DEEPDI: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly

Sheng Yu[†‡], Yu Qu[†], Xunchao Hu[‡], Heng Yin[†‡]
† University of California Riverside
‡ Deepbits Technology Inc.

Abstract

Disassembly is the cornerstone of many binary analysis tasks. Traditional disassembly approaches (e.g., linear and recursive) are not accurate enough, while more sophisticated approaches (e.g., Probabilistic Disassembly, Datalog Disassembly, and XDA) have high overhead, which hinders them from being widely used in time-critical security practices. In this paper, we propose DEEPDI, a novel approach that achieves both accuracy and efficiency. The key idea of DEEPDI is to use a graph neural network model to capture and propagate instruction relations. Specifically, DEEPDI firstly uses superset disassembly to get a superset of instructions. Then we construct a graph model called Instruction Flow Graph to capture different instruction relations. Then a Relational Graph Convolutional Network is used to propagate instruction embeddings for accurate instruction classification. DEEPDI also provides heuristics to recover function entrypoints. We evaluate DEEPDI on several large-scale datasets containing real-world and obfuscated binaries. We show that DEEPDI is comparable or superior to the state-of-the-art disassemblers in terms of accuracy, and is robust against unseen binaries, compilers, platforms, obfuscated binaries, and adversarial attacks. Its CPU version is two times faster than IDA Pro, and its GPU version is 350 times faster.

1 Introduction

A disassembler takes a binary program as input and produces disassembly code and some higher-level information, such as function boundaries and control flow graphs. Most binary analysis tasks [20, 31, 44, 51] take disassembly code as input to recover syntactic and semantic level information of a given binary program. As a result, disassembly is one of the most critical building blocks for binary analysis problems, such as vulnerability search [23, 57], malware classification [28], and reverse engineering [52].

Disassembly is surprisingly hard, especially for the x86 architecture due to variable-length instructions and interleaved code and data. As a result, a simple linear sweep approach like objdump[1] or Capstone[2], despite high efficiency, suffers from low disassembly correctness on Windows binaries and binaries compiled by the Intel C++ Compiler (where jump tables are placed in the code section), and can be easily confused by obfuscators. There has been a long history of research on improving disassembly accuracy. For instance, the recursive disassembly identifies true instructions by following control transfer targets. It largely eliminates false instructions but may miss true instructions that are not reached by other code blocks, leading to a low true positive rate. Commercial disassemblers like IDA Pro and Binary Ninja employ linear sweep and recursive traversal along with undocumented heuristics to achieve high disassembly accuracy, at price of low runtime efficiency. Our experiments show that IDA Pro can only process approximately 72 KB/s, and Binary Ninja 11 KB/s.

Recently, researchers have explored various novel approaches to further improve the disassembly accuracy, such as probabilistic inference [39, 55], static program analysis [46], logic inference [24], and deep learning [43]. However, the improved accuracy often comes at price of even lower runtime efficiency. For instance, Probabilistic Disassembly [39] can only process about 4 KB/s, Datalog Disassembly [24] 4 – 50 KB/s. Even worse, XDA [43], based on expensive BERT [19] model, when running on CPU, can only process 140 B/s according to our evaluation.

Despite the importance of disassembly, we still do not have a disassembler that is both accurate and fast to support downstream binary analysis tasks. This is especially true when dealing with malware, which is often obfuscated to thwart disassemblers for evasion.

In this paper, we present a novel deep learning-based disassembler called DEEPDI, which can achieve high accuracy and efficiency simultaneously. It can be further accelerated on GPU to gain hundreds of times speedup. In order to achieve high efficiency, DEEPDI takes a very different approach than XDA [43] to leverage deep learning. Instead of feeding raw

[1] https://www.gnu.org/software/binutils/manual/
[2] http://www.capstone-engine.org/

bytes as input to an expensive deep learning model as done in XDA, DEEPDI first decodes all possible instructions and converts them into high-level feature vectors, and then identifies true instructions from all instruction candidates by constructing logical relations (e.g., one instruction followed by another, one instruction overlapped with another, etc.) between these instruction candidates and performing graph inference on them. In particular, we use a Relational Graph Convolutional Network (Relational-GCN) [50], because it can capture different kinds of relations between nodes and it is small and efficient. After supervised training, our model is able to identify true instructions. From these identified true instructions, DEEPDI then recovers function entrypoints from the true instructions using heuristics and a simple classifier.

