# Neurosymbolic Disassembly

Zhaoqi Xiao
University of California, Riverside
Riverside, California, USA
zxiao033@ucr.edu

Yeqi Fu
National University of Singapore
Singapore, Singapore
yeqi.fu@comp.nus.edu.sg

Lambang Akbar Wijayadi
National University of Singapore
Singapore, Singapore
lambang@comp.nus.edu.sg

Zhenkai Liang
National University of Singapore
Singapore, Singapore
liangzk@comp.nus.edu.sg

Heng Yin
University of California, Riverside
Riverside, California, USA
heng@cs.ucr.edu

## ABSTRACT

The disassembly of binary programs is a fundamental yet challenging task in the field of security, particularly when addressing non-standard and complex standard binaries. Logic-based disassemblers often perform poorly in such settings, yielding either low recall or failing to disassemble the binary altogether. While neural-based approaches are more robust, they typically require extensive additional data for training on specific binary formats or instruction set architectures (ISAs). In this paper, we present NSDA, a neurosymbolic disassembler powered by Logic Tensor Networks (LTN). NSDA combines the capacity of neural networks to learn latent representations with the rigor of logical rules to constrain inference, thereby integrating both learning and logical reasoning. Requiring neither training data nor pre-trained models, NSDA outperforms both logic-based and neural disassemblers across various settings with minimal overhead. We also propose Segmented NSDA, a variant designed to handle large-scale binaries, which successfully disassembles Chromium and provides measurable improvements in accuracy.

## CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

## KEYWORDS

Neurosymbolic, Disassembly

**ACM Reference Format:**
Zhaoqi Xiao, Yeqi Fu, Lambang Akbar Wijayadi, Zhenkai Liang, and Heng Yin. 2026. Neurosymbolic Disassembly. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (CCS '26)*. ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Disassembly is a foundational yet challenging problem in cybersecurity, serving as the prerequisite for many downstream analyses such as malware detection, vulnerability discovery, binary rewriting, and reverse engineering [3, 12, 34, 40]. In practice, security analysts must first disassemble a binary to recover instruction boundaries, construct control-flow graphs, and enable higher-level analyses including decompilation and behavioral reasoning.

**Problem Statement.** Given a binary executable with incomplete or unreliable metadata, the disassembly task requires classifying memory regions as *code* or *data* while preserving global control-flow consistency. This task is inherently difficult because disassembly decisions are *globally coupled*: misclassifying a single block may invalidate large portions of the recovered control-flow graph. These challenges are particularly acute for *non-standard binaries*, where conventional assumptions about binary layout and control flow no longer hold.

For standard binaries such as ELF, PE, and Mach-O, well-defined formats with explicit headers, separated code and data sections, and recognizable control-flow metadata allow mature disassembly tools (including `objdump` [18], `Ghidra` [1], and `IDA Pro` [2]) to achieve high accuracy. In contrast, binaries deployed in Internet of Things (IoT) and Industrial Control Systems (ICS) environments frequently adopt custom or undocumented formats. Such binaries often lack reliable headers, interleave code and data within executable regions, and exhibit unconventional control-flow patterns. These characteristics undermine the assumptions embedded in traditional disassembly tools, causing them to miss valid code regions or conservatively classify executable bytes as data [7]. Vendor-specific variations further exacerbate the problem, rendering manual rule customization impractical at scale.

Existing disassembly techniques broadly fall into two categories. *Symbolic disassembly* tools, including `objdump` [18], `Ghidra` [1], `IDA Pro` [2], `Radare2` [43], `DDisasm` [14], and WIS [13], encode expert knowledge as deterministic heuristics or logical constraints. While highly effective on standard formats, these approaches are brittle when their assumptions are violated, often leading to unsatisfiable constraints, missed entry points, or premature termination.[1]

More recently, *neural disassembly* techniques such as `XDA` [33], `DeepDi` [51], `Loadstar` [7], and `DISA` [47] have leveraged deep learning models to classify code and data directly from raw bytes or instruction sequences. These methods improve robustness on certain non-standard binaries but rely heavily on large, architecture-specific labeled datasets. Adapting them to new binary formats or

[1]WIS [13] incorporates classical machine learning to tune rule weights, but does not employ neural representation learning.


*CCS '26, November 15-19, 2026, The Hague, The Netherlands.*

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

instruction set architectures (ISAs) typically requires costly data collection and retraining. Moreover, fine-tuning on non-standard binaries can degrade accuracy on standard binaries, revealing a tension between specialization and generalization [7].

Taken together, neither symbolic nor neural approaches alone provide a satisfactory solution across the full spectrum of real-world binaries. Symbolic disassemblers offer strong structural guarantees but lack flexibility, whereas neural disassemblers are adaptable but data-hungry and difficult to constrain.

**Key Insight.** We observe that binary disassembly can be naturally formulated as a *global constraint optimization problem*, in which code–data classification, control-flow consistency, and disassembly validity must be optimized jointly rather than independently.

Motivated by this insight, we explore a *neurosymbolic* approach to disassembly. Neurosymbolic AI [11, 39] integrates symbolic reasoning with neural representation learning in a unified optimization framework, enforcing logical constraints *during* learning rather than as post hoc filters. This paradigm is well-suited to disassembly, an inherently structural task with partial observability and strong global invariants. A neurosymbolic disassembler can reason over expert-derived logical rules while simultaneously learning latent patterns from binary features, enabling adaptation to previously unseen binaries without manual rule customization or retraining on labeled datasets.

We present NSDA, a neurosymbolic disassembly framework built on Logic Tensor Networks (LTN) [5]. NSDA leverages Ghidra [1] to obtain an initial, conservative partitioning of memory into code and data blocks. It then jointly optimizes a neural classifier and a set of eight architecture-agnostic logical rules encoding control-flow consistency and negative evidence (e.g., failed disassembly and high zero-byte density). Through an iterative refinement process, newly identified code blocks are fed back into Ghidra for re-disassembly, forming a closed reasoning loop that progressively improves global consistency. Furthermore, to accommodate large-scale binaries, we introduce *Segmented* NSDA, a variant specifically designed for the efficient disassembly of binaries at scale.

Our evaluation shows that NSDA improves code–data discrimination by up to 27% over the Ghidra baseline on challenging non-standard binaries, while also consistently enhancing accuracy on standard x86_64, ARM32 and MIPS binaries. These gains are achieved without any labeled training corpus or offline pretraining, and with modest computational overhead (7s–50s). Compared to neural-only approaches such as Loadstar [7], NSDA provides stronger generalization across architectures and binary formats by construction.

We summarize our contributions as follows:

(1) We introduce a neurosymbolic formulation of binary disassembly that tightly integrates neural representation learning with symbolic reasoning, enforcing global logical constraints during optimization rather than relying on ad hoc combinations of heuristics and classifiers.
(2) We design and implement NSDA, an architecture-independent disassembly system supporting both standard and non-standard binaries. Extensive evaluation on ARM32 and MIPS binaries compiled with gcc [15] and LLVM [23], as well as real-world PLC binaries [7], demonstrates that NSDA significantly outperforms purely symbolic and purely neural baselines while remaining practical.
(3) We release a new dataset of standard ARM32 and MIPS binaries derived from coreutils [17], compiled with multiple toolchains, along with the source code of NSDA, to facilitate future research on disassembly and binary analysis.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Disassembly

Disassembly typically begins with *instruction boundary identification* (IBI) [14], which determines the start and end of each instruction. This task is closely related to the *code–data separation* problem studied by Loadstar [7]. However, Loadstar focuses exclusively on the ARM architecture, where instructions are uniformly 4-byte aligned, simplifying instruction parsing once code regions are identified. In contrast, this work considers IBI as the core disassembly task across architectures and binary formats.

Some disassemblers, such as DDisasm [14], additionally perform *symbolization* to support downstream tasks like binary rewriting. Since symbolization is not a mandatory component of disassembly, we focus on IBI as the fundamental problem in this paper.

**Symbolic Methods.** Symbolic disassemblers can be broadly categorized into three classes: *linear disassemblers* [18], *traversal-based disassemblers* [1, 2, 8, 41, 43], and *superset disassemblers* [6, 13, 14, 28, 48].

Linear disassemblers, such as objdump [18], rely on correct binary headers and perform instruction decoding in a purely linear fashion. They are simple and efficient, but fundamentally incapable of handling non-standard binaries; when headers are missing or malformed, they either fail outright or disassemble the entire binary as code.

Traversal-based disassemblers, including Ghidra [1], IDA Pro [2], and Radare2 [43], constitute the most mature class of disassembly tools. These systems identify initial entry points from headers or heuristic patterns and iteratively expand code regions by following control-flow transfers, a process commonly referred to as *recursive disassembly* [49]. Supported by extensive heuristic rules, traversal-based disassemblers achieve near-perfect accuracy on standard binary formats. However, their performance degrades significantly when applied to non-standard binaries or unconventional coding patterns, such as data embedded in hand-written assembly code [32]. This is especially true for binaries or functions lacking reliable entry points, which often results in large code regions remaining undiscovered and being misclassified as data.

Superset disassemblers [6, 13, 14, 28, 48] adopt a fundamentally different strategy by decoding instructions at every possible byte offset to construct a superset of candidate instructions. The final disassembly is obtained by applying a large set of manually designed rules to eliminate invalid candidates. While this approach has demonstrated strong performance on standard binaries, it relies on hard, predefined logical constraints. When these rules conflict or fail to apply (such as in non-standard or obfuscated binaries), the disassembly process may halt entirely. For example, DDisasm [14] and WIS [13] refuse to process non-standard binaries and often fail on heavily obfuscated inputs.

**Neural Methods.** Recent neural network–based disassemblers [7, 33, 47, 51] demonstrate that learning-based approaches can improve robustness over symbolic techniques, particularly on non-standard binaries. These methods employ architectures such as Graph Neural Networks or Transformers and operate on raw bytes or decoded instruction sequences. Neural disassemblers typically offer fast inference and achieve high accuracy in scenarios where symbolic assumptions break down.

However, all existing neural disassemblers share a fundamental limitation: they require large, ISA-specific labeled training datasets. Porting these systems to new architectures or binary formats entails costly data collection and retraining. In practice, acquiring ground-truth labels can be extremely challenging. This data dependency limits the practicality and generality of purely neural approaches.

