You Can't Judge a Binary by Its Header: Data-Code Separation for Non-Standard ARM Binaries using Pseudo Labels

Hadjer Benkraouda
University of Illinois at Urbana-Champaign
hadjerb2@illinois.edu

Nirav Diwan
University of Illinois at Urbana-Champaign
ndiwan2@illinois.edu

Gang Wang
University of Illinois at Urbana-Champaign
gangw@illinois.edu

***Abstract*—Static binary analysis is critical to various security tasks such as vulnerability discovery and malware detection. In recent years, binary analysis has faced new challenges as vendors of the Internet of Things (IoT) and Industrial Control Systems (ICS) continue to introduce customized or *non-standard* binary formats that existing tools cannot readily process. Reverse-engineering each of the new formats is costly as it requires extensive expertise and analysts' time. In this paper, we investigate the first step to automate the analysis of non-standard binaries, which is to recognize the bytes representing "code" from "data" (i.e., data-code separation). We propose `Loadstar`, and its key idea is to use the abundant *labeled* data from standard binaries to train a classifier and adapt it for processing *unlabeled* non-standard binaries. We use a pseudo-label-based method for domain adaption and leverage knowledge-inspired rules for pseudo-label correction, which serves as the guardrail for the adaption process. A key advantage of the system is that it does not require labeling any non-standard binaries. Using three datasets of non-standard PLC binaries, we evaluate `Loadstar` and show it outperforms existing tools in terms of both accuracy and processing speed. We will share the tool (open source) with the community.**

1. Introduction

Static binary analysis is critical to a range of *security tasks* such as vulnerability discovery, malware detection, and control flow integrity protection [1]–[4]. In practice, software (including malware) of real-world systems is commonly presented in the form of binary executables without source code. The ability to parse and analyze binaries (e.g., disassembling a binary and reconstructing its control flow graph) is the precondition for the aforementioned security analytics tasks.

However, binary analysis has faced new challenges as the Internet of Things (IoT) and Industrial Control Systems (ICS) continue to introduce *non-standard binaries* to the ecosystem. These non-standard binaries do not follow well-documented formats and cannot be readily parsed by existing tools such as Ghidra [5], IDA Pro [6], and Radare2 [7]. Unlike standard binaries such as ELF, PE, and Mach-O that are organized into sections (e.g., `.text`, `.data`) with headers that list the section addresses, non-standard binaries can have different file formats for different devices and vary from one vendor to another. This happens for many reasons. First, IoT and ICS binaries are often compiled for atypical, specialized, or legacy operating systems (e.g., Nucleus or embedded Linux with a real-time patch) that do not conform to standard file formats. Second, due to the lack of standardization and the need to keep their software proprietary, IoT vendors often create customized file formats without public documentation. This results in unknown (non-standard) binary file formats that do not have a recognizable layout/configuration [8], [9]. Manually reverse engineering the format of each of these devices can have significant time costs.

Despite the extensive efforts to develop binary analysis tools [10]–[16], most of them cannot handle non-standard binaries. This is because existing tools commonly require parsing and preprocessing the input binaries to extract the code sections (e.g., based on known formats, using tools such as Ghidra [5] and IDA Pro [6]). In addition, some of them may even rely on information that is not easily extracted from non-standard binaries (e.g., headers, symbol tables, debug information). In our preliminary tests (Section 2.3), we attempted to use both commercial tools [5]–[7] and research-based tools [10]–[12] to load and analyze non-standard binaries. Most of the tools cannot proceed due to unrecognizable binary formats.

Goal and Method. In this paper, we investigate the very first step to analyze non-standard binaries, which is to parse the binaries to recognize bytes that represent "code" and "data" (i.e., data-code separation). This step is important for a range of downstream analyses (e.g., accurate disassembly, code similarity analysis, and function/signature detection) because most of these analyses need to take the code sections as inputs to perform further analysis. We propose to address the problem by training a machine learning classifier based on standard binaries, and then adapting the classifier to the target non-standard binary formats. To ensure practicality, the system has two main assumptions. First, the system does not require any labeled non-standard binary files (considering the high costs of manually reverse engineering non-standard binaries). Second, the system assumes the availability of labeled standard binaries, which can be easily achieved by

parsing standard binaries with existing tools (e.g., Ghidra) based on their known formats.

We design a system called `Loadstar`, based on two high-level intuitions. First, the code sections in binary executables follow language-specific patterns and semantics, but the data sections do not. Such differences can be reflected in the characteristics of the forcefully disassembled/decoded bytes. Second, the difference between *code* and *data* may not be the same between standard and non-standard formats; however, a set of rules inspired by domain knowledge can serve as the invariant to help the model adapt. Based on these intuitions, `Loadstar` is designed with two modules. The first module trains an initial classifier using *labeled* standard binaries. To do so, `Loadstar` takes raw binaries as inputs and forcefully decodes them into instruction sequences (via linear sweep). It trains a language model to embed these instructions and uses the labeled standard binaries to train the initial classifier (C_0). The second module performs domain adaptation using *unlabeled* non-standard binaries. The key idea is to use the initial classifier C_0 to produce pseudo labels on the non-standard binaries and then construct a set of domain-knowledge rules (e.g., *define-use chain*, *jump/branch destination*) to curate the pseudo labels. By iteratively training a new classifier (e.g., C_n) and performing pseudo-label correction, we gradually improve the classifier's performance on non-standard binaries.

Evaluation. Our evaluation is focused on non-standard Programmable Logic Controller (PLC) binaries under the ARM instruction set architecture. The rationale is that we need ground-truth labels for our evaluation (not training), and PLC binaries can be labeled with reasonable efforts, thanks to existing research (and reverse-engineering efforts) [17], [18]. In addition, PLC devices are widely used in industrial control environments [2], [19] and ARM is the most utilized ISA in the IoT/ICS space [8], [20]. Using three different datasets of non-standard binaries (two real-world datasets, and one synthetic dataset), we show that `Loadstar` outperforms existing methods in terms of both detection accuracy and inference speed. Our result confirms the effectiveness of pseudo-label correction while highlighting that not all domain-knowledge rules (especially statistical-based rules) are equally useful. The results show `Loadstar` is not only capable of accurately identifying data/code sections but can also capture *inline data* within code sections. `Loadstar` can process non-standard binaries with a throughput of 55KB/s on a CPU, which is orders of magnitude faster than XDA [12] and a BERT-based baseline. In addition, our transferability experiment shows the possibility of training a "global model" of `Loadstar` to handle diverse types of non-standard binaries.

Countributions. We have three key contributions.

- We propose `Loadstar`, a pseudo-label-based learning model to perform data-code separation for non-standard binaries. A key advantage is that its training does not require labeling any non-standard binaries.
- We evaluate `Loadstar` using three different datasets of non-standard PLC binaries, which shows the system's excellent accuracy and efficiency.
- Throughout our study, we labeled extensive ground-truth datasets of both standard and non-standard binaries. The ground-truth datasets and code will be made available for sharing with the community.

2. Background and Motivation

In this section, we describe the background for the data-code separation problem for non-standard binaries, followed by the motivation and the scope of this paper.

2.1. Standard vs. Non-standard Binaries

Standard Binaries. This type of binaries is compiled for typical operating systems such as Linux, Windows, and macOS, and follows known (standard) file formats such as ELF, PE, and Mach-O. Under each of these file formats, there is a clear description of sections, their functionality, and the type of data they store. Additionally, these files are usually equipped with headers that describe the exact locations of each of these sections. These files can be easily parsed by commercial binary analysis tools such as Ghidra [5], IDA Pro [6], and Radare2 [7]. Each of these commercial tools has *a limited number* of natively supported/recognized file formats (see the full list in Table 11 in Appendix A).

Non-Standard Binaries. These binaries (or binary blobs) usually do not have descriptive headers and cannot be parsed by existing binary analysis tools. Non-standard binaries are commonly seen in real-world IoT devices and industrial control systems. Various vendors create their own customized file formats in order to keep their software proprietary [8], [9]. Without considerable reverse-engineering efforts, commercial tools cannot parse/support these binaries.

2.2. Data-Code Separation Problem

When loading a binary file of a non-standard format, existing tools often fail to recognize or parse the file correctly. Figure 1 shows how Ghidra [5] and Radare2 [7] load a non-standard binary file, which represents two common ways in which existing tools handle such binaries. Ghidra loads it as a raw binary, which means that metadata such as sections (e.g. .text, .data) are not demarcated, and accurate disassembly is not feasible due to the lack of differentiation between code and data sections. Radare2 performs a linear sweep disassembly where it assumes all bytes are code. In both scenarios, the outputs do not separate data from code to support downstream analysis.

To address this problem, prior works have performed reverse engineering on certain customized/non-standard binary formats and then either built new tools [18] or modified existing tools [20], [21] to parse related files. Key challenges to the scalability of these approaches are due to (1) the extensive manual efforts and expertise required for reverse engineering, (2) the diversity/heterogeneity of IoT vendors (and their customized formats) existing on the market [8],

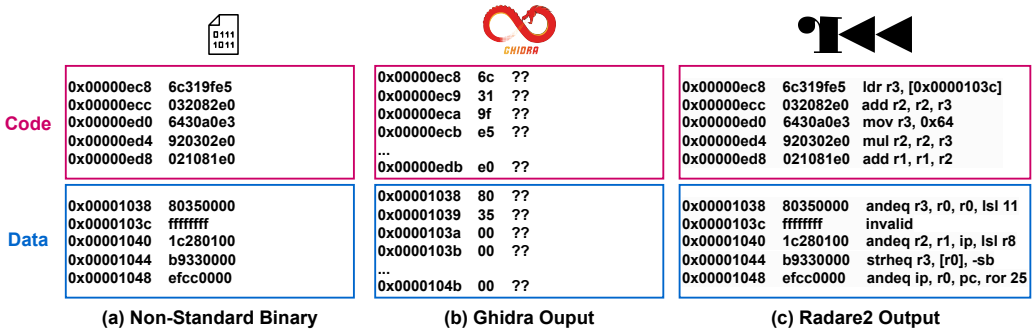


Figure 1: Outputs of `Ghidra` and `Radare2` when a non-standard binary is used as the input. We use red and blue to indicate the ground-truth code and data. `Ghidra` does not load the non-standard binary correctly, and `Radare2` treats everything as code to perform linear sweep. Neither can separate data from code.

and (3) the fact that new binary formats are continuing to be introduced by platform and compiler updates [17].

Goals. To this end, we investigate the problem of data-code separation for non-standard binaries, which is the first step for most of the binary analysis tasks [22]. The expected inputs are *raw binaries* of a non-standard format, and the expected outputs are labels on the bytes as either "code" or "data" (see example outputs in Figure 3(e)).

Inline Data. As part of our goal, we not only aim to discover the major data/code sections but also seek to discover "inline data" (i.e., data bytes mixed in with code instructions in the code sections). Such inline data is often used for jump tables and local constants. Identifying inline data is critical to downstream tasks such as accurate disassembly: if such inline data is mistakenly interpreted as code, disassembly may desynchronize the instruction stream or produce incorrect control flow graphs [11]. Figure 3(a) shows an example of a non-standard binary file. While there is a major data section at the end of the file (the large black region at the bottom), there exist smaller inline data regions in different parts of the file, mixed with the code instructions.

