCC size by strategically omitting call edges during their construction.

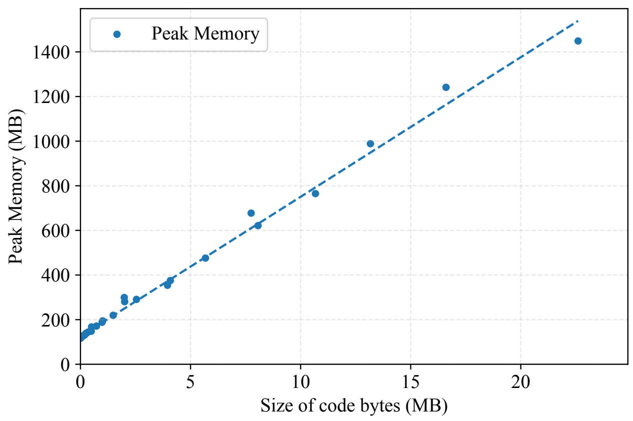


Figure 11: Peak Memory Usage for PDT Construction.

## 4.5 Ablation Study

To evaluate the impact of our core design choices, we conducted an ablation study focusing on **masked sliding window attention (MSWA)** and **message passing (MP)**. We assessed their effects on accuracy and constraint violations by training four model variants with different combinations of these mechanisms on the same dataset. Performance was evaluated on our test datasets (Tables 4 and 5).

Table 4: Ablation Study: Comparison of Model Configurations Across Datasets and Error Metrics (F: Total File Error Rate, B: Total Errors per 1MB).

| Dataset | MSWA + MP | | MSWA | | SWA + MP | | SWA | |
|---|---|---|---|---|---|---|---|---|
| | F | B | F | B | F | B | F | B |
| X86-Sok | 0.946 | 347.0 | 0.924 | 337.9 | 1.000 | 12326.9 | 1.000 | 2528.4 |
| Pangine | 0.991 | 256.9 | 0.973 | 223.6 | 1.000 | 14180.8 | 1.000 | 1639.7 |
| RW | 0.951 | 347.5 | 0.899 | 352.2 | 1.000 | 11970.7 | 0.990 | 1954.9 |
| Assemb | 0.987 | 297.5 | 0.999 | 267.5 | 1.000 | 17479.5 | 1.000 | 2587.1 |
| Obf-Ben | 1.000 | 2921.1 | 1.000 | 3779.7 | 1.000 | 17398.7 | 1.000 | 12524.6 |
| Quarks | 1.000 | 753.9 | 1.000 | 499.2 | 1.000 | 14566.6 | 1.000 | 3456.8 |

The variants, each trained for one epoch on the Pangine dataset, were: (1) Full Model: MSWA + MP; (2) MSWA only; (3) SWA + MP: Standard sliding window attention (SWA) with MP; (4) SWA only (Baseline). SWA utilizes a full attention mask within the sliding window, contrasting with MSWA's specialized reachability masks.

Results in Table 4 and Table 5 show that **MSWA** significantly outperforms SWA, consistently yielding higher ac-

Table 5: Ablation Study: Comparison of Model Configurations Across Datasets (Precision, Recall, F1-Score - Pruned).

| Dataset | MSWA + MP | | | MSWA | | | SWA + MP | | | SWA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | R | F1 | P | R | F1 | P | R | F1 | P | R | F1 |
| X86-Sok | 0.979 | 0.999 | 0.989 | 0.977 | 0.999 | 0.988 | 0.957 | 0.943 | 0.950 | 0.972 | 0.998 | 0.985 |
| Pangine | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 0.977 | 0.956 | 0.966 | 0.997 | 0.999 | 0.998 |
| RW | 0.985 | 0.991 | 0.988 | 0.982 | 0.991 | 0.987 | 0.965 | 0.936 | 0.950 | 0.978 | 0.991 | 0.984 |
| Assemb | 0.999 | 0.994 | 0.997 | 0.999 | 0.994 | 0.996 | 0.961 | 0.914 | 0.937 | 0.991 | 0.988 | 0.990 |
| Obf-Ben | 0.893 | 0.920 | 0.906 | 0.857 | 0.917 | 0.886 | 0.895 | 0.855 | 0.875 | 0.849 | 0.914 | 0.880 |
| Quarks | 0.997 | 0.997 | 0.997 | 0.996 | 0.999 | 0.997 | 0.951 | 0.863 | 0.905 | 0.985 | 0.995 | 0.990 |