## 2.2 Neurosymbolic AI

Neurosymbolic AI is a research paradigm that integrates symbolic reasoning with neural representation learning in a unified learning and inference framework [11, 39]. Symbolic AI relies on explicit, human-interpretable representations such as logical rules and constraints, providing strong guarantees for formal reasoning and explainability. Neural networks, in contrast, excel at learning latent patterns from raw data but typically lack explicit structural guarantees.

A defining characteristic of neurosymbolic AI is that symbolic knowledge is incorporated directly into the optimization objective, such that logical constraints actively guide learning and inference rather than being applied as post hoc filters. This distinguishes neurosymbolic systems [7, 24, 25, 27, 33, 45–47, 51] from loosely coupled pipelines in which neural predictions are merely refined by heuristic validation, rule-based pruning, or tool orchestration. While such pipelines can be practically effective, they do not perform joint reasoning over learned representations and symbolic structure in the strict sense.

This tight integration enables neurosymbolic systems to address problems that combine statistical regularities with strong global constraints. By jointly optimizing neural predictions and logical consistency, neurosymbolic models aim to achieve improved robustness, generalization, and interpretability [11, 39].

Binary disassembly is particularly well-suited to this paradigm. The task is inherently *partially observable*: instruction boundaries, control-flow relationships, and code–data distinctions are often ambiguous or missing, especially in non-standard binaries. Moreover, disassembly decisions are globally coupled—misclassifying a single block can invalidate large portions of the recovered control-flow graph. Many correctness conditions are therefore structural rather than statistical (e.g., control-flow consistency and call-target validity), making them difficult to capture using local features or purely data-driven loss functions. These properties motivate a framework that can learn latent representations while enforcing global logical constraints during inference.

In this work, we adopt a neurosymbolic formulation based on differentiable optimization, enabling joint reasoning over neural predictions and symbolic constraints for the task of binary disassembly.

## 2.3 Logic Tensor Networks

Logic Tensor Networks (LTN) [5] is a neurosymbolic AI framework that integrates first-order logic with neural networks through differentiable optimization. LTN employs a differentiable first-order logic formalism based on *fuzzy logic*, which allows logical predicates to take continuous truth values in the range $[0, 1]$. This design enables symbolic constraints to be enforced softly rather than as hard requirements, an essential property when analyzing real-world binaries that may violate assumed invariants. As a result, LTN supports robust reasoning in the presence of ambiguity, incomplete information, and conflicting evidence.

**Traditional First-Order Logic vs. Fuzzy Logic.** In traditional first-order logic (FOL), predicates are Boolean-valued, returning either *true* or *false*. While this formulation is well-suited to formal verification, it is often too rigid for domains such as disassembly, where heuristics and structural assumptions may hold in most—but not all—cases. Enforcing such rules as hard constraints can cause symbolic systems to fail entirely when confronted with non-standard binaries.

Fuzzy logic generalizes FOL by allowing predicates to return real values in $[0, 1]$, representing degrees of truth. This relaxation enables partial satisfaction of logical constraints, allowing the system to continue reasoning even when some assumptions are violated. In the context of disassembly, this property is crucial: patterns such as conditional branches, control-flow transfers, or disassembly failures are strong indicators of code or data, but none are universally valid.

**Neural Networks as Predicates.** In LTN, predicates need not be purely symbolic; they can be implemented as neural networks that map high-dimensional inputs to continuous truth values. This capability allows LTN to incorporate learned representations directly into logical reasoning. In our setting, neural predicates enable the system to learn block-level representations from binary features while remaining subject to global logical constraints. This design contrasts with purely neural approaches, which typically optimize local classification accuracy without enforcing structural consistency across the program.

**Constructing Logic Rules in LTN.** Once predicates are defined, LTN supports the construction of logical rules using standard connectives and quantifiers, interpreted under fuzzy semantics. In this work, we adopt commonly used fuzzy operators:

- **NOT:** $\neg u = 1 - u$
- **AND:** $u \wedge v = uv$
- **OR:** $u \vee v = u + v - uv$
- **Implication:** $u \Rightarrow v = 1 - u + uv$

These operators allow logical implications to be satisfied to varying degrees, enabling soft enforcement of constraints such as control-flow consistency. This flexibility is important for disassembly, where rules derived from expert knowledge may admit exceptions in non-standard binaries.

LTN supports both universal ($\forall$) and existential ($\exists$) quantification via smooth aggregation operators. The *p-Mean* operator is typically used for existential quantification, while the *p-MeanError* operator is used for universal quantification. With sufficiently large positive

values of $p$, these operators approximate the maximum and minimum functions, respectively, while remaining differentiable—an essential property for gradient-based optimization [37].

**Learning in LTN.** Given a set of predicates and logical rules, LTN constructs a loss function that combines data-driven objectives with logical constraint satisfaction. The learning process seeks to maximize the overall satisfiability of the rule set while adjusting the parameters of neural predicates. Importantly, this formulation optimizes global consistency across all entities and relationships, rather than making independent, local predictions.

For binary disassembly, this allows the system to learn a block-level code–data classifier whose predictions collectively satisfy structural constraints such as control-flow relationships, rather than optimizing classification accuracy in isolation.

## 3 LTN DISASSEMBLY

Following the *global constraint optimization* insight, the disassembly task is no longer treated as a series of independent classification decisions. Instead, we formulate it as a joint optimization problem: the neural component learns to identify code/data patterns from block-level features, while the symbolic component enforces structural consistency through logical rules (e.g., if a block is a jump target, it must be code) simultaneously. LTN provides the mechanism to transform these rules into a differentiable loss function, allowing the model to self-correct its predictions based on the global context of the binary. Because logical dependencies are often recursive, where identifying one code block can provide the necessary evidence to resolve another, we implement this process as an iterative loop. This loop ensures that the symbolic constraints and neural predictions progressively converge toward a globally consistent disassembly result.

We summarize our approach in Algorithm 1. The algorithm is iterative and continues until no additional code blocks are discovered. Each iteration consists of three stages: *Pre-processing*, *Training*, and *Post-training*. During *Pre-processing*, we extract block-level representations from the binary, including features and inter-block relationships. In the *Training* stage, a neural model is optimized jointly with predefined logical constraints encoded as LTN rules. In *Post-training*, the trained model predicts the likelihood of each block being code; newly identified code blocks are then re-disassembled and incorporated into the next iteration.

**Algorithm 1:** LTN-based disassembly algorithm.

**Input:** Binary file $binary$
**Output:** List of code blocks $code\_blocks$, list of data blocks $data\_blocks$

```
repeat
    blocks ← extract_blocks(binary);
    generate_feature_vector(blocks);
    generate_relationships_between_blocks(blocks);
    model ← new Model();
    train(model, blocks);
    code_blocks, data_blocks ← check_blocks(model, blocks);
    redisassemble(code_blocks);
until no new code blocks are found;
```

### 3.1 Pre-processing

The goal of the *Pre-processing* stage is to extract block-level information from the binary using a traversal-based disassembler. We leverage `Ghidra` to perform automated analysis, which provides an initial partitioning of memory into code, data, and unknown regions. Ghidra employs recursive disassembly to identify potential code blocks based on explicit and potential implicit control-flow transitions, resulting in a conservative classification in which a region is marked as code only when sufficient structural evidence exists.

Using Ghidra as the foundation offers several advantages. First, its support for a wide range of Instruction Set Architectures (ISAs) enables cross-architecture analysis through uniform block-level abstractions. Second, Ghidra is highly effective on standard binaries, accurately identifying complex structures such as exception-handling metadata and C++ virtual tables. Third, even on non-standard binaries, Ghidra provides a reliable, though conservative, baseline for subsequent refinement. Prior comparative studies also show that Ghidra achieves higher disassembly accuracy than other traversal-based tools [49], making it well suited for integration into our framework.

The pre-processing stage consists of three functions:

- `extract_blocks()`
- `generate_feature_vector()`
- `generate_relationships_between_blocks()`

The `extract_blocks()` function retrieves block-level information by invoking Ghidra's analysis pipeline. Code blocks correspond to regions successfully disassembled into instructions. Data blocks are regions assigned concrete data types, typically because they are referenced by identified instructions. Unknown blocks represent memory regions for which Ghidra lacks sufficient evidence to make a definitive classification.

While Ghidra's initial code and data blocks provide reliable heuristics, unknown blocks require further refinement. Such blocks often span large contiguous regions containing interleaved code and data, making them unsuitable for direct inference. To address this, we apply a two-step refinement process. First, we perform a linear sweep over each unknown block, which we refer to as *pseudo-disassembly*. Second, we split the block at each control-flow transfer instruction identified during this sweep (e.g., `jump`, `branch`, or `return`). This produces smaller *fine-grained unknown blocks*, each corresponding to either a pure code region, a pure data region, or a short data segment followed by code. By ensuring that any candidate code sequence appears at the end of a fine-grained block, this refinement enables more precise feature extraction and relationship modeling.

The `generate_feature_vector()` function computes a feature vector for each block produced by `extract_blocks()`. These vectors serve as compact embeddings of block-level characteristics. We adopt features similar to those used in Gemini [50], summarized in Table 1. To account for differences in block size, each feature is normalized by the total byte length of the block.

Finally, the `generate_relationships_between_blocks()` function records control-flow transfer relationships between blocks, capturing potential execution paths. These relationships are later

**Table 1: Features used to generate block feature vectors.**

| Block Features (# denotes "Number", counted per block) |
|---|
| # strings |
| # constants |
| # control-flow transfer instructions |
| # call instructions |
| # instructions |
| # arithmetic operations |
| # zero bytes |
| # def-use chains within a 16-instruction window |
| # printable characters |

used by the neurosymbolic framework to reason about block-level code–data classifications.

## 3.2 Training

The goal of the *Training* stage is to learn a block-level classifier that predicts whether a block represents executable code, while enforcing global structural constraints derived from disassembly semantics. We formulate this problem using a LTN, which jointly optimizes a neural predicate under a set of predefined logical rules.

**Predicates.** We formalize the LTN using the predicates listed in Table 2. These predicates fall into two categories: *function predicates*, whose values are fixed and derived during pre-processing, and a single *neural predicate*, which is learned during training.

Predicates P1–P7 represent block-level properties extracted during the preprocessing stage, with values grounded in the discrete set $\{0, 0.5, 1\}$. The predicates $\text{CodeBlock}_g$ and $\text{DataBlock}_g$ capture Ghidra's initial, conservative classification results. Specifically, for blocks successfully disassembled by Ghidra, we assign $\text{CodeBlock}_g = 1$ and $\text{DataBlock}_g = 0$. Conversely, for regions explicitly identified as data, the predicate values are inverted accordingly. For unknown blocks as mentioned in §3.1, they are initialized with neutral values ($\text{CodeBlock}_g = \text{DataBlock}_g = 0.5$).