Assumptions. At the high level, we propose to address the problem by training and adapting a machine learning classifier. To ensure practicality, the system has two main assumptions. First, the system *does not require any labeled non-standard binaries*. This means analysts do not need to perform reverse-engineering on any of the target non-standard binaries to train our system. Second, the system assumes the availability of *labeled standard binaries*. Such data is abundant in practice because the format of standard binaries is known. We can easily construct a large dataset of "labeled standard binaries" by running them through an existing tool (e.g., Ghidra). Note that, for standard binaries, we only require labels on the data/code sections (coarse-grained) but not necessarily labels on inline data. This is considering that existing standard binary tools (e.g., Ghidra) may not be capable of labeling inline data.

Method	Loadable	Analyzable
DeepDi [10]	✗	✗
D-ARM [11]	✗	✗
XDA [12]	✓	✓
Ghidra [5]	✓	✗
IDA Pro [6]	✓	✗
Radare2 [7]	✓	✓

TABLE 1: Summary of applicability of existing methods to process non-standard binaries. "Loadable" means the method is able to load/take the raw binaries as input. "Analyzable" means the method can perform some analysis on raw binaries (no guarantee of the correctness of the results).

2.3. Applying Prior Work to Non-standard Binaries

Besides commercial tools such as Ghidra and IDA Pro, there have been solutions from the research community that tackle the problem of accurate disassembly. In the following, we discuss the possibility of adapting them to solve the data-code separation problem for non-standard binaries. We identify the three most relevant systems: XDA [12], DeepDi [10], and D-ARM [11], all of which are focused on disassembly for *standard binaries*. We summarize the applicability of tools on non-standard binaries (and other commercial tools) in Table 1.

DeepDi. DeepDi [10] uses superset disassembly to find all possible bytes an instruction can start from and then discard unlikely starting points. It first constructs an "Instruction Flow Graph" to capture different instruction relations. Then a Relational Graph Convolutional Network is used to propagate instruction embeddings for instruction classification. The tool is designed for *standard* binaries: (1) it uses assertions to check whether the files belong to two standard types (ELF and PE). (2) It relies on tools for standard binaries for pre-processing (`pefile` and `elftools`). These tools iterate through different sections to identify the code sections (by checking whether the execute flag is enabled) and then pass them to the disassembly function. We attempted to feed non-standard binaries to DeepDi (including its web tool for ARM binaries), but it failed to process the inputs (i.e., the assertion failed and then the program ended).

D-ARM. D-ARM [11] is a static analysis tool to interpret superset instructions. The information is then used to derive a weight value for each instruction to decide whether it is an ARM instruction (code), data bytes, or a Thumb instruction. The original implementation only works for ELF binaries (standard binaries). First, `readelf` is used to find the instruction length and get section information. Next, it uses flags to label code and data sections (and leverages offsets, start, and end addresses of sections). After the `.text` section and the symbol table are identified, further analysis is done on these sections only. This method cannot be easily adapted for non-standard binaries due to the lack of section information (or tools like `readelf`). When we attempted to feed non-standard binaries, the tool returned an error message indicating "unsupported file format."

XDA. XDA [12] is a transfer-learning-based disassembly framework. It takes raw binaries as the input, trains an embedding model with Masked Language Modeling to capture the interactions among byte sequences, and then fine-tunes the model for downstream tasks such as boundary discovery for functions and instructions. While XDA is designed with standard binaries in mind, it can be adapted to process non-standard binaries. When running XDA on non-standard binaries, we were able to pass the whole binary file to it without triggering an error message. Although data-code separation is a different problem from recovering instruction boundaries, we suspect their results may overlap. Intuitively, XDA, trained to detect instruction boundaries, should be able to detect data bytes as "not within an instruction."

This led us to adapt XDA for the data-code separation task (see details in Appendix B). The basic idea is to pre-train XDA for ARM binaries (given our non-standard binaries datasets contain ARM binaries). Then we fine-tune XDA for the instruction boundary discovery task using standard binaries (given this process requires labels). Finally, we test the fine-tuned model on non-standard binaries. The results are reported in Section 5 (Table 5). We show that XDA can distinguish code sections from data sections with moderate accuracy (F1=0.825) but is not highly accurate. Also, it cannot accurately capture the smaller inline data with an F1* of 0.195[1]. An example of XDA's output is visualized in Figure 3(b) which is visibly different from the "ground truth" in Figure 3(a). The result suggests that a new method is needed to solve this problem.

2.4. Scope of This Paper

While the ultimate goal is to automate the analysis of non-standard binaries, as the *inaugural step*, this paper focuses on non-standard binaries for Programmable Logic Controllers (PLC) under the ARM instruction set architecture. The rationale is the following. *First*, the *evaluation* of our method requires "ground truth" labels on the non-standard binaries (even though the training does not require such labels). As stated before, non-standard binaries often have vendor-specific customized file formats with limited to no description. Reverse-engineering those binaries requires significant manual efforts [23]. As such, we focus on PLC binaries that have been manually analyzed by researchers and practitioners, making it feasible to label them for our evaluation. *Second*, we choose PLC binaries for their popularity and impact [24]. PLCs are widely used by Industrial Control Systems for nuclear power plants, chemical plants, critical manufacturing, and transportation systems [19]. While PLC binaries contain a header, the header does not include information about the section offset addresses or classes to separate data from code. *Third*, we focus on ARM binaries because it is the most utilized instruction set architecture (ISA) in the IoT/ICS firmware/binary space [8], [20]. ARM is a Reduced Instruction Set Computer (RISC) ISA with fixed-length instructions. We will discuss how our method can potentially be extended to other ISAs in Section 6.

3. High-level System Design

We design a system to perform data-code separation for non-standard binaries. We call the system "`Loadstar`"[2]. In the following, we describe our design goals and important intuitions behind the design choices and then present a walk-through of the system workflow.

3.1. Design Goals

We design `Loadstar` with four high-level goals in mind.

First, no label requirement for non-standard binaries. The system uses unlabeled non-standard binaries, and there is no requirement for (manually) reverse-engineering them.

Second, minimal pre-processing needed. To minimize the dependency on other pre-processing tools (see reasoning in Section 2.3), we design the system to directly take a binary file/blob as the input.

Third, high accuracy. We expect the system to classify data from code with high accuracy to support downstream analyses. This includes accurately classifying inline data too.

Fourth, reasonable efficiency. The system should be able to produce data/code labels for the binary files efficiently, to minimize the waiting time of analysts. For example, a binary file should be processed and analyzed within seconds.

3.2. Overview of `Loadstar`

The training process of `Loadstar` is illustrated in Figure 2. It contains two phases: (1) initialization (training an initial classifier using *labeled standard binaries* only), and (2) iterative training (adapting the classifier for non-standard binaries using *unlabeled non-standard binaries*).

Intuitions Behind the Designs. At a high level, binary executables have language-like structures and semantics, especially after they are dissembled. However, they are also different from natural languages and carry patterns introduced

1. F1* is a metric to capture the accuracy of inline data detection. See its definition in Section 5.1.

2. A wordplay between "Lodestar" and "Binary Loader"

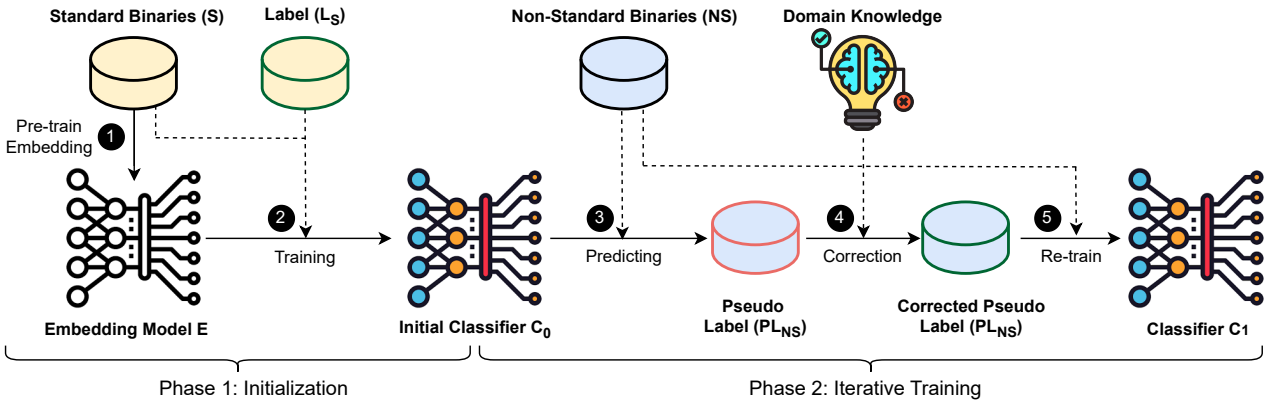


Figure 2: Overview of Loadstar.

by specific compilers and binary formats. On one hand, we take advantage of language models to provide the initial embedding for binary instructions. On the other hand, we leverage *domain knowledge* of binary executables to guide the learning process and perform domain adaption. The high-level intuitions of Loadstar are two-fold: First, while the *code* sections follow the language-specific patterns/semantics, the *data* sections do not. Such differences can be reflected in the characteristics of forcefully decoded instructions. Second, the difference between *code* and *data* may not be the same between standard and non-standard binary formats; but a set of domain knowledge could serve as the "invariant" (that does not change across binary formats). Such domain knowledge can help to adapt a classifier trained for standard binaries to work on a non-standard binary format.

Inputs and Outputs. As shown in Figure 2, Loadstar is trained on labeled standard binaries and unlabeled non-standard binaries. The set of standard binaries is denoted by S and its label is denoted by L_S. The set of non-standard binaries is denoted by NS and this dataset does not contain ground-truth labels for training. ARM binaries have fixed-length instructions: by performing linear sweep [25], the binary file will be converted to a sequence of "decoded" instructions. The goal of the classifier is to produce labels on each instruction as either "code" or "data". As an example, Figure 3(e) shows the output from our system to separate code and data.

Phase 1: Initialization. As shown in Figure 2, Phase 1 trains an initial classifier C_0 using labeled standard binaries. In step ❶, we first use standard binaries to pre-train an embedding model (E) for the instructions. This pre-trained embedding model will be also used to embed non-standard binaries later. Note that training the embedding model does not require any labels on code and data (all instructions will be used). After that, we train an initial classifier C_0 (supervised learning) using the available labels on standard binaries (❷).

Phase 2: Iterative Training. Phase 2 adapts the initial classifier C_0 for non-standard binaries, using an unlabeled non-standard binary dataset (NS). First, in step ❸, we use classifier C_0 to perform prediction on the non-standard binaries and produce an initial set of pseudo labels (PS_{NS}). Since the classifier is trained on standard binaries, we don't expect the pseudo labels to be highly accurate. As such, we leverage a series of domain-knowledge rules to make corrections to the pseudo labels (❹). This set of domain knowledge can include a wide range of heuristics or rules to describe *expected behaviors of code sections* or *unexpected behaviors of code sections* to curate the initial pseudo labels. Finally, in step ❺, we take the non-standard binary dataset (NS) and the curated pseudo labels (PS_{NS}) to perform a retraining to produce an adaptive classifier C_1. Note that, steps ❸–❺ can be done iteratively to improve the classifier (e.g., in round 2, the new C_1 will be used to produce the pseudo labels, which are then sent to perform label corrections). The process can stop if minimal changes to the pseudo labels are made by the domain-knowledge rules.

4. System Design Details

Figure 2 presents a general framework, and there are different options for implementing Loadstar. For example, there are different ways to perform instruction embedding and construct the classification model, and there are different domain-knowledge rules to use for pseudo-label correction. In this paper, we have explored alternative designs. For instance, our primary design uses an embedding model specifically trained for (standard) binaries and uses an LSTM model [26] to build the classifier (to balance accuracy and efficiency). As an alternative direction, we have explored using a large BERT model [27] to perform both tasks. In the following, we describe our main design and provide our reasoning behind the design choices. The alternative BERT design (less efficient) is detailed in the supplementary materials [28].

