USENIX Security '25 Artifact Appendix: Tady: A Neural Disassembler without Structural Constraint Violations

Siliang Qin[1,2], Fengrui Yang[3], Hao Wang[3], Bolun Zhang[1,2], Zeyu Gao [3], Chao Zhang [3], Kai Chen [1,2]

[1] *Institute of Information Engineering, Chinese Academy of Sciences, China*

[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*

[3]*Tsinghua University, China*

{qinsiliang, zhangbolun, chenkai}@iie.ac.cn

{yangfr23, hao-wang20, gaozy22}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn

A Artifact Appendix

A.1 Abstract

The artifact, Tady, is a novel neural network-based disassembler designed to address structural constraint violations commonly found in the output of existing disassemblers. It consists of a neural model with a hybrid local-global attention mechanism to better understand code context and a post-processing algorithm that uses a Post-Dominator Tree (PDT) to enforce structural consistency. The artifact includes the source code for the Tady model, the PDT-based error detection and pruning algorithms, evaluation scripts, and the datasets used in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All analyzed binaries are sourced from public academic datasets. There are no destructive steps, and the artifact does not require elevated privileges or disabling of security mechanisms. The research follows the principles of the Menlo Report and USENIX Security's ethical guidelines. The dual-use potential is acknowledged, but the focus is on advancing defensive binary analysis capabilities.

A.2.2 How to access

The artifact is publicly available under an open-source license. A snapshot of the evaluated version is permanently archived on Zenodo with the DOI: 10.5281/zenodo.15541311, which can be accessed from https://doi.org/10.5281/zenodo.15541311. The latest version of the source code is maintained and updated at the following GitHub repository: https://github.com/5c4lar/tady, where the commit *014cd1d* is used for artifact evaluation.

A.2.3 Hardware dependencies

The experiments were conducted on a machine with the following specifications:

- **CPU:** Intel Core i9-12900K
- **GPU:** NVIDIA RTX A6000 Ada generation (A GPU is required for model training and inference)
- **RAM:** 64GB
- **Storage:** 1TB NVMe SSD

A machine with similar or better specifications is recommended for reproducing the results in a timely manner. Since the intermediate results generated for the datasets can get large, we recommend at least 256GB storage are available before reproducing the results. Machine with NVIDIA GPUs other then this specific model should also work smoothly.

A.2.4 Software dependencies

The artifact can be run on host or within a Docker container.

- **Operating System:** Ubuntu 24.04.
- **Core Dependencies:**
  - C++ toolchain (build-essential)
  - LLVM 18 (llvm-dev)
  - Boost Graph Library (libboost-graph-dev)
  - Python 3 development headers (libpython3-dev)
- **Python Environment:** Managed via *uv*. Key packages include Flax, JAX, TensorFlow, NumPy.
- **Baselines:** Reproducing baseline results requires installing IDA Pro (v9.1), Ghidra (v11.3.2), ddisasm (v1.9.0), DeepDi, and XDA, each with its own specific environment (e.g., conda for XDA).

- **Docker:** A Dockerfile is provided to build a container with all necessary dependencies for Tady.

The README.md file provides detailed instructions for setting up both host and Docker environments. The artifact include the disassembly results of the baselines, which can be used if the baseline software itself is not accessible.

A.2.5 Benchmarks

The evaluation uses several public and custom datasets, all of which are provided and can be generated with the artifact's scripts. The primary datasets are:

- **Pangine:** Binaries compiled with various compilers and optimizations, with labels extracted during compilation.
- **Assemblage:** Open-source Windows programs.
- **x86-sok:** Linux binaries, including both common utils and other more complex applications.
- **rw:** Real-World dataset created following the DeepDi protocol using x86-sok's toolchain.
- **quarks:** Binaries obfuscated with Tigress and OLLVM under various obfuscation settings.
- **obf-benchmark:** 11 binaries obfuscated with binobf.

The artifact provides scripts to download, preprocess, and format these datasets. Pre-processed datasets are also available for download to expedite evaluation.

A.3 Set-up

A.3.1 Installation

The following steps describe the installation process using the provided Docker environment.

1. Use the `tady-main.zip` from zenodo of clone the artifact repository:
   `git clone https://github.com/5c4lar/tady.git && cd tady`
2. Build the Docker image:
   `docker build -t tady -f docker/Dockerfile .`
3. Download and extract the pre-processed datasets, baselines' results and trained models. The artifact provides the datasets with *bin.tar.gz* and *gt_npz.tar.gz*. This avoids the lengthy process of generating datasets from scratch. Baselines' results are given in *eval_strip_baselines.tar.gz*. The models are provided in *models.tar.gz*.

```
tar -xzvf bin.tar.gz -C data
tar -xzvf gt_npz.tar.gz -C data
tar -xzvf eval_strip_baselines.tar.gz \
-C data
tar -xzvf models.tar.gz
```

At this point, the environment is ready for evaluation.

A.3.2 Basic Test

To verify the installation, run a simple disassembly test on a the VMProtect obfuscated binary in appendix. This test will use the Tady model to disassemble a file and then run the pruning algorithm on the output, For evaluation purpose, the ground-truth is used to evaluate the accuracy after pruning.