Predicates CondJump_t and CondJump_f are motivated by the observation in Loadstar [7] that a conditional jump is a strong indicator of executable code only when it is preceded by a comparison instruction. Accordingly, a block ending with such a pattern sets $\text{CondJump\_t}(b) = 1$, while a conditional jump without a preceding comparison sets $\text{CondJump\_f}(b) = 0$.

Predicates HighZero, HighContPrintableChar, and FailDisasm capture negative evidence for code blocks. Code blocks rarely contain large regions of zero bytes or long continuous strings, and blocks that fail linear-sweep disassembly are unlikely to represent valid instructions.

Predicate $\text{Call}(a, b)$ records call relationships between blocks, including those identified during pseudo-disassembly of data blocks. If a call edge from block $a$ to block $b$ is observed, the predicate $\text{Call}(a, b)$ is set to 1.

Finally, predicate $\text{CodeBlock}(b)$ is the only trainable component. It is implemented as a multilayer perceptron (MLP) that maps the feature vector of block $b$ to a real-valued probability indicating whether $b$ is a code block.

**Logic Rules.** We encode disassembly knowledge using the logical rules shown in Table 3. Rules R1–R7 associate individual block properties with code or data classification, while R8 captures control-flow consistency across blocks.

Rules R1 and R2 treat Ghidra's results as strong but conservative hints. Rules R3 and R4 exploit conditional-branch structure described in Loadstar [7] to discover new code blocks missed by traversal-based disassembly. Rules R5–R7 encode negative evidence for code blocks. Rule R8 enforces control-flow consistency: a code block may only transfer control to another code block.

**Optimization Objective.** Each rule is evaluated under fuzzy semantics, yielding a real-valued satisfaction score in the interval $[0, 1]$. For unary rules, this score is computed individually for each block; for binary rules, such as R8, the score is calculated over block pairs exhibiting established call relationships. As an example, the local satisfaction score of R1 over a specific block $b$, denoted as $r_1(b)$, is defined as:

$$r_1(b) = 1 - \text{CodeBlock}_g(b) + \text{CodeBlock}_g(b) \cdot \text{CodeBlock}(b)$$

This formulation is derived from the fuzzy implication operator within the LTN framework. To generalize these local scores across all blocks, universal quantification is implemented using the pMeanError (pME) aggregator. This yields a single global satisfaction value $R_i$ for each rule. For instance, the final score for R1 is calculated as:

$$\begin{aligned} R_1 &= \text{pME}\left(r_1(b_1), r_1(b_2), \ldots, r_1(b_n)\right) \\ &= 1 - \left(\frac{1}{n}\sum_{i=1}^{n}(1 - r_1(b_i))^p\right)^{\frac{1}{p}}, \quad p \geq 1 \end{aligned}$$

where $\{b_1, \ldots, b_n\}$ represents the set of all blocks in the binary and $p$ is a predefined hyperparameter. When $p = 1$, the pME reduces to the arithmetic mean; as $p \to \infty$, it converges to the min operator. Intuitively, the pME serves as a differentiable, "smooth" approximation of the min operator, allowing the optimizer to account for rule violations across the entire block set while prioritizing the most significant outliers.

All rule satisfaction values are then aggregated into a global satisfiability score:

$$\text{SAT} = \text{pME}(\text{R}_1, \text{R}_2, \ldots, \text{R}_8)$$

The objective is to maximize SAT; consequently, we define the loss function as $\text{loss} = 1 - \text{SAT}$. This allows the application of gradient descent to minimize the loss, consistent with standard machine learning practices. Since the only free variables are the parameters within the CodeBlock MLP, the LTN framework tracks the gradients of these variables during the forward pass (loss calculation). A backpropagation step is then performed to optimize the MLP, aiming to minimize the loss and maximize SAT to achieve an optimal CodeBlock model through the training process.

## 3.3 Post-processing

After obtaining the CodeBlock predictions from training, we label each block accordingly. We first apply the CodeBlock predicate to classify each block and store them in two separate lists: code blocks and data blocks.

**Table 2: Predicates used in our LTN formalization.**

| ID | Predicate | Description | Type |
|---|---|---|---|
| P1 | CodeBlock_g($b$) | Block $b$ identified as code by Ghidra | Function |
| P2 | DataBlock_g($b$) | Block $b$ identified as data by Ghidra | Function |
| P3 | CondJump_t($b$) | Conditional jump with preceding comparison | Function |
| P4 | CondJump_f($b$) | Conditional jump without preceding comparison | Function |
| P5 | HighZero($b$) | Zero-byte ratio exceeds 50% | Function |
| P6 | HighContPrintableChar($b$) | Long continuous printable characters | Function |
| P7 | FailDisasm($b$) | Linear sweep disassembly failure | Function |
| P8 | Call($a, b$) | Call relationship from block $a$ to $b$ | Function |
| P9 | CodeBlock($b$) | Block $b$ is code (neural prediction) | Neural |

**Table 3: Logic rules used in our LTN.**

| ID | Logical formulation |
|---|---|
| R1 | $\forall b$ (CodeBlock_g($b$) $\Rightarrow$ CodeBlock($b$)) |
| R2 | $\forall b$ (DataBlock_g($b$) $\Rightarrow \neg$ CodeBlock($b$)) |
| R3 | $\forall b$ (CondJump_t($b$) $\Rightarrow$ CodeBlock($b$)) |
| R4 | $\forall b$ (CondJump_f($b$) $\Rightarrow \neg$ CodeBlock($b$)) |
| R5 | $\forall b$ (HighZero($b$) $\Rightarrow \neg$ CodeBlock($b$)) |
| R6 | $\forall b$ (HighContPrintableChar($b$) $\Rightarrow \neg$ CodeBlock($b$)) |
| R7 | $\forall b$ (FailDisasm($b$) $\Rightarrow \neg$ CodeBlock($b$)) |
| R8 | $\forall a, b$ (Call($a, b$) $\Rightarrow$ (CodeBlock($a$) $\Rightarrow$ CodeBlock($b$))) |

Next, we pass the list of newly identified code blocks to the *redisassemble()* function. For each code block, we invoke Ghidra to disassemble starting from the block's beginning. During this process, Ghidra attempts to discover additional cross-references, control flow transfers, and other relevant information.

The re-disassembly step is particularly valuable for resolving blocks that were previously treated as single data blocks (i.e., *fine-grained unknown block*) but actually contain mixed data and code segments, as described in §3.1. Such mixed blocks have feature vectors that are less informative because they combine both code and data. By re-disassembling, we can identify the boundary between data and code blocks using the newly discovered control flow transfers. These mixed blocks are then split into separate data and code blocks by Ghidra in subsequent iterations, improving the overall accuracy of block classification.

## 4 IMPLEMENTATION

We implement `NSDA` on `Ghidra` 11.3.2 [1] and `LTNtorch` 1.0.2 [9], with a total implementation size of approximately 1600 lines of code. LTNtorch [9] is a PyTorch-based implementation of Logic Tensor Networks; we adopt it for better engineering support and maintainability.

**Pre-processing.** The pre-processing stage is implemented using `pyghidra`, which provides direct access to the Ghidra API from a native CPython 3 environment via JPype [31]. While Ghidra performs automatic analysis on the input binary, we do not directly rely on its `BasicBlock` abstraction for two reasons:

(1) Ghidra's `BasicBlock` does not strictly conform to the conventional definition of a basic block. For example, blocks ending in a `call` instruction are not terminated, as Ghidra assumes control will return to the caller, causing the call and fallthrough instructions to be merged.
(2) Ghidra does not provide a dedicated abstraction for continuous data regions, which are required for our block-level reasoning.

Instead, we build blocks using Ghidra's `CodeUnit` abstraction, which is the superclass of both `Instruction` and `Data` classes. After Ghidra's initial analysis, every address in the binary is classified as either an `Instruction` or `Data`. We iterate over all `CodeUnit` objects to construct continuous instruction and data regions. Instruction regions are further split at control-flow transfer instructions (i.e., `call`, `branch`, and `return`), assuming these regions are correctly identified as code by Ghidra.

Data `CodeUnits` are divided into two categories. Typed data corresponds to memory regions explicitly referenced by instructions, while unknown data, typically marked by the "??" mnemonic, represents regions for which Ghidra cannot infer semantics. Contiguous typed data units are grouped into data blocks, whereas consecutive unknown data units are merged into unknown blocks.

For each unknown block, we perform a *pseudo-disassembly* using a linear sweep algorithm. Whenever a control-flow transfer instruction is encountered, the block is split at that instruction, with the instruction included in the preceding block. This process yields smaller, fine-grained blocks suitable for feature extraction and inference.

During block construction, we also record branch and call targets to capture inter-block control-flow relationships. These relationships are stored as edges in a graph and later used by the LTN during training. In addition, block-level statistics are collected to generate grounding values for property predicates (P1–P8) and to construct feature vectors.

**Training.** In our implementation, the hyperparameter $p$ for the pME aggregator is set to 4. The neural predicate CodeBlock is implemented as a multilayer perceptron (MLP) [36] with two hidden layers of sizes 32 and 64. We use the `Adam` optimizer [21] with a learning rate of 0.001. Training runs for 900 epochs, during which the LTN objective maximizes global rule satisfiability (SAT).

**Post-processing and Iteration.** After training, the CodeBlock predicate is applied to classify each block. Blocks with predicted values greater than 0.5 are labeled as code, while the remainder are treated as data. When a block newly classified as code was previously considered data, we invoke Ghidra to disassemble starting

from the block's base address. Ghidra performs recursive disassembly to discover additional instructions, control-flow transfers, and cross-references, which are incorporated into the analysis.

This process is repeated iteratively: newly discovered code blocks are added, features and relationships are regenerated, and the model is retrained. Iteration continues until no additional code blocks are identified. To ensure termination, we cap the procedure at a maximum of five iterations.

# 5 EVALUATION

In this section, we evaluate the effectiveness and efficiency of NSDA across diverse binary types.

For standard binaries, our dataset predominantly includes coreutils programs across two architectures, ARM32 and MIPS, compiled by both GCC [15] and LLVM [23]. For these configurations, accurate ground truth can be extracted during the compilation process. To extend the general applicability of NSDA, we also include a representative application, OpenSSL [44], from the x86-sok dataset [32]. Because many cryptographic algorithms are written in assembly code for efficiency, they often contain embedded data, such as special instructions and constant tables, interspersed within the code.