curacy and fewer constraint violations. This indicates our reachability attention mask effectively injects execution semantics, enhancing performance. The **MP** mechanism's impact was conditional: detrimental with SWA but beneficial with MSWA, particularly improving accuracy on the obfuscation dataset while largely preserving performance elsewhere. This supports the idea that while local context often suffices for disassembly, global information from MP is advantageous for complex cases like obfuscated code.

## 5 Discussion

Our pruning and error detection algorithm assumes an instruction with fall-through semantics is immediately post-dominated by its subsequent instruction. This generally holds, except for signal-based hardware exceptions (e.g., segmentation faults, division-by-zero errors). If an instruction triggers such an exception, control may transfer to an exception handler rather than the next instruction in memory. This scenario, particularly when exceptions are deliberately engineered for obfuscation as seen in techniques like [34], represents a current limitation of Tady. High-level language exceptions such as try-catch blocks typically use mechanisms like function calls and do not violate this post-domination assumption.

## 6 Related Work

**Regularizing Disassembly Results.** Previous works have explored regularizing disassembly output using constraints. Pdisasm [28] used an iterative algorithm with control-flow and data-flow features. Ddisasm [7, 8] employed logic programming and later modeled disassembly as a weighted interval scheduling problem. D-Arm [44] used data-flow logic, framing it as a maximum weight independent set problem solved with approximate algorithms. These approaches often assign weights and solve optimization problems but can struggle with the complexity of intricate constraints. Our work aligns with this trend but introduces the post-dominator tree as a novel backbone for efficient error detection and constraint enforcement, addressing this complexity.

**Neural Disassemblers.** Neural network-based disassemblers aim to reduce manual rule creation through data-driven learning. Early efforts like ByteWeight [2] focused on function boundary identification. Subsequent models, such as XDA [32] using transformers and DeepDi [46] employing Graph Neural Networks, advanced instruction classification from raw bytes. While promising, these neural approaches often produce outputs violating fundamental disassembly constraints. Our work specifically addresses these consistency issues to facilitate more reliable neural disassembly.

**Assembly Models.** Various neural architectures have been applied to binary analysis. Instruction2Vec [18] and DeepVSA [10] used word2vec and RNNs for instruction embeddings, while PalmTree [21], jTrans [42], and HermesSim [11] utilized transformers or graph-based methods on intermediate representations. However, these models are unsuitable for our problem because they typically: (1) presuppose already accurate disassembly with known instruction boundaries, but we need to identify valid instructions from a superset; (2) process single, valid instruction sequences, whereas we must distinguish valid instructions among multiple potential traces; and (3) are not designed to process multiple potential traces in parallel, which is required in our superset scenario.

## 7 Conclusion

In this work, we systematically regularize the solution space of the disassembly problem with the structural constraints derived from post-dominance relation and the non-overlapping instructions assumption. We propose an efficient error detection algorithm based on the post-dominator tree over the superset CFG. This error detection algorithm successfully identified various errors from neural disassemblers, rule-based disassemblers and those disassembly datasets' labels without relying on ground truth. We mitigate the constraint violations by providing a better model design, exploiting both local sequential feature and global graph structure of the problem. In addition, we provide a pruning mechanism to post-process the output of the model and completely eliminate the violations from the final disassembly result. Our proposed method can serve as a general post-processing step that enhances the usability of all neural network-based disassemblers.

## Acknowledgments

We would like to sincerely thank all the reviewers for their insightful feedback that greatly helped us to improve this paper. Additionally, special thanks are extended to Miaoqian Lin for her invaluable comments. The authors from Institute of Information Engineering are supported in part by NSFC (U24A20236, 92270204) and CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118). The authors from Tsinghua University are supported in part by NSFC (U24A20337) and the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd..