1. Start the tensorflow serving on the host:

```
docker run --rm --gpus device=0 \
-p 8500:8500 \
-v  $PWD/models/tf_models:/models -t \
--name tensorflow-serving \
tensorflow/serving:latest-gpu \
--xla_gpu_compilation_enabled=true \
--enable_batching=true \
--batching_parameters_file=\
/models/batching.conf \
--model_config_file=/models/model.conf
```

2. Enter the Tady docker container (if using Docker).
3. Run the evaluation script on a single file:

```
uv run -m tady.infer \
--path data/obf/TestApp.vmp.exe \
--model instruction_cpp_pangine_\
lite_all_64lw_64rw_16h_2l_prev000 \
--section_name .vmp0 \
--output_path \
data/obf/tady/TestApp.vmp.exe.npz \
--seq_len 569038 --batch_size 1
```

4. Run the pruning script on the result:

```
uv run -m tady.prune \
--gt data/obf/TestApp.vmp.exe.npz \
--pred data/obf/tady/TestApp.vmp.exe.npz
```

A successful run will produce a numpy file at `data/obf/tady/TestApp.vmp.exe.npz`, containing the disassembly results. The *tady.prune* script will show the precision, recall, F1 score, false positives, false in the console. The first run of the *tady.infer* will be slower than the following runs because of tensorflow-serving's warmup.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Tady's model achieves high instruction-level accuracy (Precision, Recall, F1 score), outperforming or performing competitively with state-of-the-art disassemblers on standard and obfuscated benchmarks. This is proven by experiment (E1), with results in Table 3 of the paper.

(C2): Tady's post-processing pruning algorithm effectively eliminates all detected structural constraint violations from the model's output, improving consistency while maintaining or improving F1 score. This is proven by experimen (E1), with results in Tables 3.

(C3): Existing disassemblers, including rule-based tools and neural models, frequently produce outputs with structural constraint violations, and these violations are exacerbated by obfuscation. Tady's algorithm can systematically detect these errors. This is proven by experiment (E2), with results in Tables 1 and 2 of the paper.

(C4): The entire Tady workflow is highly efficient, with time usage scaling linearly with the size of the input binary. This is proven by experiment (E3), with results in Figures 8, 9, 10, and 11.

(C5): The novel components of Tady's model, specifically the Masked Sliding Window Attention (MSWA) and Message Passing (MP), are critical to its performance. This is proven with the results of experiments (E1, E2), with results in Tables 4 and 5.

A.4.2 Experiments

Before measuring the results, follow the instructions in the Eval.Tady section in the README to generate the disassembly of Tady. The variants in the ablation study correspond to the combinations of configurations: `attention=lite,sliding connections=none,all`. Sample outputs can be found in `artifacts.tar.gz`.

(E1): Pruning and Accuracy Evaluation [30 minutes]: This experiment runs the pruning algorithm on all disassembly results and evaluates the F1 score, precision, and recall before and after pruning.
Preparation: Ensure all the disassembly results are available either by downloading from the artifacts or reproduced according to the README.
Execution: Run the following scripts.

```
uv run scripts/experiments/prune.py -m \
test_dataset=obf-benchmark,rw,x86_dataset,\
pangine,quarks,assemblage \
process=24 num_samples=1000
uv run scripts/experiments/collect_stat.py
uv run scripts/experiments/prune.py -m \
test_dataset=obf-benchmark,rw,x86_dataset,\
pangine,quarks,assemblage process=24 \
models="[gt]"
```

Results: `data/prune/all_prune_result.json`. The data in this file can be used to reproduce the accuracy results in Table 3 and Table 5, validating claims C1, C2 and C5.

(E2): Detecting Constraint Violations [30 minutes]: This experiment runs Tady's error detection algorithm over the outputs of various disassemblers (Tady, baselines) and dataset labels across all benchmarks.
Preparation: Ensure E1 has been completed and the intermediate results in `data/prune` are available. The errors are already detected in our pruning scripts.
Execution: Run the error statistics collection script. This script aggregate the error detection results.

```
uv run scripts/experiments/error\_stat.py
```

Results: `data/error_stat.json`, which contains the aggregated data used to generate Tables 1, 2, and 4 in the paper. This validates claim C3, C5.

(E3): Efficiency Benchmark [2 hours]: This experiment measures the runtime and memory usage of Tady. The time is mainly spent on slow baselines on large binaries.
Preparation: Environments to run the disassemblers.
Execution: Please follow the Efficiency section in README for instructions about this evaluation.
Results: Data of Figures 8, 9, 10, and 11, validating claim C4. The figures can be reproduced with the script `scripts/ablation/generate_uniform_plots.py` with the provided data in `argifacts.tar.gz`.

A.5 Notes on Reusability

The artifact is designed for reusability. Beyond reproducing the paper's results, it can be extended in several ways:

- **Applying to New Binaries:** The Tady inference pipeline (*tady.infer*) can be used to disassemble new, unseen x86/x86-64 binaries. The README provides an example of this in the VMProtect evaluation.
- **Training on New Datasets:** The training pipeline (*scripts/experiments/train.py*) can be adapted to train the Tady model on new, custom-labeled disassembly datasets. The data preprocessing scripts show how to convert ground truth into the required format.
- **Applying to other disassemblers** The error detection and pruning algorithms can be applied to other disassembly results. Examples on how to use the code can be found in the evaluation scripts.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.