For 32-bit architectures such as ARM32, instruction set limitations necessitate the frequent inlining of literal pools and jump tables (e.g., switch-case tables) directly within the `.text` section, which poses significant challenges for accurate disassembly (as illustrated in Table 5.5). For MIPS, the prevalence of frequent indirect calls poses additional challenges when encountering blocks that lack an explicit control flow transfer. Given that these architectures remain widely utilized in embedded systems, network equipment, and industrial control systems, we believe it is of significant practical value to focus our research on these platforms.

For non-standard binaries, we utilize the datasets published by Benkraouda et al. [7]. These datasets consist of three subsets of Programmable Logic Controller (PLC) binaries, which are ubiquitous in industrial control systems. These ARM32-based binaries were collected from various real-world repositories and compiled using specialized PLC compilers.

We aim to answer the following research questions:

- **RQ1**: To what extent do state-of-the-art symbolic methods, such as Ghidra, effectively disassemble both standard and non-standard binaries?
- **RQ2**: What degree of accuracy improvement does NSDA achieve over the Ghidra baseline, and how does its performance compare to other symbolic and neural-based approaches?
- **RQ3**: What is the computational efficiency of NSDA?
- **RQ4**: How does the integration of a neural network facilitate the logical reasoning process within the NSDA framework?
- **RQ5**: Which specific logical rules contribute most significantly to the overall performance of NSDA?
- **RQ6**: Is NSDA scalable enough for processing large-scale binaries?

In Research Question 1 (**RQ1**), we evaluate the efficacy of state-of-the-art symbolic disassemblers when applied to both standard and non-standard binaries. Regarding standard binaries, we anticipate that in the absence of inlined data within code sections, tools such as Ghidra and DDisasm can achieve near-ideal disassembly accuracy. However, for standard binaries containing inlined data, we aim to statistically quantify the level of accuracy these tools can maintain. For non-standard binaries, we investigate whether these tools can successfully parse the underlying structures and the degree of disassembly accuracy they achieve. Although the authors of Loadstar [7] claim that Ghidra performs poorly on non-standard binaries, their work lacks a comprehensive experimental evaluation and detailed statistical evidence to support this assertion.

**RQ2** investigates the performance gains of NSDA relative to Ghidra and evaluates its effectiveness against both symbolic and neural-based methods, specifically DDisasm and Loadstar. This comparison allows us to determine whether the integration of neurosymbolic reasoning yields significant advantages over purely symbolic or purely neural approaches.

**RQ3** emphasizes efficiency. Neurosymbolic reasoning combines symbolic rule evaluation with gradient-based learning, which may raise concerns about computational cost. We thus examine whether NSDA remains practical in terms of runtime performance.

**RQ4** investigates the utility of integrating neural networks into the logical reasoning process. Since logical reasoning can be performed using purely symbolic approaches (e.g., probabilistic logic), we evaluate whether a purely symbolic implementation using the same rule set can achieve comparable performance without the neural component.

**RQ5** aims to isolate the individual contributions of each logical rule within NSDA. By evaluating the performance impact of various rule configurations, we can identify the most critical rules for handling non-standard binaries and provide insights into potential optimizations for the rule set.

Finally, **RQ6** evaluates the scalability of NSDA for disassembling large-scale binaries. By investigating potential architectural limitations and exploring necessary adaptations, we assess how NSDA can be scaled to process large-scale binaries without sacrificing accuracy.

## 5.1 Evaluation Setup and Datasets

**Evaluation Setup.** We evaluate NSDA on a server running Ubuntu 22.04.4 LTS, equipped with 12 Intel(R) Xeon(R) CPUs @ 2.30GHz and an NVIDIA V100-SXM2 GPU with 16 GB of memory. The software environment utilized CUDA 12.8 for the training.

**Datasets.** Our evaluation utilizes two distinct datasets: a collection of standard binaries developed for this study and a set of non-standard PLC binaries from Loadstar [7].

For the first dataset, we compiled coreutils [17] for ARM32 and MIPS architectures using both gcc [15] and LLVM [23]. We maintained both stripped and non-stripped versions of these binaries; the stripped versions serve as the input for our disassembly evaluation, while the non-stripped versions provide the ground truth for memory layout and instruction boundaries.

Regarding ARM32, we observed that both gcc and LLVM insert mapping symbols—`$a`, `$d`, and `$t`—into the symbol table to demarcate the start of ARM instructions, data, and Thumb instructions,

**Table 4: Precision (P), Recall (R), and F1 score of NSDA and baselines**

| Dataset | Ghidra | | | DDisasm | | | Loadstar | | | NSDA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | R | F1 | P | R | F1 | P | R | F1 | P | R | F1 |
| Coreutils - ARM32 - gcc | 0.9996 | 0.9554 | 0.9768 | **0.9997** | 0.9842 | 0.9906 | 0.9993 | 0.8947 | 0.944 | 0.9948 | **0.9984** | **0.9966** |
| Coreutils - MIPS - gcc | **1** | 0.8599 | 0.9237 | **1** | **0.9921** | **0.996** | - | - | - | 0.999 | 0.979 | 0.9889 |
| Coreutils - ARM32 - LLVM | 0.9905 | 0.9134 | 0.9499 | 0.9921 | 0.996 | 0.9937 | **1** | 0.8959 | 0.9449 | 0.9882 | **0.9998** | **0.994** |
| Coreutils - MIPS - LLVM | **1** | 0.7921 | 0.88 | **1** | **0.9934** | **0.9967** | - | - | - | 0.9984 | 0.9834 | 0.9908 |
| OpenSSL - x64 | **0.9888** | 0.8878 | 0.9356 | 0.9738 | 0.2724 | 0.4257 | - | - | - | 0.9503 | **0.9972** | **0.9731** |
| PLC - NS1 | 0.9936 | 0.9815 | 0.9875 | - | - | - | **0.9992** | **0.9971** | **0.9982** | 0.9911 | 0.9819 | 0.9865 |
| PLC - NS2 | 0.9939 | 0.9812 | 0.9875 | - | - | - | **0.9996** | **0.9976** | **0.9986** | 0.9924 | 0.9815 | 0.9869 |
| PLC - NS3 | 0.9865 | 0.5864 | 0.7355 | - | - | - | **0.9993** | 0.9382 | **0.9678** | 0.9332 | **0.9386** | 0.9357 |

respectively. These symbols facilitate the accurate recovery of the original memory layout for code and data sections.

Regarding OpenSSL, we additionally included this application from the x86-sok [32] dataset. This collection comprises six OpenSSL binaries (version 1.1.0l) compiled across six distinct optimization levels (O0–O3, Ofast, and Os).

The `Loadstar` dataset is partitioned into three distinct subsets—NS1, NS2, and NS3—each characterized by unique architectural features and generation methodologies. These subsets encapsulate a diverse range of PLC binaries derived from industrial control systems and generated using multiple specialized toolchains. A detailed description of the dataset characteristics and the collection process is provided in Benkraouda et al. [7].

## 5.2 Accuracy

To address **RQ1** and **RQ2**, we evaluate and compare the accuracy of NSDA against Ghidra, DDisasm, and Loadstar.

We adopt precision, recall, and the F1 score (the harmonic mean of precision and recall) for the *Code* class as our primary accuracy metrics. Since our classification for each memory address is binary—categorized as either *Code* or *Data*—the performance metrics for both classes are intrinsically linked. An increase in the classification accuracy for the *Data* class necessitates a corresponding improvement in *Code* class performance. Consequently, to ensure conciseness in our experimental reporting, we present only the metrics for the *Code* class as the representative measure of performance. Finally, to ensure a fair comparison across all baselines, we employ the byte as our fundamental unit of measurement for accuracy evaluation, with the exception of §5.5.

The results are presented in Table 4. Our evaluation reveals that only Ghidra and NSDA support the complete range of datasets; in contrast, DDisasm is constrained by its dependency on specific binary formats and metadata requirements, often failing to process binaries with unrecognized formats. Furthermore, while the Loadstar framework necessitates re-training its embedding layer and classification components to accommodate new Instruction Set Architectures (ISAs), our neurosymbolic approach provides broader out-of-the-box compatibility, requiring no additional architecture-specific modifications to achieve high accuracy across diverse platforms.

Our results indicate that DDisasm outperforms Ghidra across all standard datasets except for OpenSSL; moreover, it lacks support for non-standard PLC binary formats. Notably, Ghidra demonstrates high performance on the NS1 and NS2 subsets of Loadstar despite their non-standard nature. This effectiveness stems from the fact that control-flow transitions within these two subsets are predominantly explicit, enabling Ghidra to reconstruct the control-flow graph (CFG) with high fidelity and generate accurate code-block classifications. These findings contrast with the intuition provided in the Loadstar paper [7] and demonstrate that Ghidra possesses a degree of inherent robustness when disassembling binaries in previously unseen formats.

In comparison to Ghidra, NSDA achieves competitive parity on the NS1 and NS2 subsets while demonstrating a marked performance improvement across the standard datasets (2%—12%) and the Loadstar NS3 subset (27%). These results underscore the efficacy of NSDA, particularly its capacity to significantly enhance disassembly accuracy over the Ghidra baseline in more complex scenarios. Furthermore, when compared to DDisasm, NSDA elevates Ghidra's initial output to a level of performance comparable to DDisasm's specialized results. Crucially, NSDA maintains the architectural versatility required to process non-standard binaries—a vital capability that DDisasm lacks due to its rigid metadata dependencies.

The Loadstar framework achieves near-optimal performance on its native datasets (NS1, NS2, and NS3) due to several methodological factors. First, since the authors do not explicitly define the boundaries between their training and testing partitions—stating only that a 70%-30% split was utilized—the reported results likely represent an optimistic upper bound rather than a robust measure of generalized performance. Second, the extensive fine-tuning of their classifier on these specific subsets introduces a high risk of overfitting. This is particularly evident in the `Coreutils ARM32 gcc` and `Coreutils ARM32 LLVM` results; despite being initially trained on this data, Loadstar's F1 score dropped to 0.944 following subsequent fine-tuning on the PLC subsets.

This performance degradation, the lowest among all evaluated disassemblers for standard binaries, suggests that Loadstar's specialized optimization for non-standard formats significantly compromises its generalizability to standard binaries. Furthermore, adapting the Loadstar framework to a different ISA, such as MIPS, necessitates the training of a new embedding layer and classifier. This requirement imposes a significant operational burden, as it relies on the availability of extensive labeled training data for the target architecture. In contrast, NSDA maintains architectural agnosticism, providing immediate support for diverse ISAs without the need for architecture-specific re-training or additional dataset curation.