## Ethical Considerations

Our research was conducted with careful consideration of ethical implications, following the principles outlined in The Menlo Report and USENIX Security's ethical guidelines. We identified and analyzed potential impacts on all stakeholders, including end users, system administrators, and security practitioners. Our methodology prioritized minimizing risks while maximizing benefits to the security community. We maintained compliance with relevant terms of service and legal requirements throughout the study. We acknowledge the dual-use potential of our findings and have carefully balanced research transparency with potential misuse concerns.

## Open Science

In accordance with USENIX Security's open science policy, the research artifacts associated with this paper are made publicly available at https://doi.org/10.5281/zenodo.15541311, including datasets, models, and source code. The latest version of the code will be maintained and updated at https://github.com/5c4lar/tady.

## References

[1] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 189–200. ACM, 2016.

[2] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 845–860. USENIX Association, 2014.

[3] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax, 2018.

[5] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural Static Slicing of Binary Executables. In *1997 International Conference on Software Maintenance (ICSM '97), 1-3 October 1997, Bari, Italy, Proceedings*, page 188. IEEE Computer Society, 1997.

[6] Cristina Cifuentes and K. John Gough. Decompilation of Binary Programs. *Softw Pract Exp*, 25(7):811–829, 1995.

[7] Antonio Flores-Montoya, Junghee Lim, Adam Seitz, Akshay Sood, Edward Raff, and James Holt. Disassembly as Weighted Interval Scheduling with Learned Weights. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3033–3050. IEEE Computer Society, April 2025.

[8] Antonio Flores-Montoya and Eric M. Schulte. Datalog Disassembly. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1075–1092. USENIX Association, 2020.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[10] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1787–1804. USENIX Association, 2019.

[11] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *USENIX Security Symposium*. USENIX Association, 2024.

[12] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX. http://github.com/google/flax, 2024.

[13] Hex-Rays. IDA Pro. https://hex-rays.com, 2025.

[14] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison. *IEEE Trans. Software Eng.*, 47(6):1241–1258, 2021.

[15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM - Software Protection for the Masses. In Paolo Falcarin and Brecht Wyseur, editors, *1st IEEE/ACM International Workshop on Software Protection, SPRO 2015, Florence, Italy, May 19, 2015*, pages 3–9. IEEE Computer Society, 2015.

[16] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In Daniel Große and Rolf Drechsler, editors, *Methoden Und Beschreibungssprachen Zur Modellierung Und Verifikation von Schaltungen Und Systemen, MBMV 2017, Bremen, Germany, February 8-9, 2017*, pages 51–52. Shaker Verlag, 2017.

[17] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, San Jose, CA, USA, 2004. IEEE Computer Society.

[18] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seungho Lim, Sung Hoon Baek, and Ki-Woong Park. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN. *Applied Sciences*, 9(19):4086, 2019.

[19] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[20] Kaiyuan Li, Maverick Woo, and Limin Jia. On the generation of disassembly ground truth and the evaluation of disassemblers. *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2020.

[21] Xuezixiang Li, Qu Yu, and Heng Yin. PalmTree: Learning an assembly language model for instruction embedding. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[22] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: An attention-based neural decompiler. *Cybersecurity*, 4(1):5, 2021.

[23] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007, 2017.

[24] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. Association for Computing Machinery.

[25] Chang Liu, Rebecca Saul, Yihao Sun, Edward Raff, Maya Fuchs, Townsend Southard Pantano, James Holt, and Kristopher K. Micinski. Assemblage: Automatic binary dataset construction for machine learning. *ArXiv*, abs/2405.03991, 2024.

[26] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[28] Kenneth A. Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, X. Zhang, and Zhiqiang Lin. Probabilistic disassembly. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198, 2019.

[29] National Security Agency. Ghidra. `https://github.com/NationalSecurityAgency/ghidra`, 2025.

[30] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. TensorFlow-serving: Flexible, high-performance ML serving. *ArXiv*, abs/1712.06139, 2017.

[31] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun 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)*, pages 833–851, 2020.

[32] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Sekhar Jana. XDA: Accurate, robust disassembly with transfer learning. *ArXiv*, abs/2010.00770, 2020.