We have conducted extensive experiments to evaluate DEEPDI with respect to accuracy, efficiency, generalizability and robustness. To evaluate the accuracy, we use four datasets (i.e., BAP corpora [17], LLVM 11 on Windows[3], SPEC CPU2006 [6], and SPEC CPU2017 [7]), and compare with five disassemblers (i.e., IDA Pro [3], Binary Ninja [1], Ghidra [2], Datalog Disassembly [24] and XDA [43]). Experimental results show that DEEPDI is comparable or superior to these disassemblers in terms of accuracy on regular binaries. For efficiency, the single-core CPU version of DEEPDI can achieve a throughput of 146 KB/s, which is two times faster than commercial disassemblers. A CUDA implementation of DEEPDI can further improve the throughput by 170 times on a modest GPU, reaching 24.5 MB/s, which is 350 times faster than IDA Pro. To evaluate its generalizability, we first train our model with BAP corpora on each optimization, and evaluate on LLVM 11 to show the performance on unseen binaries compiled with different compilers and on a different platform. The result shows that our instruction precision and recall are at least 97.1%. We use the model for the accuracy test and test it on ten unseen real-world software to show the performance on real-world binaries, and the result is comparable with XDA. For robustness, we evaluate the performance on obfuscated binaries provided by Linn and Debray [36] and some real-world binaries obfuscated by Hikari [58]. Our model achieves 84.1% precision and 95.2% recall within 1.2 seconds in the first test, whereas XDA and IDA Pro takes over 200 seconds and are less accurate. In the second test, our model has very consistent performance on five different obfuscation techniques, and is several orders of magnitude faster than the other disassemblers.

We further demonstrate how DEEPDI is used in malware classification. We use the malware dataset from Microsoft Malware Classification Challenge [48], and extend Gemini [57] and EMBER [9] to use high-level features for malware classification. Our evaluation shows our Gemini model can achieve 98.2% training accuracy and beat MalConv [47] in testing loss value. The extended EMBER model achieves 99.5% training accuracy and beats the original EMBER. While the traditional feature extractions take hours and even days on this dataset, ours only takes 9 minutes in Gemini and 3 minutes in EMBER, showing the capability of classifying malware accurately and efficiently. We provide a binary release of DEEPDI at https://github.com/DeepBitsTechnology/DeepDi.

[3]https://github.com/llvm/llvm-project

Paper Contributions. In summary, we make the following contributions in this paper:

- We design a novel deep learning-based disassembler that can achieve accuracy and efficiency simultaneously. It exemplifies how a deep learning-based system can substantially improve the efficiency and accuracy over the existing approaches.
- We propose a novel graph representation called "Instruction Flow Graph" to model different relations between instructions. We then use a Relational-GCN to perform inference and classification on Instruction Flow Graph to classify instructions accurately.
- We conduct extensive experiments to show the practical application value of DEEPDI. Experimental results show that DEEPDI is comparable or superior to the state-of-the-art disassemblers in terms of accuracy. DEEPDI is also robust against unseen compilers and platforms, obfuscated binaries, and adversarial attacks. Its efficiency is several orders of magnitude higher than the baseline approaches.
- We showcase malware classification as a downstream application for DEEPDI. We show that DEEPDI can enable fast and accurate malware classification by providing high-level features efficiently.

2 Background

2.1 Traditional Disassembly Methods

Linear Sweep Disassembly. Linear sweep disassembly is the most straightforward yet fast disassembly method. It disassembles from the beginning of the buffer and assumes there is no data in the buffer, meaning the starting point of an instruction is the ending point of the previous instruction. However, this assumption may not hold as compilers may insert jump tables or strings [10], so the false positive rate and false negative rate can be high, especially for obfuscated binaries. Modern compilers do not place strings in the code section, but it happens a lot in shellcode. Besides that, the `Microsoft Visual C++ Compiler` and `Intel C++ Compiler` will place jump tables in the code section, adding errors to linear disassembly results.

Recursive Traversal Disassembly. Recursive traversal disassembly can greatly eliminate false positives. It starts from the entry point of a binary file and follows control flow edges. However, it cannot follow indirect jumps or calls, so it may miss quite a number of code blocks. This method is usually

Table 1: Comparison of Disassembly Approaches

Method	Pros	Cons	Efficiency[1]	
			CPU	GPU
Traditional Approaches	Close to 100% accuracy on regular files	Slow and vulnerable to obfuscation	10 – 200 KB/s	N/A
Superset Disassembly [13]	Very fast and no false negative	85% false positive [39]	4 – 5 MB/s	1+ GB/s
Shingled Graph Disassembly [55]	Similar accuracy to IDA Pro and 2x faster	Small dataset and not open source	70+ – 200 KB/s	N/A
Probabilistic Disassembly [39]	No false negative	3% false positive and slow	4 KB/s	N/A
Datalog Disassembly [24]	Nearly 100% accuracy	Slow and limited file format support	4 – 50 KB/s	N/A
XDA [43]	Close to 100% accuracy	Slow	140 B/s	47 KB/s
DEEPDI (this work)	Close to 100% accuracy	–	146 KB/s	24.5 MB/s