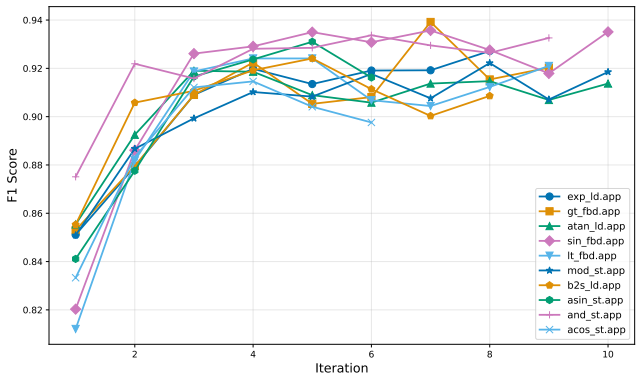



**Figure 1: Code F1 score change over iterations on 10 samples from NS_3.**

To more clearly demonstrate the performance gains provided by NSDA over Ghidra, we randomly selected 10 samples from the NS3 subset and tracked their F1 scores over successive iterations, as illustrated in Figure 1. For this experiment, we extended the refinement process to 10 iterations to better observe the convergence behavior.

The results indicate that most samples exhibit a steady increase in F1 score during the first five iterations, followed by a period of minor oscillation. Crucially, all samples converge at a significantly higher F1 score compared to the initial baseline provided by Ghidra.

These 10 representative samples confirm that the integration of neural learning and symbolic logic reasoning within NSDA facilitates a notable improvement in disassembly accuracy, consistently refining the preliminary outputs generated by Ghidra.

## 5.3 Efficiency

To address **RQ3**, we evaluate the execution time of NSDA across its three constituent phases: pre-processing, training, and post-processing. The pre-processing and post-processing stages are managed by the Ghidra framework and are consequently governed by its internal implementation and efficiency. These phases are primarily CPU-bound but benefit from partial parallelization, as Ghidra distributes its analysis tasks across multiple available cores. The training phase encompasses the initialization of the LTN, followed by forward propagation for satisfiability (SAT) and loss calculation, and finally backward propagation for gradient descent. This stage is computationally flexible, supporting execution on both CPU and GPU architectures, with the latter providing a significant performance advantage for tensor operations.

We present the efficiency evaluation in Figure 2, which illustrates the execution time across all datasets. To maintain readability given the large volume of binaries, we group binaries of similar sizes and report the average execution time for each group. The binaries are sorted by size along the x-axis, with the y-axis denoting execution time in seconds. Each bar is partitioned into three segments: the blue portion represents pre-processing, the green represents Ghidra-based post-processing performed on the CPU, and the orange portion signifies the neural network training time. Additionally, Table 5 details the minimum and maximum training durations for each dataset.

**Table 5: Min and Max Training Time Across Datasets**

| Dataset / Subset | Train Time Min / Max (s) |
|---|---|
| Coreutils – ARM32 – gcc | 19.06 / 50.74 |
| Coreutils – MIPS – gcc | 28.24 / 49.85 |
| Coreutils – ARM32 – LLVM | 18.39 / 45.75 |
| Coreutils – MIPS – LLVM | 26.53 / 50.95 |
| OpenSSL – x64 | 87.22 / 106.03 |
| Loadstar | 7.06 / 47.64 |

As shown in Figure 2, the average training time does not exhibit a strictly linear increase relative to binary size. This stability is attributed to two primary factors: (1) the training duration is fundamentally a function of the number of blocks—representing the complexity of the logical reasoning task—rather than the raw file size or total instruction count; and (2) the training process is offloaded to the GPU, which leverages parallel processing to handle varying block counts with high efficiency. Furthermore, according to Table 5 the training phase for a single binary typically concludes within 7–50 seconds. This remains well within an acceptable threshold for practical use, ensuring that the enhanced accuracy of our neurosymbolic approach does not come at the cost of prohibitive computational overhead.

## 5.4 Ablation Study

To address **RQ4** and **RQ5**, we conduct two ablation studies. In the first study, we remove the neural network component of NSDA, leaving only the fuzzy logic framework for reasoning. This experiment aims to demonstrate the necessity of a neurosymbolic approach compared to a purely symbolic method for logic reasoning. In the second study, we systematically remove logical rules R1–R8 one by one to evaluate their individual impact and identify which rules contribute most significantly to the overall performance improvement.

**Impact of the Neural Component.** In this experiment, we remove the Multi-Layer Perceptron (MLP) and the block-level feature vectors, leaving only the logical rules (R1–R8). We utilize the fuzzy logic framework of LTN to assign a probability score to each block, indicating its likelihood of being code, and apply the same iterative disassembly strategy with Ghidra. We refer to this purely symbolic variant as *FuzzyNSDA* (Fuzzy-logic NSDA).

The results are presented in Table 6. We observe that while NSDA performs comparably to FuzzyNSDA on the NS1 and NS2 subsets, it significantly outperforms the symbolic baseline across all other datasets with a marked improvement in F1 score. This illustrates that the neural network is a critical component of the reasoning process, providing a substantial accuracy boost in the majority of disassembly scenarios.

We also observe that FuzzyNSDA performs similarly to Ghidra. This aligns with our intuition that Ghidra has already integrated similar fundamental heuristics into its disassembly logic. Consequently, applying a purely probabilistic reasoning approach using

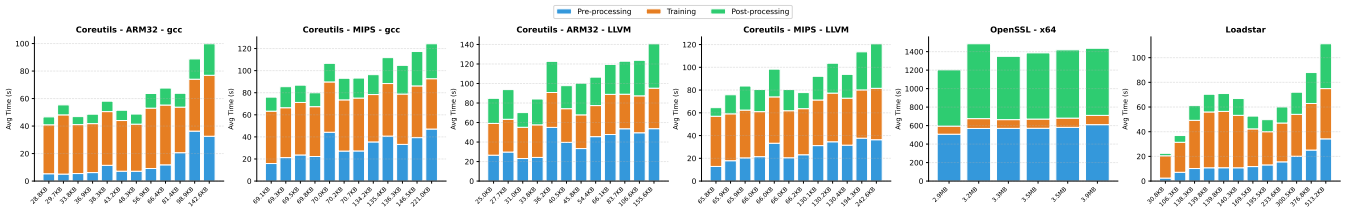



**Figure 2: Time cost of NSDA, with data points grouped by file size and averaged within each group.**

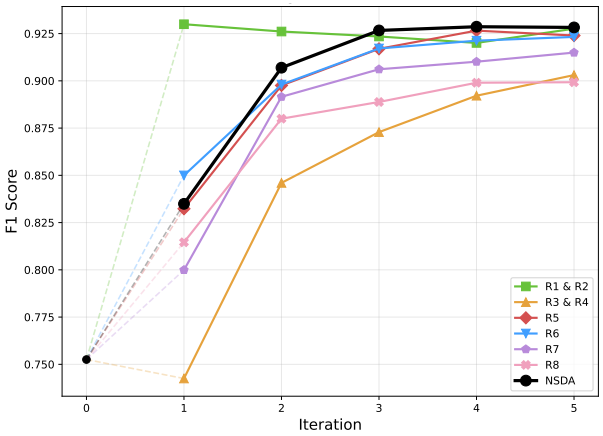



**Figure 3: Effect on F1 score over iterations of removing specific rule(s).**

only these basic rules does not yield a significant performance gain. This further justifies the integration of neural components in NSDA to capture more complex patterns that transcend simple symbolic rules.

**Table 6: F1 score comparison between NSDA and FuzzyNSDA**

| Dataset | NSDA | FuzzyNSDA |
|---|---|---|
| Coreutils – ARM32 – gcc | **0.9966** | 0.9768 |
| Coreutils – MIPS – gcc | **0.9889** | 0.9237 |
| Coreutils – ARM32 – LLVM | **0.9940** | 0.9499 |
| Coreutils – MIPS – LLVM | **0.9908** | 0.8799 |
| OpenSSL – x64 | **0.9731** | 0.9356 |
| PLC – NS1 | 0.9865 | **0.9875** |
| PLC – NS2 | 0.9869 | **0.9875** |
| PLC – NS3 | **0.9357** | 0.7355 |

**Impact of Individual Logical Rules.** We investigate the individual contributions of each rule to the overall performance of NSDA. Initial experiments indicate that removing any single rule results in negligible variation (less than 5%) in the final metrics. To better visualize these nuances, we randomly selected 10 binaries from the NS3 subset and tracked their iterative F1 scores under different rule configurations. The averaged results are presented in Figure 3.

In Figure 3, the data point at iteration 0 represents the Ghidra baseline, while the black line denotes the performance of NSDA under its optimal configuration. The colored curves illustrate the performance trajectories when specific rule sets are removed. We observe that most configurations yield lower F1 scores than the complete NSDA framework. Specifically, the removal of R3 and R4 (conditional branch-related rules) results in a significant drop in code accuracy, underscoring their critical role in identifying valid instruction sequences. Furthermore, control-flow-related rules (R8) contribute significantly to discovering latent code sections, as their removal reduces accuracy for code recognition.

Interestingly, removing R1 and R2 (rules derived from Ghidra's initial labels) actually improves both the convergence speed and final performance on the NS3 subset. This indicates that Ghidra's initial mislabeling of certain NS3 binaries can occasionally mislead NSDA's reasoning loop. However, guided by the remaining structural and semantic rules, NSDA eventually overcomes these erroneous starting labels to achieve superior results. In contrast, for standard binaries, NS1 and NS2 subsets, R1 and R2 provide a robust initialization, and their absence results in a significantly slower convergence rate.

## 5.5 Large Scale Binary

To address **RQ6**, we evaluate the effectiveness of NSDA in handling large-scale binaries, complementing our efficiency analysis on small-scale binaries. For this experiment, we selected Chromium (an x64 PE binary, 179 MB) from the dataset provided by Springer et al. [42] as a representative case study.

We observed that Ghidra requires approximately seven hours to complete its initial analysis of Chromium, and its pseudo-disassembly module becomes significantly inefficient when processing binaries of this magnitude. To address this limitation, we introduce *Segmented* NSDA for the analysis of large-scale binaries.

In the Segmented NSDA approach, the system executes Ghidra's full-binary auto-analysis only once. Subsequently, during the pre-processing stage, it bypasses full-binary analysis and exclusively extracts blocks from smaller segments of a fixed length (e.g., 1 MB). This strategy substantially reduces the computational load on the pseudo-disassembler and accelerates both pre-processing and post-processing.