[33] Marius Popa. Binary code disassembly for reverse engineering. *Journal of Mobile, Embedded and Distributed Systems*, 4:233–248, 2012.

[34] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, 2007.

[35] Meng Qiao, Xiaochuan Zhang, Huihui Sun, Zhen Shan, Fudong Liu, Wenjie Sun, and Xingwei Li. Multi-level cross-architecture binary code similarity metric. *Arabian Journal for Science and Engineering*, 46:8603–8615, 2021.

[36] Quarkslab. Diffing_obfuscation_dataset. https://github.com/quarkslab/diffing_obfuscation_dataset, 2025.

[37] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. Loop rerolling for hardware decompilation. *Proceedings of the ACM on Programming Languages*, 7:420–442, 2023.

[38] Vector 35 Inc. Binary Ninja. https://binary.ninja, 2025.

[39] VMProtect Software. VMProtect. https://vmpsoft.com/vmprotect/overview, 2025.

[40] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xiangwei Xiao. CLAP: Learning transferable binary code representations with natural language supervision. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[41] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xiangwei Xiao. CEBin: A cost-effective framework for large-scale binary code similarity detection. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[42] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jTrans: Jump-aware transformer for binary code similarity detection. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[43] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Differentiating code from data in x86 binaries. In *ECML/PKDD*, 2011.

[44] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and X. Zhang. D-ARM: Disassembling ARM binaries by lightweight superset instruction interpretation and graph modeling. *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2391–2408, 2023.

[45] Li Yong-cheng. Program understanding approach for binary code based on data flow analysis. *Computer Engineering*, 2010.

[46] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *USENIX Security Symposium*, pages 2709–2725. USENIX Association, 2022.

[47] Wu Zhi-yong. Method based on data flow analysis to understanding binary program. *Computer Engineering and Applications*, 2010.

[48] Li Zhoujun. Disassembly method based on control flow refining. *Journal of Tsinghua University*, 2011.

## Appendix

## A Complexity Analysis

Our pre-processing step involves a superset disassembly, where decoding occurs at every possible address. Although this step can be parallelized or GPU-accelerated, we observed it does not represent a computational bottleneck unless printable disassembly output is required. Decoding to the internal representation is efficient, even for large binaries, and exhibits linear time complexity, $O(L)$, where $L$ is the size of the binary.

For our model, each instruction attends to a fixed-size local context (via a sliding window) and a fixed number of global connections. Consequently, the computational cost scales linearly with the sequence length, $O(S)$, where $S$ is the number of instructions in the sequence. This linear scaling enables the model to process long sequences effectively. The sliding window size determines the number of neighboring instructions to which each instruction attends, directly influencing the total computation in a linear fashion.

The PDT construction process comprises several key steps: WCC computation, SCC computation, and the calculation of immediate post-dominators within each WCC using the Lengauer-Tarjan algorithm.

Classical algorithms for WCC and SCC detection have a complexity of $O(N+E)$, where $N$ is the number of nodes (instructions) and $E$ is the number of edges in the Control Flow Graph CFG. In our specific CFGs, each node has at most two outgoing edges (for conditional branches), meaning $E \leq 2N$. Therefore, the complexity for WCC and SCC computation simplifies to $O(N)$.

The Lengauer-Tarjan algorithm for computing immediate post-dominators exhibits a time complexity of $O(E \cdot \alpha(N))$, where $\alpha$ is the extremely slowly growing inverse Ackermann function [19]. Given that $E \leq 2N$ in our case, and $\alpha$ grows so slowly it is considered nearly constant for practical input sizes, this complexity is effectively linear.

Since we perform these computations within individual WCCs, and call edges are not connected during this phase (resulting in smaller WCCs), the actual runtime for PDT construction is determined by the size of the largest WCC and remains, in practice, linear with respect to the total number of nodes. Thus, the overall time complexity for PDT construction can be considered $O(N)$.

Following PDT construction, the error detection and pruning algorithms involve straightforward traversals of the PDT. Error detection requires a single pass. Pruning utilizes two passes: one for weight propagation up the tree and another for result collection down the tree. In a tree structure like the PDT, each node has at most one parent. Therefore, these traversals are also $O(N)$.