[1] Measured on our server, please refer to Section 4.1 for more details.

combined with some heuristics to detect missing code blocks. Indirect control transfers are very common in complex programs. These programs have switch-case statements, virtual functions, function pointers, etc. Jump tables, such as `jmp dword ptr [addr+reg*4]`, are relatively easy to resolve. However, there exist different variants of jump tables, and some can be difficult to resolve.

These two methods are straightforward and simple, but neither is perfect. IDA Pro has a signature-based approach to scan common patterns of code, others may have dedicated data flow analysis to resolve indirect jumps. Neither is cheap. Code patterns can be affected by compilers, optimization levels, architectures, etc. Therefore, searching in such a large knowledge base is time-consuming. Data flow analysis generally uses an iterative algorithm and requires a lot of computational time. Since the manually-defined heuristics are not complete and slow, we build a machine learning model to automatically capture relations among instructions and use GPU and SIMD instructions in CPU to accelerate the computation.

2.2 Superset Disassembly

Superset Disassembly [13] was proposed for binary rewriting. It disassembles every executable byte offset. Figure 1 (a) and (b) show an example of superset disassembly. Although most of instructions are false positive, all true positives are included in the result so that every possible transfer target can be instrumented during binary rewriting.

2.3 Probabilistic Inference

Shingled Graph Disassembly [55] and Probabilistic Disassembly [39] are both probability-based approaches, and they both start from superset disassembly. Shingled Disassembly maintains an opcode state machine that gives a probability of transition from one opcode to another. It removes execution paths with low probabilities (according to the opcode state machine) to find an optimal execution path with a maximum likelihood. Their algorithm runs in $O(n)$ and according to the paper, their approach is two to three times faster than IDA Pro v6.3. Shingled Disassembly also has a similar accuracy compared to IDA Pro and has fewer missing instructions. Probabilistic Disassembly is a recently proposed binary rewriting approach that uses probabilities to model uncertainties (interleaved code and data, indirect transfer targets, etc.). It considers register define-use relations, control flow convergence, control flow crossing, and computes a probability for each address based on these features. Its experiment shows that it has no false negative, and false positive rate is only 3.7% on average, making it particularly suited for binary rewriting.

2.4 Datalog Disassembly

Datalog Disassembly [24] is also a recently proposed binary rewriting approach. Similar to Probablistic Disassembly, Datalog is based on Superset Disassembly, and it defines a series of rules to remove invalid instructions. For instance, if an instruction falls-through, or jumps, or calls an invalid instruction, this instruction is also invalid. Combined with some heuristics and potential references in data sections, it resolves overlaps and achieves very high accuracy. The downside though, is that such analyses are expensive and can take a lot of time.

2.5 XDA

XDA [43] is a deep learning-based disassembly approach. It takes raw bytes as input, and then randomly masks some of these bytes to learn a language model for instructions. For example, XDA learns `sub rsp` and `add rsp`, a typical function prologue and epilogue, is a pair, which can be used to indicate function boundaries. With this pre-trained language model, one can fine-tune it for various tasks (instruction boundary, function boundary, etc.) with very little training data. XDA also has a good accuracy on unseen real-world projects and is robust to different optimizations. However, it has 12 multi-head attention layers and a large hidden size, or 86,838,795 trainable parameters in total, which make this model very complex and hinder the efficiency benefits brought by GPUs.

2.6 Summary

Each approach has pros and cons. Linear Sweep is the fastest, but the disassembly results may be inaccurate. Recursive has no false positives but can miss a substantial amount of code due to indirect transfers. Superset Disassembly has bloated false positives. Probabilistic Disassembly inherits the advantage of Superset Disassembly, but its runtime performance