4.1. Embedding Model

For step ❶ in Figure 2, we need to train an *embedding model E* to map an instruction into a vector, before perform-

ing classification tasks. One possible direction is to treat binary code as a form of "natural language" and use large language models to perform instruction embedding. For example, numerous language models [29]–[31] perform "word" embedding to capture the contextual/semantic meaning of the word. The idea is that words with similar meanings/contexts will be closer to each other in the latent vector space. However, the drawback is these natural language models do not consider the unique characteristics and semantic structures of binary code [32] and often are less efficient due to their large model size [33].

With these considerations, we train/adapt a lightweight embedding model Palmtree [33] designed for *standard binaries*. This model considers both inter-instructional and intra-instructional relationships. First, it treats each instruction as a "sentence", fragments each instruction into basic tokens (e.g., space separated), and then performs embedding for the tokens. It also customizes the original self-supervision tasks within large language models for binary analysis tasks, to improve the embedding quality. More specifically, (1) the original Masked Language Model (MLM) is adapted to predict the "masked" tokens within an instruction. (2) The Next Sentence Prediction (NSP) is adapted into Context Window Prediction (CWP) which predicts the occurrence of 2 instructions within a window in the *control flow* of the binary. (3) It introduces a new Def-Use Prediction (DUP) task to predict whether two instructions have a define-use relationship. These self-supervision tasks are designed to improve the embedding of tokens and individual instructions (e.g., by learning context relationships with other instructions). A better embedding usually leads to better downstream classifiers.

The original Palmtree model is trained on x86 binaries (and their code sections). For our purpose, we instead train a new embedding model for ARM binaries using standard binaries only. Note that the training of the embedding model E does not require any labels. In addition, we do not use non-standard binaries for training E because the model training requires identifying control flow graphs (CFG) and define-use relationships, which are infeasible for non-standard binaries.

4.2. Classification Model

For step ❷ in Figure 2, we take the embedding vectors to train a classifier C_0, using labeled standard binaries data. We use a sequential model for this classification task to capture the sequential dependency of tokens in an instruction. We use a Long Short Term Memory (LSTM) network [26] which is a type of Recurrent Neural Network (RNN). LSTM overcomes the challenges of gradient vanishing and exploding in traditional RNNs that impede their ability to learn from long sequences. Binary analysis usually deals with long input sequences, which makes LSTM suitable for this task.

We take the raw binaries as inputs and first perform linear sweep disassembly on the binaries using Radare2 [7]. Radare2 treats all bytes as code and decodes them accordingly (i.e., data sections are also decoded as instructions). Intuitively, the disassembled code instructions will carry meaningful semantic structures while the forcefully decoded data sections will likely produce uncommon/incorrect instructions (e.g., `uqsub16vs r2, pc, r2`). The decoded instructions are first sent to the embedding model E to generate the embeddings for their tokens, which are then formulated into a sequence. Note that we did not use any data/control flow graph for formulating the feature vector because this classifier will be eventually used to classify non-standard binaries where such data/control flow graphs cannot be generated with existing tools. In addition, data sections are forcefully decoded into instructions, and generating a dependency graph for data is not meaningful.

Our LSTM contains (1) an Embedding layer initialized to our adapted Palmtree embedding, (2) an LSTM layer used to learn from the sequential dependency of inputs, (3) a Dropout layer for regularization to reduce overfitting, and (4) a Dense layer with Sigmoid activation to produce the prediction probability (between 0 and 1). For training, the input is split into batches of size 128. Training loss is computed using cross-entropy, and the weights are updated using the RMSprop optimizer. The learning rate is set to be 1×10^{-3}. We implement the LSTM using Keras [34] with TensorFlow 2.13 [35].

4.3. Pseudo-Label Generation and Correction

To adapt classifier C_0 (trained on standard binaries) to classify non-standard binaries, the key idea is to generate pseudo labels and curate the labels using domain knowledge. As shown in Figure 2, we first apply C_0 to non-standard binaries to provide the initial pseudo labels (PS_{NS}), and then apply heuristics and rules to make corrections on these labels (steps ❸–❹). In the following, we provide a detailed explanation of the domain knowledge rules we considered.

There are two high-level considerations behind the rule designs. First, we focus on rules that describe the behavior/patterns of *code* instead of those that describe the *data*. The reason is that code is expected to carry meaningful semantics and structures while the data does not have such constraints. Second, for code-related rules, we consider rules that describe *expected* behaviors of code as well as those that describe *uncommon* behaviors of code. However, we don't expect all the rules to be equally effective. For example, rules that are based on statistical properties (e.g., Rules 5, 6, 7) should be weaker than those based on logical assertions (e.g., Rules 1, 2, 3, 4). This will be validated later in Section 5.2.

Below, we describe the high-level intuitions behind these rules. Additional implementation details of the rules can be found in Table 12 in the Appendix.

(1) Short Code/Data Section.[3] This rule (referred to as *Short Sec*) is designed based on the intuition that most compilers usually do not produce extremely short code or data sections[4]. To apply this rule, we scan the predicted pseudo

3. "Code (Data) Section" here refers to the longest uninterrupted sequence of consecutive code (data) bytes based on the prediction results.

4. Extremely short inline data sections are indeed possible, but they are corner cases/uncommon (see Table 3). We set the threshold conservatively (i.e., 3 instructions and less).

labels, identify extremely short sequences of code/data bytes (i.e., 3 instructions or less), and flip their pseudo labels. We perform a round of correction for short data sequences first (and then code) because empirically C_0 produces a higher number of short data sequences or spikes.

(2) Branch Destination. This rule (referred to as *Br Dest*) is based on the expected behavior of code, i.e., the branch (jump) destination of a true code instruction should not land on an invalid address. Here, an invalid address means it does not align with the instruction boundaries. This leads to two sub-rules. First, if a branch instruction's destination lands on an invalid address, this branch instruction should be labeled as "data." Second, if a branch instruction jumps to a valid address, then this instruction should be labeled as "code." These heuristics are easier to apply to fixed-length instructions such as ARM.

(3) Compare-Branch. This rule (referred to as *Cmp-Br*) describes the expected order of "compare" and "branch" instructions. In the code section, conditional branches (i.e., branches that happen based on the result of a condition) should be preceded by a compare instruction. Intuitively, this pattern is less likely to happen in data sections by chance. For conditional branch instructions that follow this pattern, we correct their label as "code." For conditional branches that violate this rule, we change their label to "data."

(4) Define-Use. This rule (referred to as *Def-Use*) describes the expected order between the instruction that defines a variable and the instruction that uses the variable. In other words, the "use" of a variable should be preceded by its "definition". Here, we specifically focus on the sequence of `mov/ldr` instructions followed by `str` instructions. This is a common scenario where a value is loaded/moved to a specific register, changed/operated on, and then stored in another location to be used later. This define-use chain happens often during important stages such as stack preparation for loading new functions. For this rule, we consider direct succession between `mov/ldr` and `str`, and succession within a window of 16 instructions. Additionally, we consider different variations of these instructions (i.e., conditional `mov/ldr` and `str`). Note that this rule is only used to identify "code", i.e., instructions within a correct define-use chain will be labeled as "code." We did not perform the inverse check for this rule, because it's valid to have a standalone `mov` (or `ldr`) that is not followed by a `str` instruction in the code section.

(5) Repeated Addresses. This rule (referred to as *Rep-Addrs*) is based on prior research [36] that suggests that exact addresses occurring many times in a binary are likely to point to shared resources (e.g. library functions). As such, instructions that contain the repeated exact addresses are more likely to be "code". We define a rule to first capture repeated addresses (i.e., addresses that appear at least 2 times in the file) and then find the corresponding instructions to correct their pseudo label as "code." In later experiments, we find this rule can lead to major errors on pseudo labels because data sections also have repeated values (see Section 5.2).

Data	# Files	# Total Instructions	% of Code	Compiler
S	444	20,131,565	18.6%	GCC
NS_1	65	3,707,125	78.5%	Codesys 2.3
NS_2	201	12,425,750	94.2%	Codesys 2.3
NS_3	119	4,271,587	77.8%	Codesys 3.5

TABLE 2: Summary of datasets.

(6) Instruction Suffixes. This rule (referred to as *Inst-Suffix*) is based on the rare occurrence of complex suffixes in code sections. ARM instruction may use different suffixes to perform operations with specific conditions but this is expected to be rare. Some suffixes are conditional (e.g., `moveq` limits the `mov` only if the equal flag is set), and others are concerned with the size of the operands (e.g., `strb` stores the least significant byte). This rule is to correct the corresponding instruction's label to "data" given the rareness of complex suffixes. Our later analysis also finds this rule introduces more errors than corrections.

(7) Operand Type. This rule (referred to as *Operand*) is based on uncommon behavior of code. More specifically, special-purpose registers are not commonly used in code. Examples of special-purpose registers include co-processor registers used with co-processor instructions and floating-point registers used to hold floating-point operands for scalar floating-point instructions. Other operand variations consider the use of shift operations and write-back marks within the operand. Our later analyses also find this rule less effective.

4.4. Re-Training

As shown in step ❺, with the corrected pseudo labels (PS_{NS}) on non-standard binaries, we perform another round of re-training on the classifier to produce an adapted classifier C_1. This model follows the same architecture as the initial model C_0, but is specially tuned for classifying code from data in non-standard binaries.

5. Evaluation

In this section, we seek to evaluate `Loadstar` to answer the following research questions:

RQ1: *Which domain-knowledge-based rules help to curate pseudo-labels?* **(Design Choices)**

RQ2: *How accurate is* `Loadstar` *for data-code separation for non-standard binaries?* **(Effectiveness)**

RQ3: *How computationally efficient is* `Loadstar` *in comparison with LLM-based alternatives?* **(Efficiency)**

RQ4: *How well can* `Loadstar` *generalize to a new dataset/binary format?* **(Transferability)**

5.1. Datasets and Experimental Setup

Data. To the best of our efforts, we collected and labeled three datasets of non-standard binaries and one dataset of standard binaries. The datasets are summarized in Table 2.

1) **S:** This is a dataset of *standard* binaries, drawn from the Clemens [37] dataset. The Clemens dataset is obtained from various software packages (e.g., coreutils) with binaries compiled with different optimization levels (O0–O3). These files are compiled for typical operating systems such as Linux, Windows, and MacOS, following known file formats such as ELF, PE, and Mach-O. For our evaluation, we identify 32-bit ARM ELF binaries from this dataset (444 files and 20 million instructions).
2) **NS_1:** This is our main dataset for non-standard binaries, drawn from the PLCSEC dataset [17]. This dataset includes real-world PLC binaries collected by connecting to Wago-750 PLCs and downloading the control applications/programs. The dataset includes 65 PLC binaries with a total of 3.7 million instructions. These files are in a non-standard format (complied by Codesys 2.3). We chose this dataset because the researchers have spent significant manual efforts on reverse-engineering this file format [18], making it possible for us to label the ground truth.
3) **NS_2:** To evaluate the transferability of our method, we construct an additional dataset of non-standard binaries. This set is drawn from the dataset published by [18]. The authors of the dataset constructed specific search queries using the GitHub advanced search options to locate target public binaries in GitHub repositories (e.g., `.pro` for Codesys 2.3) compiled from different languages, vendors, and architectures. For our evaluation, we use 201 ARM PLC executables (12.4 million instructions).
4) **NS_3:** To diversify the evaluation datasets, we have NS_3, which is a *synthetic* dataset generated by researchers [17]. This dataset is constructed systematically by compiling simple PLC programs composed of basic operations. These programs cover 8 categories of operations: arithmetic, logic, selection, bit shift, comparison, numeric, type conversion, and function calls. Each program contains variable initialization and one instance of the basic operation. The programs are compiled with CodeSys 3.5 (a different compiler from NS_1 and NS_2) for ARM. This dataset includes 119 files (4.2 million instructions).