Following this, the complete pipeline—comprising iterative pre-processing, training, and post-processing—is applied to each segment of the binary sequentially until all segments are processed. Segmented NSDA reduces the analysis overhead of large binaries by partitioning them into smaller, manageable units. The efficiency of pre-processing and post-processing for large-scale binaries is significantly enhanced, provided that the block information within

each segment remains sufficiently rich to facilitate neural network training.

Specifically, in this experiment, we partitioned the Chromium binary into 1 MB segments and performed Segmented NSDA on all individual segments sequentially.

The F1 score results are presented in Table 7, and the corresponding time breakdown is detailed in Table 8. To align with the ground truth provided by Springer et al. [42], we utilized functions as the evaluation unit rather than bytes in this experiment. We exclusively included Ghidra [1] as the baseline here, as DDisasm [14] encountered out-of-memory errors on our 128 GB machine when processing the Chromium binary. The execution time per segment is reported as the average across all segments.

The results indicate that while Ghidra successfully disassembles the majority of the functions, Segmented NSDA still yields measurable improvements in the overall F1 score. Furthermore, the time overhead incurred during training is limited compared to the initial processing time required by Ghidra. The entire pipeline for Chromium takes approximately 39 hours to complete, which is an acceptable timeframe for a binary of this scale.

**Table 7: Accuracy Result of Segmented NSDA and baselines on Chromium**

| Method | Precision | Recall | F1 Score |
|---|---|---|---|
| Ghidra | **0.969233** | 0.996423 | 0.982640 |
| Segmented NSDA | 0.969184 | **0.999296** | **0.984010** |

**Table 8: Processing Time Breakdown of Segmented NSDA for Chromium**

| Pipeline Stage | Time (s) | Execution Scope |
|---|---|---|
| Auto-analysis | 25,900 | Once per binary |
| Pre-processing | 534 | Per 1 MB segment |
| Training | 12 | Per 1 MB segment |
| Post-processing | 46 | Per 1 MB segment |

## 5.6 Case Study

We present a case study to demonstrate how NSDA improves upon Ghidra's results, utilizing the base64 program from coreutils compiled with gcc for ARM32.

In base64, invoking the "–version" flag prompts the program to output version, license, and author information to the standard output. This process is executed by calling the library function `version_etc_arn()` located in 'gnulib/lib/version-etc.c'. As this is a common function shared across the coreutils suite, it contains a switch-case statement designed to display author information in different formats based on the number of authors for the corresponding program.

In ARM32, this switch-case table is embedded within the text section and utilizes an 'ldrls pc, [pc, r6, lsl #2]' instruction to jump to the corresponding case branch. As illustrated in Listing 1, the jump instruction is located at 0x15c74, with the jump table spanning from 0x15c7c to 0x15ca0 to accommodate the 10 entries defined in the source code. However, Listing 1 reveals that Ghidra fails to identify three entries (0x15c80, 0x15c84, and 0x15c88), labeling them as DAT_* instead of code addresses (LAB_*). Consequently, the switch-case table is not successfully recovered.

An in-depth analysis reveals that Ghidra fails to disassemble the blocks at 0x15d68, 0x15d85, and 0x15da8, treating them as data (as shown in the lower part of Listing 1) and consequently missing these entries in the switch-case table. This issue typically occurs in Ghidra when control flow cannot be recovered due to indirect jumps (such as the 'ldrls pc' instruction in this instance), and there is no definitive evidence identifying the block as code. Adopting a conservative strategy, Ghidra refrains from disassembling such uncertain blocks, which subsequently results in an unrecovered jump table.

```
...
.text:00015c74  ldrls pc,[pc,r6,lsl #0x2]
                => PTR_LAB_00015c7c
.text:00015c78  b     LAB_00015ec4
PTR_LAB_00015c7c:
.text:00015c7c  addr  LAB_00015cf0
.text:00015c80  addr  DAT_00015d68
.text:00015c84  addr  DAT_00015d84
.text:00015c88  addr  DAT_00015da8
.text:00015c8c  addr  LAB_00015dd4
.text:00015c90  addr  LAB_00015e04
.text:00015c94  addr  LAB_00015e3c
.text:00015c98  addr  LAB_00015e7c
.text:00015c9c  addr  LAB_00015ca4
.text:00015ca0  addr  LAB_00015d10

LAB_00015ca4: ;XREF[1,0]: 00015c9c
.text:00015ca4  add   r2,r5,#0x14
.text:00015ca8  ldmia r2,{r2,r3}
...

DAT_00015d68: ;XREF[1,0]: 00015c80
.text:00015d68  ??    01h
.text:00015d69  ??    10h
...

DAT_00015d84: ;XREF[1,0]: 00015c84
.text:00015d84  ??    01h
.text:00015d85  ??    10h
...

DAT_00015da8: ;XREF[1,0]: 00015c88
.text:00015da8  ??    04h
.text:00015da9  ??    00h
...
```

**Listing 1: Disassemble result of base64 by Ghidra**

NSDA mitigates this gap; during training, the MLP model learns to identify code blocks by evaluating feature vectors against established patterns. The model trained by NSDA correctly determines that the blocks at 0x15d68, 0x15d85, and 0x15da8 should be classified as code. This insight is then fed back to Ghidra, enabling a superior disassembly result, as demonstrated in Listing 2. The corresponding switch-case blocks are disassembled successfully so that the switch-case is recovered successfully. This result is even slightly better than the one produced by Ghidra with non-stripped base64 binary, which contains the debug symbols.

```
...
.text:00015c74  ldrls pc,[pc,r6,lsl #0x2]
                => switchD_00015c74
```

```
default:
.text:00015c78  b     LAB_00015ec4

switchdataD_00015c7c:
.text:00015c7c  addr  switchD_00015c74::caseD_0
.text:00015c80  addr  switchD_00015c74::caseD_1
.text:00015c84  addr  switchD_00015c74::caseD_2
.text:00015c88  addr  switchD_00015c74::caseD_3
.text:00015c8c  addr  switchD_00015c74::caseD_4
.text:00015c90  addr  switchD_00015c74::caseD_5
.text:00015c94  addr  switchD_00015c74::caseD_6
.text:00015c98  addr  switchD_00015c74::caseD_7
.text:00015c9c  addr  switchD_00015c74::caseD_8
.text:00015ca0  addr  switchD_00015c74::caseD_9

caseD_8: ;XREF[2,0]: 00015c74,00015c9c
.text:00015ca4  add   r2,r5,#0x14
.text:00015ca8  ldmia r2,{r2,r3}
...

caseD_1: ;XREF[2,0]: 00015c74,00015c80
.text:00015d68  mov   r1,#0x1
.text:00015d6c  mov   r0,r4
...

caseD_2: ;XREF[2,0]: 00015c74,00015c84
.text:00015d84  mov   r1,#0x1
.text:00015d88  mov   r0,r4
...

caseD_3: ;XREF[2,0]: 00015c74,00015c88
.text:00015da8  mov   r0,r4
.text:00015dac  mov   r1,#0x1
...
```

**Listing 2: Disassemble result of base64 by NSDA**

## 6 DISCUSSION

**Performance Degradation in Loadstar NS1 and NS2 Compared to Ghidra.** We observe that in NS1 and NS2, Ghidra already achieves strong performance (F1 of 0.9875). This indicates that Ghidra has already discovered the vast majority of code through control flow analysis. The amount of undiscovered code is relatively small, and under such conditions, our current simple rules may struggle to identify these remaining code blocks. Since most code that can be discovered by our rules has already been found by Ghidra, any further exploration by the model based on these rules is likely to result in incorrect predictions.

In such cases, more targeted rules may be necessary to detect the missed code blocks. Because the model's additional exploration tends to be erroneous, the final results oscillate slightly below those of the symbolic methods. However, as shown in Table 4, the difference is minimal (within 0.001), and thus entirely acceptable.

**Pre-trained Model for Better Classification.** In the current implementation, each training iteration instantiates a new multilayer perceptron (MLP) [36] to classify the feature vector of each block. There are two potential directions for improvement. First, instead of training a completely new model in every iteration, we could pre-train a model on standard binaries for the code/data block classification task, and then fine-tune it within our LTN training process. This would likely yield more stable and accurate results compared to the current approach.

Second, more sophisticated techniques could be employed to generate code block embeddings. For instance, Loadstar [7] utilizes a simplified version of the PalmTree [26] model to produce instruction-level embeddings. However, such architecture-dependent embedding layers compromise our architecture-agnostic design by requiring retraining for every new ISA. To preserve our framework's versatility, we adopt a lightweight embedding strategy inspired by Gemini, tailored specifically for our task. This approach ensures that NSDA remains computationally efficient while capturing essential block-level semantics without relying on architecture-specific features.

**Limitation and Future Work.** A key limitation of the current work is the lack of explicit support for the ARM Thumb instruction set. Although Ghidra theoretically supports mixed ARM/Thumb disassembly, our experimental results on the mixed-mode dataset were suboptimal. Notably, while the Loadstar dataset is ostensibly ARM-only, enabling mixed-mode disassembly in Ghidra causes significant portions of the code to be interpreted as Thumb instructions. This behavior typically indicates the presence of explicit mode-switching mechanisms, such as branch-and-exchange (BX) or branch-with-link-and-exchange (BLX) instructions, or indirect modifications to the CPSR T-bit via load instructions targeting the program counter.

This inconsistency suggests two possibilities: (1) Ghidra's mixed-mode heuristics may be prone to false positives, incorrectly decoding ARM regions as Thumb; or (2) the Loadstar dataset contains latent Thumb-mode segments that challenge its "ARM-only" assumption. Due to the lack of ground-truth Thumb labels in the original dataset, we restricted our scope to ARM-only disassembly, leaving comprehensive Thumb-mode support for future work.

A promising path forward involves extending NSDA to support a tri-state classification: ARMCode, ThumbCode, and DataBlock. In this configuration, each block $b$ would be governed by a mutual exclusivity constraint:

$$\text{ARMCode}(b) + \text{ThumbCode}(b) + \text{DataBlock}(b) = 1$$

Furthermore, we intend to introduce predicates to capture cross-mode control transfers facilitated by BX/BLX instructions, alongside specialized logical rules to model the propagation of the TMode state across the control-flow graph.

**Applying Neurosymbolic AI to Other Security Tasks.** During our study of LTNs, we also considered their potential applications to broader security domains. Two directions emerged from our exploration: malware detection and malware classification.