Based on the preceding analysis, the entire Tady workflow, including the error detection and pruning algorithms (as discussed in Section 2), demonstrates an overall linear time complexity and is expected to scale efficiently with the size of the input binaries.

## B Evaluation on Commercial Obfuscator

In addition to open-source obfuscators, we evaluated the disassemblers against a prominent commercial obfuscator, VMProtect [39]. This obfuscator supports several protection methods, including mutation, virtualization, or "Ultra" (a combination of the previous two). Mutation obfuscates the binary using instructions from the same Instruction Set Architecture (ISA). Virtualization, on the other hand, introduces a virtual machine and encodes the protected function into byte sequences specific to that virtual machine. Such potent obfuscation techniques, particularly virtualization, are beyond the scope of our current work. Consequently, our evaluation focused on the mutation capabilities of VMProtect. To simplify the evaluation and maintain a focus on static analysis, we disabled VMProtect's "Pack the output file" option. This option typically compresses the output binary, with decompression occurring only at runtime, effectively acting as an anti-static analysis measure. Analyzing such packed binaries is outside the scope of our current research. Our primary interest in this work is the analysis of statically obfuscated code.

Under this configuration, we obfuscated an example binary provided with the VMProtect Demo: `Licensing/BCB`. Recognizing that obfuscation patterns are often repetitive, we focused our efforts by obfuscating only the first protected function, `btTryClick`, and then manually analyzing it to establish the ground truth for this function. The results for this test case are presented below:

Our manual investigation of this obfuscated binary revealed a comparatively straightforward obfuscation methodology: the basic blocks of the protected function were scattered to distant locations in memory. Apart from this scattering, the instructions themselves appeared standard. However, IDA and Ghidra failed to correctly identify the function's entry point, resulting in exceptionally low recall. In contrast, DeepDi, ddisasm, and our *TadyA* model, which utilize superset disassembly incorporating global graph information (due to control flow edges between these spatially distant instructions), successfully recalled them. *Tady* missed two instructions, but successfully recalled them after pruning. XDA, despite achieving higher recall than IDA Pro, was still constrained by its reliance on local sequential context. However, applying our post-processing algorithm to XDA effectively propagated information along the execution trace, boosting its recall from 0.789 to 0.921. This serves as a clear example of how our post-processing algorithm can enhance model performance.

Table 6: Performance on the VMProtect Example case.

| Disassembler | P | R | F1 |
|---|---|---|---|
| IDA | 1.000 | 0.013 | 0.026 |
| Ghidra | 0.000 | 0.000 | 0.000 |
| DeepDi | 1.000 | 1.000 | 1.000 |
| Ddisasm | 1.000 | 1.000 | 1.000 |
| TadyA | 1.000 | 1.000 | 1.000 |
| Tady | 1.000 | 0.974 | 0.987 |
| Tady(pruned) | 1.000 | 1.000 | 1.000 |
| XDA | 0.896 | 0.789 | 0.839 |
| XDA(pruned) | 0.972 | 0.921 | 0.946 |

## C Generalizability to Unseen Settings

To further evaluate our models' ability to generalize to unseen settings, we conducted additional experiments. While their ability to handle unseen binaries can be inferred from the preceding experiments, here we specifically investigate generalization to unseen compilers and optimization levels. We explicitly splitted the Pangine dataset (used for training in previous experiments) by compiler and optimization level. We then conducted cross-validation to assess whether a model trained under one specific setting (e.g., a particular compiler and optimization level) could generalize to others. Specifically, we selected two compilers (clang-6.0.0 and gcc-7.5.0) and four optimization levels (O0-O3). For each combination, we trained the *Tady* model for five epochs on the corresponding dataset partition. The increased epoch count was chosen to compensate for the smaller size of these individual training datasets. Detailed results are omitted for brevity, as the models generally generalized well across settings, achieving F1 scores close to 1.0 when tested on configurations different from their training set. The only notable exception was the model trained on clang-6.0.0 with O0 optimization; while its precision remained near 1.0, its recall on other settings was slightly lower, but still achieved around 0.98. These results demonstrate *Tady*'s robust generalization capabilities.