Binary Annotation and Labeling. First, for standard binaries (S), we can annotate the code and data sections following their standard/known format using Ghidra [5]. We automate the process by loading the binary files using Ghidra's Headless Analyzer, and annotating the code/data sections based on Ghidra's output information of sections, addresses, and permissions.[5] Note that, for standard binaries, we only perform section-level labeling and do not label "inline data" (i.e., data within the code sections). There are two considerations. The first reason is for the ease of labeling (i.e., inline data often requires manual annotation). The second reason is to test `Loadstar`'s ability to learn from noisy labels (i.e., inline data labeled as "code"). Recall that the standard binaries will only be used for embedding and the training of the initial classifier. This dataset helps to test how well `Loadstar` can be trained with standard binaries of coarse-grained (noisy) labels.

Second, for non-standard binaries, we cannot use conventional tools such as Ghidra or IDA Pro for labeling since they do not follow standard formats. Instead, we first rely on a prior work's method [18] to reverse engineer the PLC binaries and identify each section (i.e., function block) and the preceding data section. This allows us to assign coarse-grained labels. In the second stage, we perform manual analyses to identify any missing sections or inline data. The inline data is often introduced for efficient access (e.g., jump tables within the main program), and examples are provided in Appendix C (Figure 7). We label inline data based on heuristics identified after extensive manual analysis of these files. For example, one of such heuristics is to look for a sequence of an unconditional branch instruction (i.e., to introduce a block of data bytes without interfering with the logic of the code), semantic NOP instruction (usually "`mov r0, r0`"), followed by `0xCDCDCDCD` (a pattern used to indicate memory initialization [38]). This sequence is followed by bytes within address ranges that are loaded from other instructions within the binary/section. In Appendix C, we provide examples and extra explanations for these labeling strategies. While these heuristics may not cover all inline data, they cover *all* the inline data we could manually discover without creating any false matches. We label the inline data for non-standard binaries because they will serve as our *testing data*, and thus accurate labels are important to assessing the system's prediction performance.

Training and Testing Data. For each dataset, we randomly split it into a training set and a testing set with a 70:30 ratio, at the *file* level. In other words, 30% of the files are held out for testing. In addition, as stated in Section 2.2, we only use *labeled* standard binaries (S) and *unlabeled* non-standard binaries (NS_1, NS_2, NS_3) for training (i.e., only their training sets). We do not use any ground-truth labels from the non-standard binaries during the training time.

Evaluation Metrics. As shown in Table 2, all the datasets have an imbalanced ratio of "code" and "data" labels. As such, metrics such as *accuracy* can be misleading (e.g., for NS_2, if a system predicts everything as "code", it still obtains an accuracy of 94.2%). Instead, we report Precision (P), Recall (R), and the F1 score (the harmonic mean of precision and recall), for "code" and "data", respectively.

In addition, we are interested in assessing the system performance on inline data. As shown in Figure 3(a), non-standard PLC binaries have a large data section at the end of the file, which can dominate the prediction results. For instance, a predictor that performs poorly on inline data can still appear accurate as long as it predicts this large data section well. To assess the prediction performance on inline data detection, we introduce F1*, which is the F1 score computed on *the remaining part of the file* after excluding the large data section at the end.

5. We discern the size of each section and the associated bytes based on the section addresses and the length information. Additionally, using the permission information, we can label the type of the sections (e.g., code sections are not writable and data sections are not executable).

DK Rules	Against Ground-Truth		Correcting Pseudo Labels			Data				Code			
	Total # of Occurrence	# of Correct Occurrences (%)	Triggers	Corrections	Errors	F1	ΔF1	F1*	ΔF1*	F1	ΔF1	F1*	ΔF1*
Short Sec	NA	NA	237,470	232,153	5,317	0.991	0.105	0.955	0.355	0.997	0.041	0.997	0.041
Br Dest	152,246	131,219 (86%)	67,400	46,608	20,792	0.895	0.009	0.649	0.049	0.961	0.005	0.965	0.009
Cmp-br	87,105	86,339 (99%)	40,386	39,652	734	0.903	0.017	0.643	0.043	0.963	0.007	0.964	0.008
Def-Use	482,349	481,881 (99%)	303,978	301,151	2,827	0.931	0.045	0.697	0.097	0.973	0.017	0.973	0.017
Rep-Addrs	715,611	511,233 (71%)	275,906	71,911	203,995	0.795	-0.091	0.648	0.048	0.935	-0.021	0.967	0.011
Inst-Suffix	1,285,993	731,177 (56%)	353,428	850	352,578	0.758	-0.128	0.376	-0.224	0.882	-0.074	0.882	-0.074
Operand	638,074	418,939 (65%)	219,271	445	218,826	0.802	-0.084	0.438	-0.162	0.961	-0.044	0.911	-0.045

TABLE 3: The effectiveness of different domain-knowledge rules on correcting pseudo labels. Before applying the rules, the initial classifier's pseudo labels have a baseline F1=0.886 (F1*=0.600) for data, and F1=0.956 (F1*=0.956) for code.

Baselines. As we focus on non-standard binaries, we do not have many existing tools to compare with. First, we can not use commercial tools such as IDA Pro and Ghidra as baselines because they cannot process non-standard binaries (see Section 2). Second, most state-of-the-art research tools utilize metadata (e.g., headers) to parse standard binaries as explained in Section 2.3. To this end, we compare our work with tools that take raw binary as inputs, namely XDA [12]. XDA does not perform code and data separation therefore we adapt their instruction boundary detection task for our purpose. The released code of XDA does not include a pre-trained model for ARM and thus we use their code to train an ARM model (see Section 2.3 and Appendix B for details). Finally, we include a BERT implementation of our proposed framework (details in the supplementary materials [28]).

5.2. RQ1: Domain Knowledge (Design Choices)

We start with RQ1 by exploring the usefulness of different domain-knowledge rules for curating pseudo-labels. This analysis uses all the non-standard binary files in NS_1. Most of the rules described in Section 4.3 can be formulated as "*if condition C, then action A*" where condition *C* describes a pattern in the binary file, and action *A* either changes the label from "code" to "data" or vice versa. Given a rule, we measure how often its condition *C* appears in the non-standard binaries, how often its action *A* leads to a correct label, and the rule's overall impact on the label accuracy.

The results are shown in Table 3. We use "Branch Destination" (Br Dest) as an example, to explain the numbers in the table. This rule says "*if a branch (jump) destination lands on a valid address, then this branch instruction should be labeled as code*"; also, "*if a branch destination lands in an invalid address, then this branch instruction should be labeled as data*". First, we directly measure the total occurrences of the rule condition (i.e., jump/branch) among all the instructions (out of 3.7 million in NS_1). We find that the condition has occurred 152,246 times. Among 131,219 of them (86%), this rule leads to correct labels. This means in 14% of these cases, data bytes were decoded as a branch instruction that happened to have a valid destination address.

Recall that in our design (Figure 2), we use the initial classifier (C_0, trained on standard binaries) to generate the pseudo labels for non-standard binaries before applying rules to make corrections. In Table 3, we show the number of actual triggers of the rule (i.e., if a predicted label is already consistent with the "Br Dest" rule, then this rule does not need to trigger a label correction). For "Br Dest", we find that there are fewer triggers (67,400) compared with the total number of condition occurrences (152,246) because the initial classifier C_0 already predicts these labels correctly. Among the triggered ones, 46,608 leads to correct pseudo labels while 20,792 leads to errors. The overall impact is still positive, leading to an increase in F1 score (data) from 0.886 to 0.895. More importantly, for inline data (F1*), the increase is more obvious, from 0.600 to 0.649.

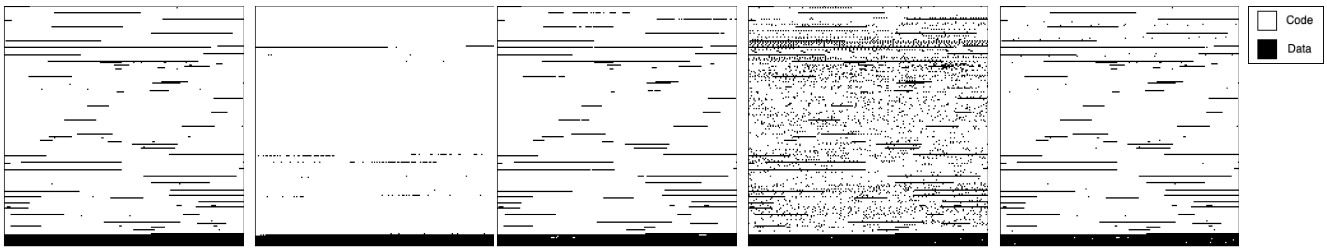
DK Rules	Data				Code			
	F1	ΔF1	F1*	ΔF1*	F1	ΔF1	F1*	ΔF1*
Combo	0.991	0.105	0.966	0.366	0.997	0.041	0.998	0.042
Combo w/o Short Sec	0.934	0.048	0.737	0.137	0.976	0.020	0.977	0.021

TABLE 4: Applying a combination of four domain-knowledge rules (Short Sec, Br Dest, Cmp-Br, Def-Use), in comparison with applying the combination without "Short Sec". Before applying any rules, the initial classifier's pseudo labels have a baseline F1=0.886 (F1*=0.600) for data, and F1=0.956 (F1*=0.956) for code.

Individual Domain-knowledge Rules. Comparing different rules in Table 3, we have four key observations. First, the "Short Code/Data Section" rule is the most effective one to correct pseudo labels. The reason is illustrated in Figure 3(d). The initial classifier trained on standard binaries predicts many short data/code sections. Using the heuristics that most compilers do not produce extremely short sections, we remove these "spikes" in the predicted labels and thus improve the pseudo-label accuracy.

Second, comparing the remaining rules, we find that rules based on logical assertion of code behaviors (e.g., Branch Destination, Compare-Branch, and Define-Use) perform well and positively influence pseudo-label accuracy. In contrast, rules based on statistical properties, especially those that describe statistically "*uncommon*" code behaviors (e.g., Instruction Suffixes, and Operand Type), often produce false labels. This implies that, although certain behaviors are rare for code (e.g., using suffixes), they are not necessarily strong indicators for data either.

Third, the "Repeated Addresses" rule hurts F1 but slightly improves F1* (inline data detection). The rule hypothesizes that repeated addresses are more likely to appear in code sections. However, the mixed result indicates that such "repeated" addresses can also (commonly) appear in the



(a) Ground-truth (b) XDA (c) BERT (d) `Loadstar` (before iterative training) (e) `Loadstar` (after iterative training)

Figure 3: Visualization of a non-standard binary file. The white color represents "code" and the black color represents "data".

Method	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
`Loadstar`	0.990	0.996	**0.993**	**0.975**	0.999	0.997	**0.998**	**0.998**
BERT	0.998	0.961	0.978	0.883	0.986	0.999	0.992	0.993
XDA	0.959	0.737	0.825	0.195	0.950	0.995	0.972	0.974

TABLE 5: Effectiveness of different methods. All classifiers are trained with the training sets of labeled S and unlabeled NS_1. Then they are tested on the testing set of NS_1.

data sections, especially in the large data section at the end of the file, which leads to a reduced F1. We confirm this intuition in Appendix E.