For malware detection, both symbolic and neural approaches have been studied extensively [4, 16]. Symbolic methods [22, 29, 38] based on handcrafted rules are typically simple and efficient but often lack robustness. In contrast, neural network approaches [10, 35] are generally more robust but can be easily circumvented by adversarial attacks. We envision that neurosymbolic AI could provide a middle ground, where logical constraints are enforced during online training to regularize the model and make evasion by attackers significantly more difficult.

For malware classification, the idea builds upon detection by further distinguishing between malware families. Logical constraints could be introduced to guide clustering and classification, thereby enabling multi-class categorization of malware samples.

## 7 CONCLUSION

In this paper, we propose a neurosymbolic AI-based disassembly framework and implement a prototype, NSDA, which integrates symbolic logic reasoning with neural learning to solve complex disassembly tasks. We also present a dataset based on *coreutils*, specifically designed to evaluate disassembly across diverse architectures and compilers. Our evaluation demonstrates that among all evaluated disassemblers, NSDA and Ghidra provide the strongest support for a wide range of ISAs. NSDA outperforms Ghidra, achieving up to a 27% improvement in accuracy with minimal computational overhead. Furthermore, NSDA maintains competitive performance against specialized symbolic or neural disassemblers while offering superior architectural flexibility. These results indicate that NSDA is robust, effective, and practical for disassembling various binary formats. More broadly, our work showcases the significant potential of neurosymbolic AI to address fundamental challenges related to logic reasoning within the cybersecurity domain.

## ACKNOWLEDGMENTS

This paper was edited for grammar using ChatGPT and Google Gemini.

## REFERENCES


[1] [n. d.]. Ghidra – Software Reverse Engineering Framework. https://www.nsa.gov/resources/everyone/ghidra/.

[2] [n. d.]. The IDA Disassembler and Debugger. https://www.hex-rays.com/products/ida/.

[3] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. 2017. ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (Orlando, FL, USA) *(ACSAC '17)*. Association for Computing Machinery, New York, NY, USA, 437–448. https://doi.org/10.1145/3134600.3134618

[4] Ömer Aslan Aslan and Refik Samet. 2020. A Comprehensive Review on Malware Detection Approaches. *IEEE Access* 8 (2020), 6249–6271. https://doi.org/10.1109/ACCESS.2019.2963724

[5] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. 2022. Logic Tensor Networks. *Artificial Intelligence* 303 (Feb. 2022), 103649. https://doi.org/10.1016/j.artint.2021.103649

[6] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. https://doi.org/10.14722/ndss.2018.23304

[7] Hadjer Benkraouda, Nirav Diwan, and Gang Wang. 2025. You Can't Judge a Binary by Its Header: Data-Code Separation for Non-Standard ARM Binaries Using Pseudo Labels. 3727–3745. https://doi.org/10.1109/SP61157.2025.00036

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: a binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469.

[9] Tommaso Carraro. 2023. LTNtorch: PyTorch implementation of Logic Tensor Networks. Zenodo. https://doi.org/10.5281/zenodo.7778157

[10] Chia-Mei Chen, Shi-Hao Wang, Dan-Wei Wen, Gu-Hsin Lai, and Ming-Kung Sun. 2019. Applying Convolutional Neural Network for Malware Detection. In *2019 IEEE 10th International Conference on Awareness Science and Technology (iCAST)*. 1–5. https://doi.org/10.1109/ICAwST.2019.8923568

[11] Brandon C. Colelough and William Regli. 2025. Neuro-Symbolic AI in 2024: A Systematic Review. arXiv:2501.05435 [cs.AI] https://arxiv.org/abs/2501.05435

[12] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. https://doi.org/10.14722/ndss.2016.23185

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

[14] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium*. 1075–1092.

[15] Free Software Foundation. 2025. GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Version 11.4.0.

[16] Matthew G. Gaber, Mohiuddin Ahmed, and Helge Janicke. 2024. Malware Detection with Artificial Intelligence: A Systematic Literature Review. *ACM Comput. Surv.* 56, 6, Article 148 (Jan. 2024), 33 pages. https://doi.org/10.1145/3638552

[17] GNU Coreutils team and Free Software Foundation, Inc. Ongoing development. GNU Core Utilities (Coreutils). urlhttps://www.gnu.org/software/coreutils/. Accessed: May 18, 2026.

[18] GNU Project. 1983. Objdump. https://web.mit.edu/gnu/doc/html/binutils_5.html.

[19] Google. 2026. BoringSSL. https://github.com/google/boringssl. Accessed: 2026-04-29.

[20] Google. 2026. Chromium. https://www.chromium.org/. Accessed: 2026-04-29.

[21] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). https://api.semanticscholar.org/CorpusID:6628106

[22] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. 2009. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) *(SSYM'09)*. USENIX Association, USA, 351–366.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[24] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. https://doi.org/10.1145/3649828

[25] Haonan Li, Hang Zhang, Kexin Pei, and Zhiyun Qian. 2025. The Hitchhiker's Guide to Program Analysis, Part II: Deep Thoughts by LLMs. arXiv:2504.11711 [cs.SE] https://arxiv.org/abs/2504.11711

[26] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *ACM CCS*.

[27] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=9LdJDU7E91

[28] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

[29] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. 231–245. https://doi.org/10.1109/SP.2007.17

[30] National Institute of Standards and Technology. 2015. *Secure Hash Standard (SHS)*. Technical Report FIPS PUB 180-4. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.180-4

[31] KarlE. Nelson and MartinK. Scherer. 2020. JPype. https://doi.org/10.11578/dc.20201021.3

[32] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. 2022. Ground Truth for Binary Disassembly is Not Easy. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2479–2495. https://www.usenix.org/conference/usenixsecurity22/presentation/pang-chengbin

[33] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, robust disassembly with transfer learning. In *NDSS*.

[34] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R. Sekar. 2023. SAFER: Efficient and Error-Tolerant Binary Instrumentation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1451–1468. https://www.usenix.org/conference/usenixsecurity23/presentation/priyadarshan

[35] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2017. Malware Detection by Eating a Whole EXE. (10 2017). https://doi.org/10.48550/arXiv.1710.09435

[36] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65 6 (1958), 386–408. https://api.semanticscholar.org/CorpusID:12781225

[37] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. *Learning representations by back-propagating errors*. MIT Press, Cambridge, MA, USA, 696–699.

[38] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. 2001. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. 38–49. https://doi.org/10.1109/SECPRI.2001.924286

[39] Amit Sheth, Kaushik Roy, and Manas Gaur. 2023. Neurosymbolic AI – Why, What, and How. arXiv:2305.00813 [cs.AI] https://arxiv.org/abs/2305.00813

[40] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. https://doi.org/10.14722/ndss.2015.23294

[41] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[42] Raphael Springer, Alexander Schmitz, Artur Leinweber, Tobias Urban, and Christian Dietrich. 2025. Padding Matters - Exploring Function Detection

in PE Files: Data/Toolset paper. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy* (Pittsburgh, PA, USA) *(CODASPY '25)*. Association for Computing Machinery, New York, NY, USA, 179–184. https://doi.org/10.1145/3714393.3726003

[43] Radare2 Team. 2017. *Radare2 Book*. GitHub.

[44] The OpenSSL Project. 2026. *OpenSSL: Cryptography and SSL/TLS Toolkit*. https://www.openssl.org/ Accessed: April 21, 2026.

[45] Chengpeng Wang, Yifei Gao, Wuqi Zhang, Xuwei Liu, Qingkai Shi, and Xiangyu Zhang. 2024. LLMSA: A Compositional Neuro-Symbolic Approach to Compilation-free and Customizable Static Analysis. arXiv:2412.14399 [cs.PL] https://arxiv.org/abs/2412.14399

[46] Michael Wang, Kexin Pei, and Armando Solar-Lezama. 2025. CALLME: Call Graph Augmentation with Large Language Models for Javascript. In *Second Conference on Language Modeling*. https://openreview.net/forum?id=xZi2rMUcAO

[47] Peicheng Wang, Monika Santra, Mingyu Liu, Cong Sun, Dongrui Zeng, and Gang Tan. 2025. Disa: Accurate Learning-based Static Disassembly with Attentions. arXiv:2507.07246 [cs.CR] https://arxiv.org/abs/2507.07246

[48] Richard Wartell, Yan Zhou, Kevin Hamlen, and Murat Kantarcioglu. 2014. Shingled Graph Disassembly: Finding the Undecideable Path. 273–285. https://doi.org/10.1007/978-3-319-06608-0_23

[49] Lambang Akbar Wijayadi, Yuancheng Jiang, Roland H.C. Yap, Zhenkai Liang, and Zhuohao Liu. 2025. Evaluating Disassembly Errors With Only Binaries *(ASIA CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1741–1755. https://doi.org/10.1145/3708821.3733884

[50] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 363–376. https://doi.org/10.1145/3133956.3134018

[51] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *USENIX Security Symposium*.

# A OPEN SCIENCE

For the purpose of the double-blind review process, our codebase and datasets have been made available in several anonymous GitHub repositories. These resources will remain accessible throughout the review period. Upon acceptance of the paper, we will provide permanent links to the official public repositories for the research community.

- `NSDA` code base: https://anonymous.4open.science/r/NSDA-5762/
- Datasets:
  - Coreutils - ARM32 - gcc: https://anonymous.4open.science/r/coreutils-arm-1425/
  - Coreutils - MIPS - gcc: https://anonymous.4open.science/r/coreutils-mips-F281/
  - Coreutils - ARM32 - LLVM: https://anonymous.4open.science/r/coreutils-arm-llvm-C269/
  - Coreutils - MIPS - LLVM: https://anonymous.4open.science/r/coreutils-mips-llvm-0357/

## B OPENSSL EXPERIMENT

Pang et al. [32] note that obtaining the disassembly ground truth for the OpenSSL [44] project is challenging due to complex constructs, such as hand-written assembly code and data-in-code. We observe that Ghidra often misses these functions when function definition symbols are absent or when data-in-code causes decoding failures.