Finally, across all the rules, we observe that the number of triggered label corrections (the "Triggers" column) is much lower than the number of occurrences of the condition patterns (the "Total Occurrence" column). This indicates that the initial classifier (trained with only standard binaries) is already predicting many of the labels correctly.

Combined Domain-knowledge Rules. Based on these results, we then combine the four positive domain-knowledge rules (including "Short Code/Data Section", "Branch Destination", "Compare-Branch", and "Define-Use") and apply them consecutively to the pseudo labels to improve the accuracy. The results are shown in Table 4. Compared with applying individual rules only, the combined rules reached the highest gain in pseudo-label accuracy. As mentioned before, the biggest contributor is the "Short Code/Data Section" rule, without which the gain is less significant. For the rest of the paper, we will use the combination of these four rules for pseudo-label correction.

5.3. RQ2: Effectiveness

Next, we focus on RQ2 to investigate the effectiveness of `Loadstar` and compare it with other baselines. For all the methods, we train them with *labeled* training set of standard binaries (S) and *unlabeled* training set of non-standard binaries (NS_1). After the training, the models are tested on the testing set of NS_1. Due to the space limit, we only report the result for NS_1 for this section. We have also trained `Loadstar` by using NS_2 (and NS_3) to produce pseudo labels for retraining. The result confirms the effectiveness of `Loadstar` on all these datasets (see Table 13 (row "NS_2") and Table 14 (row "NS_3") in the Appendix).

Setting	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
Before	0.731	0.997	0.838	0.614	0.999	0.918	0.957	0.957
After	0.990	0.996	0.993	0.975	0.999	0.997	0.998	0.998

TABLE 6: The effectiveness of `Loadstar` before and after pseudo-label correction and iterative training (NS_1).

Comparing with Existing Methods. The results are presented in Table 5. For `Loadstar` and BERT, we only perform one round of pseudo-label correction and retraining. The result shows that `Loadstar` outperforms XDA and BERT with a higher F1 score and a higher F1*. For example, for data identification, `Loadstar` has an F1 score of 0.993, which is higher than that of XDA (0.825) and BERT (0.978). The advantage of `Loadstar` is further illustrated for inline data detection with an F1* of 0.975, which is much higher than that of XDA (0.195) and BERT (0.883). XDA's performance can be in part attributed to the fact that it is not designed for data-code separation (its original goal is to find instruction boundaries, which is adapted for our purpose). The performance gap between `Loadstar` and BERT is smaller (compared with the gap with XDA), but `Loadstar` is much more efficient than BERT (see later in Section 5.4).

To further visualize the performance differences of these systems, we present a case study in Figure 3. Extra examples are included in the Appendix (see Figure 6). Comparing Figure 3(d) and (e), we show the impact of iterative training (i.e., the use of domain knowledge to curate pseudo-labels). Compared with `Loadstar`, XDA and BERT can also detect most of the code sections and the large data section at the end of the file. However, XDA fails to detect inline data within the code regions. BERT has a good performance but is not as accurate as `Loadstar` on smaller inline data regions (e.g., BERT occasionally breaks a long data sequence into disconnected ones).

Pseudo-label Correction and Re-Training. Next, we systematically evaluate the impact of iterative training. In Table 6, we take the initial classifier (trained on standard binaries (S)) and test it on the testing set of non-standard binaries (NS_1). Then we compare the results with the classifier trained with one round of pseudo-label correction and re-training (using unlabeled NS_1 training set). The result shows that *after* just one round of iterative retraining,

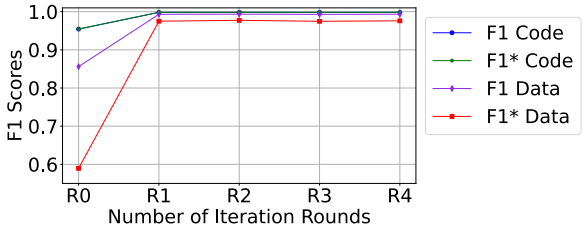


Figure 4: Impact of multiple rounds of iterative training.

Method	Testing	Training			
	Predict (B/s)	Pretrain (hrs)	Training (hrs)	Pseudo-Label Correction (s)	Re-train (hrs)
`Loadstar`	55,539	380	2.4	15.6	0.4
XDA	1,797	110	94.9	-	-
BERT*	80	72	1.7	15.6	1.7

TABLE 7: The efficiency of different methods. For inference/testing, we report bytes per second (B/s) on a CPU. For training, we report the training duration with the CPU. *BERT's training is done entirely with the GPU (training with the CPU suffers from extremely slow speed).

the performance (F1) is increased significantly from 0.838 to 0.993. Essentially, the pseudo-label correction helps with domain adaption to increase the model's performance on the new domain (i.e., non-standard binaries).

Multiple Rounds of Iterative Training. In the above experiment, we only perform one round of pseudo-label correction and re-training. Here, we further explore the benefit of running multiple rounds. As shown in Figure 4, the major performance gain happens from R0 (initial classifier) to R1 (first round of iterative training). After that, the performance gain is minimal. The takeaway is that one round of pseudo-label correction and retraining is sufficient for our dataset. In practice, we can run multiple rounds and stop when the number of label corrections between two consecutive rounds is minimal (see extra results in Appendix F).

5.4. RQ3: Efficiency

To answer RQ3, we evaluate the efficiency of `Loadstar`. Our focus is on the inference efficiency *at the testing time*. The main consideration is that the speed of the tool is important to users when analyzing non-standard binaries. Additionally, we report the model's training overhead, which is a one-time effort. For this evaluation, we use an 8-threaded CPU (Intel Xeon Silver 4214 2.20GHz) and an NVIDIA RTX 5000 GPU.

Inference Performance. The inference speed is important as it directly affects user experience. In Table 7 (the "Testing" column), we compare the inference speed of the different models on our NS_1 testing set. We show that `Loadstar` is the fastest among the three systems with a throughput of 55,539 bytes per second (B/s) on a CPU. XDA has a speed of 1,797 B/s, and BERT is the slowest one with 80 B/s (all measured on the same CPU). In Figure 5, we further plot the cumulative distribution function (CDF) for

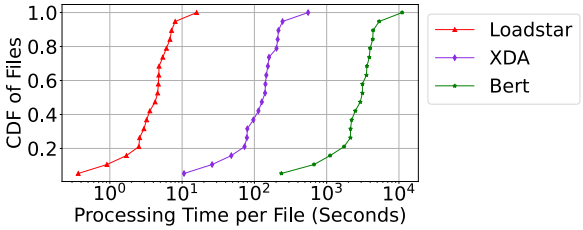


Figure 5: Inference performance. Our method (`Loadstar`) is orders of magnitude faster than XDA and BERT.

the processing time per binary file in the NS_1 testing set. `Loadstar` can process most of the binary files within 10 seconds which is orders of magnitude faster than XDA and BERT. The reason is that `Loadstar` uses the lightweight LSTM instead of using large language models. Our LSTM we use has 93 million parameters and XDA has 87 million parameters; both are smaller models compared with BERT (385 million parameters). More importantly, LSTM has a simpler model architecture than those of BERT and XDA (XDA is transformer-based), which leads to a higher inference speed. This is a critical design decision: using domain knowledge (for pseudo-label correction) combined with a lightweight model, `Loadstar` can achieve similar (or even better) accuracy while being orders magnitude faster than large/complicated models.

Training Performance. In Table 7, we also break down the training performance for different methods. While the training performance is important to measure, most of these steps occur once and do not introduce continuous overhead. The training performance of `Loadstar` and XDA is reported on the CPU while BERT is on the GPU. We use the GPU to train BERT because training BERT on a CPU is extremely slow (infeasible to complete within two weeks). As a reference, we have trained our `Loadstar` on the GPU too, and the pertaining step can be completed within just 3.3 hours (instead of using 380 hours on a CPU). Across different models, the pre-training step (for the instruction embedding) is the most time-consuming part.

A key result to highlight is that `Loadstar` can run the pseudo-label correction and the re-training step quickly (15.6 seconds and 0.4 hours, respectively). This means once the expensive steps are completed (i.e., the pre-training of the embedding model and the training with standard binaries), adapting this model to a specific non-standard binary dataset (unlabeled) will be quick with negligible overhead. For example, if we want to apply `Loadstar` to a new non-standard binary dataset (e.g., NS_3), we only need to execute the lightweight pseudo-label correction and iterative retraining.

5.5. RQ4: Transferability

Finally, we evaluate transferability (RQ4). The goal is to explore feasible strategies to apply `Loadstar` to a new dataset of non-standard binaries.

Test Set	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
S	0.975	0.997	0.986	0.783	0.986	0.920	0.952	0.958
NS_1	0.731	0.997	0.838	0.614	0.999	0.918	0.957	0.957
NS_2	0.428	0.992	0.586	0.412	0.999	0.888	0.940	0.941
NS_3	0.837	0.987	0.906	0.857	0.995	0.932	0.963	0.964

TABLE 8: Baseline: transferring from S to other datasets. We train the classifier using standard binaries only (S), and then test it on standard (S) and non-standard binaries (NS_1, NS_2, NS_3) to establish the baseline transferability.

Test Set	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
S	0.982	0.997	0.990	0.774	0.989	0.927	0.957	0.962
NS_1	0.990	0.996	0.993	0.975	0.999	0.997	0.998	0.998
NS_2	0.974	0.994	0.984	0.974	0.999	0.997	0.998	0.998
NS_3	0.835	0.995	0.908	0.853	0.998	0.930	0.963	0.963

TABLE 9: Transferring from NS_1 to other datasets. We perform iterative training with (NS_1), and then test it on standard (S) and non-standard binaries (NS_1, NS_2, NS_3).

Baseline Performance. We start by establishing a baseline, by training `Loadstar` with only standard binaries (S) and directly testing it on different non-stand binaries. As shown in Table 8, the classifier performs well on the test set of standard binaries (non-transfer setting) and performs worse on NS_1 and NS_2 (transferred setting). The exception is NS_3 on which the classifier performs well. Recall that NS_3 is a *synthetic* dataset generated by researchers [17] which contains simple PLC programs composed of basic operations. Even though it is of a non-standard format, the binaries often have a simple structure with minimal interleaving of code and inline data. As such, it is closer to the format of standard binaries, which may lead to its good performance.

Transferring between Non-standard Binaries. Next, we further investigate the transferability between datasets of non-standard binaries. In this setting, we train `Loadstar` by running pseudo-label correction and retraining on the NS_1 training set (unlabeled) and testing it on all other testing sets. The results are shown in Tables 9. We find that a classifier tuned on NS_1 transfers well to NS_2 (a different dataset of real-world non-standard binaries). The testing accuracy on the original standard binaries (S) also remains high. The transferability to NS_3 is lower due to the reasons mentioned above: NS_3 is a synthetic dataset with different characteristics from real-world non-standard binaries (NS_1, NS_2).

In Appendix D, we repeat the above experiment by using the training set of NS_2 (and NS_3) for iterative training. The overall conclusion is consistent. A classifier tuned on a non-standard binary dataset (unlabeled) performs the best on the same non-standard binary dataset. In practice, when encountering a new dataset of non-standard binaries, the best strategy is to directly tune the classifier with this target dataset using pseudo-label correction and retraining. Note that the adaptation process does not require labeling any of the target binaries, and thus involves minimal human effort. In practice, it is possible that some human effort is needed if additional or different domain-knowledge rules are needed for the new dataset; however, we have not encountered this requirement during our experiment. This process is also efficient with low computational overhead (see Section 5.4).

Test Set	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
S	0.972	0.988	0.980	0.680	0.947	0.883	0.913	0.937
NS_1	0.990	0.996	0.993	0.977	0.999	0.997	0.998	0.998
NS_2	0.972	0.978	0.975	0.960	0.998	0.998	0.998	0.998
NS_3	0.972	0.965	0.968	0.995	0.987	0.990	0.989	0.990

TABLE 10: Iteratively training a global model with a combined training set (NS_1, NS_2, NS_3), then test it on different testing sets. This global model performs well on all testing sets.

Training a "Global" Model. Finally, we explore a different idea, which is to train a global model of all the non-standard binary datasets. More specifically, we first train the initial classifier with standard binaries (S), and then run iterative training on the combined training sets of non-standard binaries (NS_1, NS_2, and NS_3). Again, all of these non-standard binaries are unlabeled. Then this classifier is tested on different testing sets. As shown in Table 10, this global classifier performs well on all testing sets (including NS_3). This shows that, in practice, users can train a global model for different types of non-standard binaries of interest, by tuning with a diverse set of *unlabeled* non-standard binaries in those target formats.

6. Discussion

Domain-knowledge Rules. An integral part of `Loadstar` is the domain-knowledge rules for pseudo-label correction. Intuitively, a classifier trained for standard binaries can make various mistakes when directly applied to unseen formats of non-standard binaries. The domain-knowledge rules can act as the "guard rails" to ensure the pseudo labels are improving through each iteration. Such rules can be derived from the domain knowledge of devices, assembly, compilers, and programming paradigms. In this paper, we investigate a set of rules based on our domain knowledge and prior works, which is by no means exhaustive. We find that rules based on logical assertions (or golden rules, e.g., "a `br` should always point to a valid destination in code") are more reliable than those based on statistical patterns (e.g., "suffixes are not frequently used in code"). However, golden rules may have a lower occurrence in the code and thus make fewer corrections. Our work presents a general framework—future work can explore the possibility of adding other domain-knowledge rules to this module. Another future direction is to explore automated ways to expand or adjust the rules based on feedback from the data or the pseudo-label correction process, e.g., using methods such as reinforcement learning. The challenge is to formulate a reward function that requires minimal to no labels from non-standard binaries.

Downstream Tasks. Data-code separation is the first step for many downstream binary analyses. `Loadstar` can be potentially integrated as a plugin to existing commercial

and research-based tools to act as a file parser/loader. By separating code from data (including inline data), downstream tools can generate accurate control flow graphs or data flow graphs for non-standard binaries to support advanced analyses such as vulnerable function detection.

Impact of the "Initial Classifier." The initial classifier trained on standard binaries is used to produce the preliminary pseudo labels. A potential concern is that, if this initial classifier was too inaccurate, pseudo-label correction may not be sufficient to recover the true labels. In our evaluation, we show that while this initial classifier is not highly accurate, it provides a good starting point for pseudo-label correction. Intuitively, standard and non-standard binaries share key similarities that make domain transfer possible (e.g., instruction structure, decoding rules, registers). As such, it is reasonable to assume the initial classifier has a decent accuracy to start with. In practice, one may use more diverse standard binary samples to train the initial classifier to ensure the accuracy of the preliminary pseudo labels.

Limitations and Future Work. Our work has a few limitations. First, as stated in Section 2.4, we focus on PLC binaries due to their popularity in the IoT/ICS domain and also the feasibility of labeling "ground-truth" datasets. While we attempted to diversify the data (by including different compiler versions and using both real-world and self-complied binaries), the generalizability to other non-standard binary formats needs to be further explored. To broadly study generalizability, future work needs to first overcome the challenges of collecting and labeling new/diverse binary formats [8], [39] (this is also our future plan).

The second direction is to explore how `Loadstar` can work with other ISAs. (1) `Thumb mode`: RISC processors offer mixed-length instructions (16 bits and 32 bits) to improve performance [40]. The 16-bit instructions are called Thumb instructions. ARM executable can be built as either ARM-only (32-bit) or as a mixture of ARM/Thumb instructions (32-bit and 16-bit). Although our datasets do not include ARM/Thumb executables (they are not as common as ARM-only binaries), there is a potential to apply `Loadstar`. Note that Thumb instructions and ARM instructions do not overlap and are located in separate regions in a file (they use special instructions to switch from one mode to the other [41]). A possible direction is to identify patterns (e.g., certain control transfer instructions) that indicate switching/jumping between modes. The two most popular ones are the branch-and-exchange (BX) instruction, and branch-with-link-and-exchange (BLX) instruction [41]. Also, LDR/LDM and POP instructions can cause a mode switch depending on the least significant bit of the program counter. Using these patterns, one can recognize sections generated from mode switching, which helps to minimize breaking 32-bit instructions. Another direction is to combine `Loadstar` (with mode switching detection) and superset disassembly to improve accuracy.

(2) `MIPS`: According to two measurement studies [8], [20], MIPS is the second most common ISA in the IoT/ICS domain (right after ARM). MIPS is also a *fixed-length* instruction set architecture like ARM, which means `Loadstar` is potentially applicable. As stated above, the challenge is to label the "ground-truth" binaries for an extensive evaluation. (3) `x86`: x86 has variable-length instructions and is not commonly seen in the IoT/ICS space [8], [20]. As such, x86 is out of our current scope. To apply the idea to x86, the main point of change should be *input representation* because linear sweep can be error-prone for x86. Instead, we should formulate/label the inputs at the byte level (instead of the instruction level) and construct the model accordingly (e.g., like XDA). That being said, the idea of pseudo-labeling and label correction can be applied similarly to x86.

7. Related Work

Binary Analysis. Researchers have investigated a wide range of binary analysis techniques from decompilation [42] to binary instrumentation [4], [43] as well as complex frameworks [44], [45] that combine static analysis with other techniques such as symbolic execution [3], [46] and fuzzing [47]–[49]. Recently, researchers explored the use of deep learning for different binary analysis tasks. A major area of focus in function similarity analysis [50]–[55], which can be used to search for vulnerable or proprietary functions in binaries. An extension of this line of research is cross-architecture binary code analysis [1], [13], [56]–[59] which helps in finding bugs or similar functions across different architectures. In addition, deep learning has been applied to other binary analysis tasks such as function name prediction, extracting function type signatures, and value-set analysis [14], [15], [60]–[62]. Most of these tasks rely on disassembled code or information extracted from disassembled code. Therefore, disassembly is an important first step.

Disassembly. Traditional disassembly tools fall under three categories: linear sweep [7], [25], recursive traversal [5], [6], [63], [64], and probabilistic disassembly [36]. For linear sweep, the disassembler (e.g., `Radare2`) assumes all bytes are code bytes and decodes the instructions accordingly. This assumes all instructions are sequential and does not consider control or data flow. This method is lightweight but can introduce major false positives [11]. Recursive traversal disassemblers, such as `Ghidra` and `IDA Pro`, start at a given point (usually the program's entry point) and then follow the control flow of the binary. Although these tools considerably reduce false positives, they rely on information not readily available for non-standard binaries. Additionally, this method can miss indirect jumps which leads to incomplete disassembly results. Finally, probabilistic disassembly [36], [65], [66] is based on superset disassembly [67], which returns a *super set* of possible disassembly results by considering all bytes as potential starting points. The disassembler then calculates the probability of each generated output to be a valid output. The probability calculation can be based on either heuristics or machine learning.

Data-Code Separation. Disassemblers need to perform code and data separation (or code discovery) as an initial step for accurate disassembly. For *standard* binaries, de-

tecting "code sections" can be easily done based on their known formats (and information in the header) [5]. Existing research is mainly focused on further detecting *inline data* within the code sections for standard binaries for the x86 architecture [68], [69]. Automating the binary analysis for *non-standard* binaries is still an open problem, and our paper explored the data-code separation problem as an initial step.

IoT Binary/Firmware. Recent research has investigated attacks against IoT devices [9], [70] including attacks that are specifically tailored for PLC devices [24], [39], [71], [72]. To design countermeasures, researchers have studied defense techniques such as attack detection [73], fuzzing [39], [74], patching [75], instrumentation [76] and techniques to ensure control flow integrity [77], [78]. Additionally, there have been attempts to build IoT-specific analysis tools [18], [20], [79]. These tools often require significant reverse engineering efforts to develop.

Semi-Supervised Learning. Semi-supervised learning bridges the gap between supervised and unsupervised learning by utilizing both labeled and unlabeled data for training. Pseudo-labeling [80] is one of the techniques used in semi-supervised learning. It works by generating pseudo labels based on either the labeled data or the model trained on the labeled data (or both). These pseudo labels are then used for retraining the model. Several recent research efforts [81]–[83] have shown competitive results. Most existing methods are applied to image/text classification with a few exceptions that are focused on security applications (e.g., malware detection) [84], [85]. Existing pseudo-label generation is often based on statistical properties. Our method is specifically designed for binary analysis, by introducing pseudo-label correction based on a set of domain knowledge of binary executables and compilers.

8. Conclusion

This paper introduces `Loadstar` to automate data-code separation for non-standard binaries by combining deep learning with domain knowledge. The key idea is to use the abundant labeled data from standard binaries to train a classifier and adapt it to process unlabeled non-standard binaries. Pseudo-label correction is introduced to serve as the guardrail for domain adaptation. We evaluate `Loadstar` using three datasets of non-standard PLC ARM binaries and demonstrate the effectiveness and efficiency of the proposed method. We will share the tool with the community, which can open the door for more accurate and accessible binary analysis on non-standard binaries in the IoT/ICS domain.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grants 2229876, 2055233, and 2326576. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or sponsors.

References

[1] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," *NDSS*, 2016.

[2] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "Ecfi: Asynchronous control flow integrity for programmable logic controllers," *ACSAC*, 2017.

[3] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Krügel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," *NDSS*, 2015.

[4] S. Priyadarshan, H. Nguyen, R. Chouhan, and R. C. Sekar, "Safer: Efficient and error-tolerant binary instrumentation," *USENIX Security*, 2023.

[5] NSA, "Ghidra," *https://ghidra-sre.org/*, 2024.

[6] I. Guilfanov, "Ida pro," *https://hex-rays.com/ida-pro/*, 2024.

[7] S. Alvarez, "Radare2," *https://rada.re/n/*, 2024.

[8] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," *USENIX Security*, 2014.

[9] R. Tsang, D. P. Joseph, Q. Wu, S. Salehi, N. A. Carreon, P. Mohapatra, and H. Homayoun, "Fandemic: Firmware attack construction and deployment on power management integrated circuit and impacts on iot applications," *NDSS*, 2022.

[10] S. Yu, Y. Qu, X. Hu, and H. Yin, "Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," *USENIX Security*, 2022.

[11] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, "D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling," *IEEE SP*, 2023.

[12] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *NDSS*, 2020.

[13] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. X. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," *CCS*, 2017.

[14] J. Patrick-Evans, M. Dannehl, and J. Kinder, "Xfl: Naming functions in binaries with extreme multi-label learning," *IEEE SP*, 2023.

[15] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," *CCS*, 2022.

[16] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," *USENIX Security*, 2015.

[17] H. Benkraouda, A. Agrawal, D. Tychalas, M. Sazos, and M. Maniatakos, "Towards plc-specific binary analysis tools: An investigation of codesys-compiled plc software applications," *CPSIoTSec*, 2023.

[18] A. Keliris and M. Maniatakos, "ICSREF: A framework for automated reverse engineering of industrial control systems binaries," *NDSS*, 2019.