Using the OpenSSL binary (version 1.1.0l) from the x86-sok dataset [32] as our target, we examine a failure case involving hand-written assembly to illustrate Ghidra's limitations and evaluate whether NSDA can improve results in such scenarios. Our target function is 'padlock_sha256_blocks()', which exemplifies how missing function definition symbols can cause Ghidra to ignore an entire function in its disassembly results. As shown in Listing 3, this function is implemented in '/engines/asm/e_padlock-x86_64.pl' using x86_64 assembly code.

```
.globl  padlock_sha256_blocks
.type   padlock_sha256_blocks,@function,3
.align  16
padlock_sha256_blocks:
    mov %rdx,%rcx
    mov %rdi,%rdx       # put aside %rdi
    movups  (%rdi),%xmm0        # copy-in context
    sub $128+8,%rsp
    movups  16(%rdi),%xmm1
    movaps  %xmm0,(%rsp)
    mov %rsp,%rdi
    movaps  %xmm1,16(%rsp)
    mov $-1,%rax
    .byte   0xf3,0x0f,0xa6,0xd0 # rep xsha256
    movaps  (%rsp),%xmm0
    movaps  16(%rsp),%xmm1
    add $128+8,%rsp
    movups  %xmm0,(%rdx)        # copy-out context
    movups  %xmm1,16(%%rdx)
    ret
.size   padlock_sha256_blocks,.-padlock_sha256_blocks
```

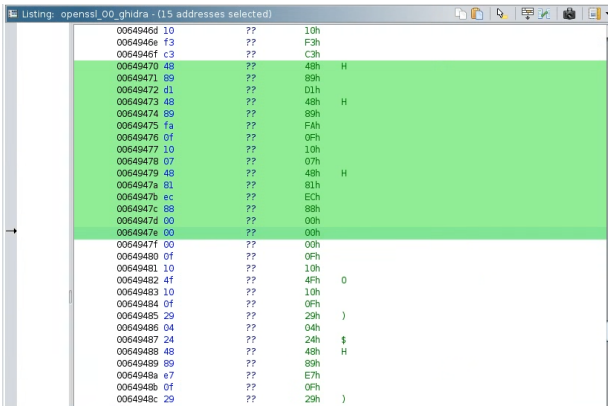
**Listing 3: OpenSSL Padlock SHA256 Assembly Snippet**

A four-byte array is embedded within the code, followed by the comment "rep xsha256". Ghidra successfully disassembles this function in the non-stripped OpenSSL binary, where it is explicitly defined via the '@function' directive. However, because these symbols are removed in the stripped binary, Ghidra fails to recognize this code section (as shown at 0x649470 in Figure 4a). In contrast, NSDA successfully recovers this function, as illustrated in Figure 4b.

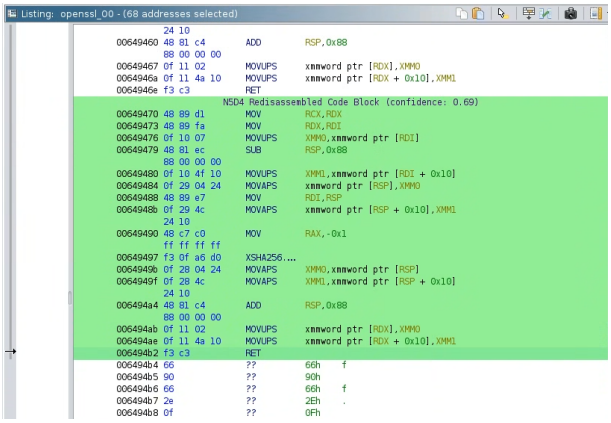
**Table 9: Processing Time Breakdown for OpenSSL (seconds)**

| Metric | Time (seconds) |
|---|---|
| Pre-processing | 568.246 |
| Training | 97.797 |
| Redisassembly | 713.773 |
| **Total** | **1379.816** |

Inspired by this case study, we evaluate whether NSDA can improve disassembly results generally for OpenSSL binaries in such scenarios. We utilize the six OpenSSL binaries from the x86-sok [32] dataset, compiled across six optimization levels (O0–O3, Os, and Ofast), to test the effectiveness of NSDA when processing complex




**(a) By Ghidra.**




**(b) By NSDA.**

**Figure 4: Padlock SHA256 Disassembly Result.**

x86_64 binaries. For completeness, we include all baseline methods from our primary evaluation in this experiment.

The accuracy results are summarized in Table 10. We find that NSDA outperforms all baselines in terms of F1 score and recall. Consistent with our case study, Ghidra fails to identify certain code sections, leading to lower recall. Notably, Datalog Disassembly (DDisasm) [14] performs poorly in this evaluation, suggesting that DDisasm may face challenges when handling large and complex binaries.

Efficiency is also a key consideration in our evaluation. Since the sizes of these six binaries are comparable, we report the average pre-processing, training, and re-disassembly times in Table 9. Notably, training time scales significantly more slowly than pre-processing and re-disassembly. For context, the 'base64' binary (compiled by LLVM for MIPS) is approximately 81 KB and requires 25s for pre-processing, 26s for training, and 15s for re-disassembly. In contrast, the OpenSSL binary used here is approximately 4 MB. While pre-processing and re-disassembly times scale nearly linearly with file size, the training time only increases fourfold despite a nearly 50-fold increase in file size. This indicates that the overhead associated with LTN framework is quite small.

**Table 10: Precision (P), Recall (R), and F1 score of NSDA and baselines**

| Dataset | Ghidra | | | DDisasm | | | FuzzyNSDA | | | NSDA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | R | F1 | P | R | F1 | P | R | F1 | P | R | F1 |
| openssl_O0 | **0.9935** | 0.8956 | 0.9420 | 0.9926 | 0.2687 | 0.4229 | **0.9935** | 0.8956 | 0.9420 | 0.9699 | **0.9983** | **0.9839** |
| openssl_O1 | **0.9914** | 0.8921 | 0.9391 | 0.9904 | 0.2735 | 0.4286 | **0.9914** | 0.8921 | 0.9391 | 0.9566 | **0.9970** | **0.9764** |
| openssl_O2 | **0.9856** | 0.8844 | 0.9322 | 0.9542 | 0.2726 | 0.4241 | **0.9856** | 0.8844 | 0.9322 | 0.9378 | **0.9970** | **0.9664** |
| openssl_O3 | **0.9864** | 0.8879 | 0.9345 | 0.9586 | 0.2725 | 0.4244 | **0.9864** | 0.8879 | 0.9345 | 0.9427 | **0.9972** | **0.9692** |
| openssl_Of | **0.9864** | 0.8880 | 0.9346 | 0.9586 | 0.2725 | 0.4244 | **0.9864** | 0.8880 | 0.9346 | 0.9427 | **0.9972** | **0.9692** |
| openssl_Os | **0.9895** | 0.8787 | 0.9308 | 0.9887 | 0.2744 | 0.4296 | **0.9895** | 0.8787 | 0.9308 | 0.9519 | **0.9963** | **0.9736** |
| Average | **0.9888** | 0.8878 | 0.9356 | 0.9738 | 0.2724 | 0.4257 | **0.9888** | 0.8878 | 0.9356 | 0.9503 | **0.9972** | **0.9731** |

This experiment demonstrates that NSDA can effectively process large, complex x86_64 binaries, maintaining its efficacy even in the presence of hand-written assembly and embedded data. Furthermore, these results support our earlier claim that NSDA is architecture-agnostic, supporting various architectures without requiring modifications to its underlying implementation.

# C CHROMIUM EXPERIMENT

We evaluated NSDA on Chromium [20] using the dataset provided by Springer et al. [42]. This dataset focuses on function entry point recovery, assessing disassembler capabilities at the function level. The Chromium binary is in PE format, compiled by LLVM, and published by Google (version 109.0.5415.0).

We observed that Ghidra requires approximately nine hours to complete its initial analysis of the binary. Maintaining the original pipeline would incur a prohibitive time overhead prior to training, as this analysis is performed during the pre-processing stage of each iteration. Furthermore, Ghidra's pseudo-analysis module performs inefficiently when processing large binaries in a single pass. Consequently, we adjusted our pipeline to accommodate this large binary.

Throughout the process, Ghidra executes its auto-analysis only once. Subsequently, rather than invoking full auto-analysis at each iteration, we restrict Ghidra to analyze only the modified sections. Although this constrains the scope of Ghidra's analysis, it substantially reduces the processing time for large binaries.

To address the inefficiency of Ghidra's pseudo-disassembly module, we divide the entire binary into 1 MB segments. We then sequentially perform pre-processing, training, and re-disassembly on each individual segment until all segments are processed.

We also evaluated function entry point recovery in this experiment to align with the dataset's configuration. The results are presented in Table 7, and the corresponding time breakdown is detailed in Table 8. We employed only Ghidra [1] as the baseline in this experiment, as DDisasm [14] encountered out-of-memory errors on our 128 GB machine, and FuzzyNSDA's performance is closely comparable to that of NSDA. Therefore, to optimize evaluation time, we exclusively retained Ghidra.

As anticipated, because Ghidra already captures the majority of functions within the binary, the improvement yielded by NSDA is relatively limited. Specifically, out of a total of 542,902 functions, NSDA identified 82 more functions than Ghidra, comprising 50 true positives and 32 false positives. However, this does not imply that all false positives are merely data blocks. Because the ground truth exclusively labels function entry points, some of these false positives are actually valid code instructions that simply do not serve as entry points. Consequently, they are strictly classified as false positives within our evaluation metrics. Therefore, these reported results represent a lower bound for the overall performance of NSDA.

The training efficiency of NSDA is also highly favorable, requiring only 12 seconds to train a 1 MB segment. Given that the total size of the binary is 179 MB, the overall training time for the entire binary is estimated at approximately 2,200 seconds, which constitutes a minimal overhead relative to the processing time required by Ghidra.

Due to the lack of non-stripped binaries in the datasets, we are unable to identify all hand-written assembly code within the binary. Consequently, we focus on the BoringSSL [19] library within Chromium [20], which is a fork of OpenSSL [44]. Based on our prior experience with OpenSSL, we anticipated the presence of hand-written assembly code in this library.

Specifically, we examine the code located in 'src/third_party/boringssl/win-x86_64/crypto'. We manually searched for these functions by identifying distinctive algorithmic patterns in memory. There are 17 source files written in assembly code, 15 of which were successfully disassembled by Ghidra, while two (`aes128gcmsiv-x86_64.asm` and `chacha20_poly1305_x86_64.asm`) could not be found in the memory search results. Therefore, we believe these identified hand-written assembly code snippets are handled by Ghidra.

In certain cryptographic algorithms, embedded data—such as the K-table in SHA-256 [30]—is located between functions. Ghidra handles these tables effectively. Following the analysis by NSDA, such embedded data is correctly maintained as data. This demonstrates NSDA's capability to accurately distinguish between code and data, even within hand-written assembly code.