[19] K. A. Stouffer, J. A. Falco, and K. A. Scarfone, "Sp 800-82. guide to industrial control systems (ics) security: Supervisory control and data acquisition (scada) systems, distributed control systems (dcs), and other control system configurations such as programmable logic controllers (plc)," NIST, Gaithersburg, MD, United States, Tech. Rep., 2011.

[20] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, and R. A. Beyah, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," *ISSTA*, 2022.

[21] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. B. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," *CCS*, 2017.

[22] X. Meng and B. P. Miller, "Binary code is not easy," *ISSTA*, 2016.

[23] C. Pang, T. Zhang, R. Yu, B. Mao, and J. Xu, "Ground truth for binary disassembly is not easy," *USENIX Security*, 2022.

[24] E. L'opez-Morales, U. Planta, C. Rubio-Medrano, A. Abbasi, and A. A. Cardenas, "Sok: Security of programmable logic controllers," *USENIX Security*, 2024.

[25] G. Project, "Objdump," *https://web.mit.edu/gnu/doc/html/binutils_5.html*, 1983.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, 1997.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *NAACL*, 2019.

[28] H. Benkraouda, N. Diwan, and G. Wang, "Supplementary materials: Bert implementation details," https://github.com/Benksy/Loadstar, 2024.

[29] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, 2000.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *ICLR*, 2013.

[31] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *NIPS*, 2013.

[32] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," *Usenix Security*, 2024.

[33] X. Li, Q. Yu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," *CCS*, 2021.

[34] F. Chollet, "Keras," *https://keras.io/*, 2015.

[35] Google, "Tensorflow," *https://www.tensorflow.org/*, 2015.

[36] K. A. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," *ICSE*, 2019.

[37] J. Clemens, "Automatic classification of object code using machine learning," *Digital Investigation*, 2015.

[38] E. N. McKay and M. Woodring, *Debugging windows programs: Strategies, tools, and techniques for visual C++ programmers*. Addison-Wesley, 2000.

[39] D. Tychalas, H. Benkraouda, and M. Maniatakos, "Icsfuzz: Manipulating i/os and repurposing binary code to enable instrumented fuzzing in ICS control applications," *USENIX Security*, 2021.

[40] X. Tan, Z. Ma, S. Pinto, L. Guan, N. Zhang, J. Xu, Z. Lin, H. Hu, and Z. Zhao, "SoK: Where's the "up"?! a comprehensive (bottom-up) study on the security of arm Cortex-M systems," *WOOT*, 2024.

[41] J.-Y. Chen, B.-Y. Shen, Q. Ou, W. Yang, and W. Hsu, "Effective code discovery for arm/thumb mixed isa binaries in a static binary translator," *CASES*, 2013.

[42] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," *IEEE SP*, 2016.

[43] L. Bartolomeo, H. Moghaddas, and M. Payer, "Armore: Pushing love back into binaries," *USENIX Security*, 2023.

[44] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," *ICISS*, 2008.

[45] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," *CAV*, 2011.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," *IEEE SP*, 2016.

[47] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Krügel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," *CCS*, 2017.

[48] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," *USENIX Security*, 2013.

[49] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," *NDSS*, 2016.

[50] A. Marcelli, M. Graziano, and M. Mansouri, "How machine learning is solving the binary function similarity problem," *USENIX Security*, 2022.

[51] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *NDSS*, 2019.

[52] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," *AAAI*, 2020.

[53] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," *NDSS*, 2020.

[54] K. Pei, Z. Xuan, J. Yang, S. S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *IEEE Transactions on Software Engineering*, 2023.

[55] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, 2020.

[56] J. Wang, M. Sharp, C. Wu, Q. Zeng, and L. Luo, "Can a deep learning model for one architecture be used for others? retargeted-architecture binary code analysis," *USENIX Security*, 2023.

[57] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search," *NDSS*, 2023.

[58] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," *IEEE SP*, 2015.

[59] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "α diff: Cross-version binary code similarity detection with dnn," *ASE*, 2018.

[60] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," *USENIX Security*, 2017.

[61] W. Guo, D. Mu, X. Xing, M. Du, and D. X. Song, "Deepvsa: Facilitating value-set analysis with deep learning for postmortem program analysis," *USENIX Security*, 2019.

[62] E. C. R. Shin, D. X. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," *USENIX Security*, 2015.

[63] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," *International Conference on Computer Aided Verification*, 2011.

[64] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," *SecDev*, 2017.

[65] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu, "Shingled graph disassembly: Finding the undecideable path," *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2014.

[66] A. Flores-Montoya and E. M. Schulte, "Datalog disassembly," *USENIX Security*, 2019.

[67] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," *NDSS*, 2018.

[68] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," *ECML/PKDD*, 2011.

[69] N. Karampatziakis, "Static analysis of binary executables using structural svms," *NIPS*, 2010.

[70] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," *IEEE SP*, 2016.

[71] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. Zonouz, "Hey, my malware knows physics! attacking plcs with physical model aware rootkit," *NDSS*, 2017.

[72] E. Sarkar, H. Benkraouda, and M. Maniatakos, "I came, i saw, i hacked: Automated generation of process-independent attacks for industrial control systems," *AsiaCCS*, 2020.

[73] X. Tan and Z. Zhao, "Sherloc: Secure and holistic control-flow violation detection on embedded systems," *CCS*, 2023.

[74] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," *NDSS*, 2018.

[75] P. H. N. Rajput, C. Doumanidis, and M. Maniatakos, "Icspatch: Automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs," *Usenix Security*, 2022.

[76] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," *IEEE SP*, 2020.

[77] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "µrai: Securing embedded systems with return address integrity," *NDSS*, 2020.

[78] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," *USENIX Security*, 2013.

[79] H. Wen and Z. Lin, "Egg hunt in tesla infotainment: A first look at reverse engineering of qt binaries," *USENIX Security*, 2023.

[80] D.-H. Lee, "Pseudo-label : The simple and efficient semi-supervised learning method for deep neural networks," *Workshop on challenges in representation learning, ICML*, 2013.

[81] B. Zhang, Y. Wang, W. Hou, H. Wu, J. Wang, M. Okumura, and T. Shinozaki, "Flexmatch: Boosting semi-supervised learning with curriculum pseudo labeling," *NeurIPS*, 2021.

[82] P. Cascante-Bonilla, F. Tan, Y. Qi, and V. Ordonez, "Curriculum labeling: Revisiting pseudo-labeling for semi-supervised learning," *AAAI*, 2020.

[83] E. Arazo, D. Ortego, P. Albert, N. E. O'Connor, and K. McGuinness, "Pseudo-labeling and confirmation bias in deep semi-supervised learning," *IJCNN*, 2019.

[84] Y. Chen, Z. Ding, and D. A. Wagner, "Continuous learning for android malware detection," *USENIX Security*, 2023.

[85] S. Thirumuruganathan, M. Nabeel, E. Choo, I. M. Khalil, and T. Yu, "Siraj: A unified framework for aggregation of malicious entity detectors," *IEEE SP*, 2022.

Appendix A.
List of Standard Binary Formats

Table 11 shows the standard binary file formats that existing binary analysis tools support.

Tools	Supported File Formats
Radare2	ELF, Mach-O, Fatmach-O, PE, PE+, MZ, COFF, XCOFF, OMF, TE, XBE, SEP64, BIOS/UEFI, Dyldcache, DEX, ART, Java class, Android boot image, Plan9 executables, Amiga HUNK, ZIMG, MBN/SBL bootloader, ELF coredump, MDMP, PDP11, XTAC, CGC, WASM, Commodore VICE emulator, QNX, WAD, OFF, TIC-80, GB/GBA, NDS and N3DS, filesystems (NTFS, FAT, HFS+, EXT)
IDA Pro	NE, LX, LE, PE, Windows CE PE, Mach-O, DEX, EXE File, EPOC, DMP, XBE, Intel Hex, NLM, COFF, Raw Binary, OMF file/library, S-record format, ZIP archive, JAR archive, ELF, AIAFF, PEF, Sony Playstation PSX executable files, object (psyq) files, library (psyq) files, W32RUN, AOUT, N64, SMC, MOS Technology Hex Object File, PalmPilot program file, QNX 16 and 32-bits, Motorola DSP56000 .LOD, Sony Playstation PSX executable files, MS DOS (+COM File, Driver)
Ghidra	XML Input Format, Android APK, COFF, DYLD Cache, DEX, DBG, Dump File Loader, ELF, Java Class File, MS COFF, Mach-O, DEF, NE, PE, PEF, MAP, OMF, MZ, Intel Hex, Motorola Hex, CDEX, Raw Binary, GZF Input Format, Ghidra Data Type Archive Format
Binary Ninja	ELF, Mach-O, PE, COFF, NES, Raw Binary

TABLE 11: Supported formats by commercial tools

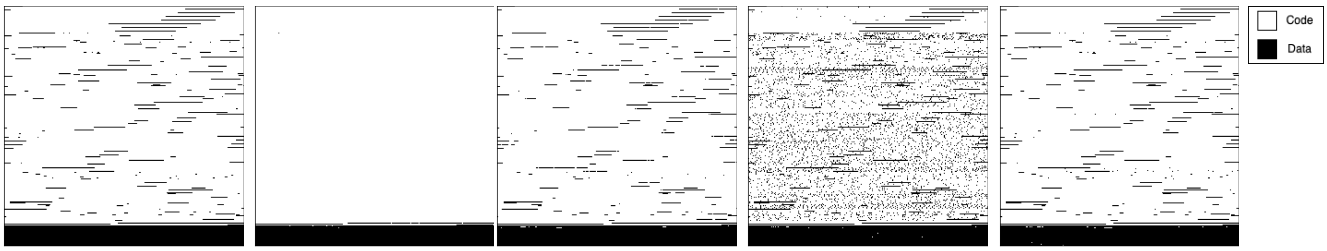
Appendix B.
Adapting XDA for Data-Code Separation

To transfer XDA to perform data-code separation for non-standard PLC ARM binaries, we make the following adaptations. First, we pre-train the model for ARM binaries using a randomly selected set of 103 million instructions from our standard binaries (dataset S). Second, we then fine-tune the model to perform instruction boundary detection. This fine-tuning process is *supervised*, using labeled data (700K instructions randomly sampled from standard ARM binaries). Note that XDA's ground-truth label is different from our system (`Loadstar`). For XDA, the label is applied to each byte to indicate whether the byte represents the "start of an instruction" (denoted by "S"), the "body of an instruction" (denoted by "B"), or others (denoted by "-"). For the data-code separation task, "S" and "B" are regarded as "code", and any "-" predictions are regarded as "data". For our XDA training, both data and code sections are used by re-coding the labels as "S", "B" and "-" labels. We use the same hyperparameters set by the authors in their released code for both pre-training and model fine-tuning. We pre-train the model for 30 epochs and fine-tune the model for another 30 epochs. The evaluation/testing of the XDA model uses non-standard binary datasets. We feed 510-byte sequences to the fine-tuned model. The predicted labels are then mapped back to "code" and "data" to calculate the inference accuracy.

Appendix C.
Labeling Non-Standard Binaries

This section provides extra details on our method of labeling non-standard binaries, in particular, how we label inline data. Labeling the "ground truth" for disassembly tasks is a challenging task [23], which is further exacerbated for non-standard binaries given most existing tools are not applicable. For data-code separation, our labeling method contains two primary phases.

In the first phase, we use ICSREF [18] for coarse-grained labeling. This tool has been developed based on significant reverse engineering efforts to analyze PLC binaries. The tool systematically identifies all subroutines within a PLC binary, including primary functions and helper subroutines. It also detects any data bytes appended to these subroutines. We use this information to label the bytes/instructions within the detected subroutines as "code" and the trailing bytes as



(a) Ground-truth (b) XDA (c) BERT (d) `Loadstar` (before iterative training) (e) `Loadstar` (after iterative training)

Figure 6: An extra example: visualization of a non-standard binary file.

Method		Correction Level	Correction Direction	Implementation Details
Short Sec	**Short Data**	Seq	Data → Code	Find all data sequences that are shorter than threshold (w) and change their label to code.
	Short Code	Seq	Code → Data	Find all code sequences that are shorter than threshold (w) and change their label to data.
Br Dest	**Br → Mid**	Inst	Code → Data	Find all branch instructions, and check whether their destination is divisible by 4. If the destination is *not* divisible by 4 and the prediction is code, change it to data.
	Br → Top	Inst	Data → Code	Find all branch instructions, check whether their destination is divisible by 4. If the destination is divisible by 4 and the prediction is data, change it to code.
Cmp-Br	**Cmp-Br(c)**	Inst	Data → Code	Find all conditional branches, and check whether the preceding instruction is a `cmp`. If the previous instruction is a `cmp` and the prediction of the `br` instruction is data, change it to code.
	NotCmp-Br(c)	Inst	Code → Data	Find all conditional branches, and check whether the preceding instruction is not a `cmp`. If the previous instruction is not a `cmp` and the prediction of the `br` instruction is code, change it to data.
Def-Use	**Def-Use**	Seq	Data → Code	Check if either `mov/ldr` is followed by a `str` instruction within a window of 16, if it exists and the predicted label for `mov/ldr` is data change all the sequence to code.
	Def(c)-Use(c)	Seq	Data → Code	Check if either (cond)`mov/ldr` is followed by a (cond)`str` instruction within a window of 16, if it exists and the predicted label for `mov/ldr` is data change all the sequence to code.
	Def(c)-Use	Seq	Data → Code	Check if either (cond)`mov/ldr` is followed by a `str` instruction within a window of 16, if it exists and the predicted label for `mov/ldr` is data change all the sequence to code.
	Def-Use(c)	Seq	Data → Code	Check if either `mov/ldr` is followed by a (cond)`str` instruction within a window of 16, if it exists and the predicted label for `mov/ldr` is data change all the sequence to code.
Rep-Addrs		Inst	Data → Code	Find all repeated (at least 2 times) addresses (per file), then go through the binary and check each if address exist in the repeated addresses list. If it does and the instruction that the address belongs to is labeled as data switch it to code.
Inst-Suffix		Inst	Code → Data	Check if the instruction has 1 or 2 suffixes, if it does and is labeled as code, change it to data.
Operand		Inst	Code → Data	Check the operand type in the instructions, if any of these operand types: co-processor regs, floating point regs, shift ops, write-back mark are present and the instruction is labeled as code, change it to data.

TABLE 12: Summary of the domain-knowledge-based rules to make corrections to pseudo labels. Corrections are made either on individual instructions (Inst) or a sequence of instructions within the related window (Seq).

"data". In addition, any addresses that are not included within the detected subroutines were classified as "data".

In the second phase, we manually analyze randomly selected binaries from the different non-standard datasets. This intensive process required approximately one person-week. We start by analyzing the output from the coarse-grained labels (based on the outputs of ICSREF). We first locate any instructions decoded as `inavlid` and yet labeled as "code". Second, we locate uncommon/unconventional instructions labeled as "code". We rely on intuition and our domain knowledge when considering instructions as unconventional. For example, the "presence of numerous suffixes" or "branching to atypically large addresses" are signals of unconventional instructions for further analysis. Finally, we undertake a meticulous line-by-line review of the code sections, carefully following the logical flow of instructions and comprehending the functional intent of each code snippet.

Through these two phases, we identify key patterns that help to label inline data. The first pattern is illustrated in Figure 7(a). This pattern starts with an unconditional branch (highlighted in green). This branch allows the introduction of a block of data bytes without interfering with the logic of the code instructions. This is then followed by a NOP instruction (usually "`mov r0, r0`"), followed by `0xCDCDCDCD` (a pattern used to indicate memory initialization). The addresses that follow this "prologue" pattern are addresses used in `ldr` instructions (highlighted in red, yellow, and blue). All of these factors solidify that these bytes are indeed data bytes. The second pattern we find (Figure 7(b)) was a branch and link instruction (highlighted in green) preceded by an instruction that saves the link register (LR) (highlighted in yellow) and succeeded by data bytes. These bytes were decoded and included error messages, month names, and prompts. In the example shown in Figure 7(b), it shows an error message for *illegal function code*. This is then followed by an instruction that restores the LR register (highlighted in yellow). These two patterns are repeated multiple times in all the files we manually analyzed. These patterns were

Address	Byte Value	Decoded Bytes
0x00000f20	60809fe5	ldr r8, [0x00000f88]
…		
0x00000f38	44809fe5	ldr r8, [0x00000f84]
…		
0x00000f4c	2c809fe5	ldr r8, [0x00000f80]
…		
0x00000f70	390000ea	b 0x105c
0x00000f74	0000a0e1	mov r0, r0
0x00000f78	cdcdcdcd	stclgt p13, c12, [sp, 0x334]
0x00000f7c	cdcdcdcd	stclgt p13, c12, [sp, 0x334]
0x00000f80	300f0000	andeq r0, r0, r0, lsr pc
0x00000f84	00200030	andlo r2, r0, r0
0x00000f88	4db40000	andeq fp, r0, sp, asr 8
…		
0x0000105c	04009de4	pop{r0}

(a) Example 1

Address	Byte Value	Decoded Bytes
0x0001e570	0e10a0e1	mov r1, lr
0x0001e574	0c0000eb	bl 0x1e5ac
0x0001e578	496c6c65	ille
0x0001e57c	67616c20	gal
0x0001e580	66756e63	func
0x0001e584	74696f6e	tion
0x0001e588	20636f64	cod
0x0001e58c	652028cd	e (Í
…		
0x0001e5ac	0e00a0e1	mov r0, lr
0x0001e5b0	01e0a0e1	mov lr, r1

(b) Example 2

Figure 7: Labeling inline data for non-standard binaries.

Test Set	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
S	0.982	0.997	0.990	0.774	0.989	0.927	0.957	0.962
NS_1	0.991	0.986	0.989	0.975	0.996	0.997	0.997	0.998
NS_2	0.980	0.962	0.971	0.954	0.996	0.998	0.997	0.997
NS_3	0.842	0.966	0.900	0.856	0.987	0.935	0.960	0.964

TABLE 13: Transferring from NS_2 to other datasets. We perform iterative training with (NS_2) and then test it on standard (S) and non-standard binaries (NS_1, NS_2, NS_3).

then used to automatically correct the other files to rectify erroneously labeled inline data.

Appendix D. Extra Transferability Results

We repeat the same transferability experiments in Section 5.5, by using NS_2 (or NS_3) to generate pseudo labels for classifier retraining. The classifier is then tested on all different testing sets. Table 13 shows the result for NS_2, and Table 14 shows the result for NS_3. The conclusion is consistent with Section 5.5. A classifier tuned with NS_2 training set performs the best on the NS_2's testing set. The same observation can be made for the NS_3 experiment.

Test Set	Data				Code			
	P	R	F1	F1*	P	R	F1	F1*
S	0.982	0.997	0.990	0.774	0.989	0.927	0.957	0.962
NS_1	0.596	0.983	0.742	0.414	0.994	0.827	0.903	0.904
NS_2	0.333	0.976	0.496	0.356	0.997	0.823	0.902	0.903
NS_3	0.977	0.960	0.968	0.955	0.986	0.992	0.989	0.991

TABLE 14: Transferring from NS_3 to other datasets. We perform iterative training with (NS_3) and then test it on standard (S) and non-standard binaries (NS_1, NS_2, NS_3).

Appendix E. Domain Knowledge Variations

As discussed in Section5.2, we suspect the "repeated-address" (Rep-Addrs) rule does not work well because there are too many hits in the data section. Here, we run experiments to (1) examine this intuition, and (2) explore ways to improve this rule.

First, to confirm the intuition, we first remove the large "ground-truth" data section at the end of the binary and apply the Rep-Addrs rule to the remaining area, a similar experiment of Table 3. We find this indeed leads to positive improvements (ΔF1_D=0.024, ΔF1*_D=0.048, ΔF1_C=0.011, ΔF1*_C=0.011)). This is different from the negative ΔF1_D and ΔF1_C in Table 3. This shows our intuition is correct.

Second, however, in practice, we do not have the "ground truth" of the data section in non-standard binaries. Realistically, we may use the predicted "pseudo labels" (instead of ground truth) to locate/remove the largest data section (which can be inaccurate). We then repeat the above experiment, but this leads to decreased F1 and F1* (ΔF1_D=-0.068, ΔF1*_D=-0.087, ΔF1_C=-0.015, ΔF1*_C=-0.015). This indicates this rule may not be usable in practice.

Appendix F. Extra Results for Iterative Training

Figure 4 shows the model achieves high F1/F1* scores on the unseen/withheld testing dataset. In practice, we can determine when to stop the iterative training, by counting the number of label corrections on both the training and testing sets for each iteration round.

First, we count the number of pseudo-label corrections in each round (measured on the training set), all compared to the R0 initial classifier. We show that R1 has the largest corrections: R1=155,420 (6.33%), R2=13,216 (0.54%), R3=12,557 (0.51%), and R4=12,432 (0.51%). This indicates we can stop iterative training after R1.

Second, We also count the changes in the testing results for each round (the testing set), all compared with the R0 initial classifier. We observe the same conclusion: R1=84,514 (3.44%), R2=939 (0.038%), R3=953 (0.038%), and R4=365 (0.015%). After the first round, for our datasets, we observe a plateauing in a performance gain.

Appendix G.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

G.1. Summary

The paper presents a methodology that combines machine learning and pseudo-label correction to automate data-code separation in non-standard binaries, utilizing labeled data from standard binaries.

G.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

G.3. Reasons for Acceptance

1) The paper addresses the data-code separation challenge in non-standard raw binary analysis, which is a critical step in obtaining accurate disassembly and performing further analysis.
2) The paper explores several creative ideas, including using language models for instruction embedding, training a new embedding model for ARM binaries with the lightweight Palmtree model and standard binaries, and adding rules for pseudo-label correction based on code characteristics.

G.4. Noteworthy Concerns

1) The presented system is only able to distinguish the code and data section of fixed-length instruction set architectures.
2) The initial classifier trained on standard binaries might not perform well on non-standard binaries without adequate domain adaptation, leading to lower accuracy in the pseudo labels.
3) The evaluation focuses on PLC binaries under the ARM instruction set. The generalizability to other types of non-standard binaries and instruction sets is not considered.

Appendix H.
Response to the Meta-Review

The authors agree with the meta-review in general, and would like to provide extra context for the noteworthy concern (2). The initial classifier trained on standard binaries is used to produce the preliminary pseudo labels. It is not supposed to be very accurate before pseudo-label correction. Indeed, if this initial classifier is too inaccurate, a potential concern is that pseudo-label correction may not be enough to recover the true labels. We did not run into this issue during our evaluation. Intuitively, standard and non-standard binaries share key similarities, which makes domain transfer possible (e.g., instruction structure, decoding rules, registers). As such, it is reasonable to assume the initial classifier has a decent accuracy to start with. In practice, one may use more diverse standard binary samples to train the initial classifier to ensure the accuracy of the preliminary pseudo labels.