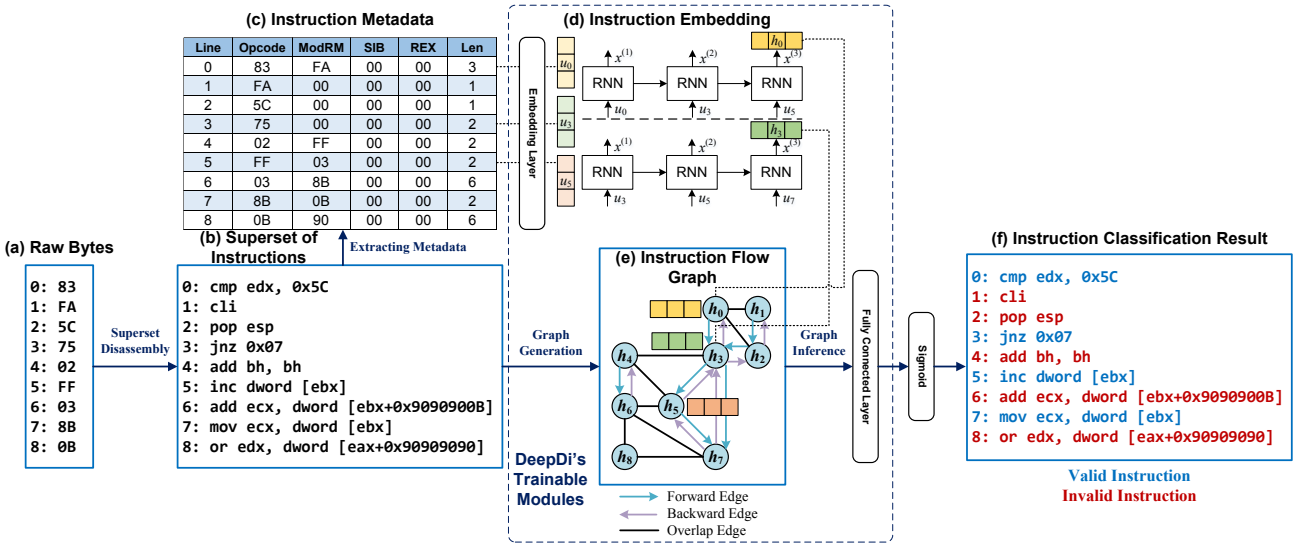


Figure 1: Overview of DEEPDI with a Concrete Example

is much worse than traditional approaches. Shingled Graph Disassembly has an opcode state machine to measure the probability of transition from one opcode to another and removes execution paths with low probabilities. However, it is not open-source, so we cannot evaluate it on a large-scale dataset. Datalog Disassembly adopts a similar idea, thus suffers the same runtime performance issue. XDA uses GPU to accelerate the analysis, but its complex and heavyweight model still hinders its efficiency. These approaches show a trade-off between accuracy and efficiency: a more accurate result requires more sophisticated analysis, resulting in lower efficiency. Table 1 summarizes and compares these existing approaches. Our approach can achieve both high accuracy and high efficiency, so it is applicable to time-sensitive tasks.

3 Design

We envision a good disassembler should achieve the following design goals:

- **High Accuracy.** It should correctly identify instructions and functions with very high recall and precision.
- **High Efficiency.** It should disassemble a binary program at a very high speed, without compromising accuracy.
- **Reasonable Robustness.** While it is impossible to achieve complete robustness against strong adversaries that can be explicitly designed against a disassembler, a good disassembler should be resilient to common obfuscations such as junk code and computed jumps.
- **Support for Downstream Tasks.** In addition to identifying instructions and functions, a good disassembler should provide auxiliary information like call graph, control follow graph, etc., which is useful for downstream analysis tasks.

Figure 1 serves as an overview and a running example of DEEPDI. Our approach first uses superset disassembly to disassemble raw bytes. According to the disassembled instructions, we build an instruction flow graph (IFG) representing all possible execution paths. Each instruction is also converted to a feature vector via instruction embedding while maintaining its semantic meaning. The feature vectors are propagated on the IFG using an R-GCN model to obtain neighboring information, and then are fed into a classification layer to predict whether the corresponding instructions are valid. All the aforementioned layers are connected and are trained in an end-to-end supervised fashion.

Moreover, we further leverage the prediction results to recover function entrypoints (not shown in Figure 1). We treat instructions that are not reachable by non control transfer instructions as function candidates. We then train a classifier to identify true function entrypoints from the candidates.

3.1 Superset Disassembly

We use Superset Disassembly [13] to ensure our input to the model is a superset of true instructions. Given N raw bytes $b_{0,1,\ldots,N-1}$, the output of superset disassembly is as follows:

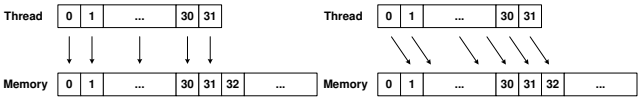
$$t_i = D(b_{i,\ldots,i+14}), \forall i \in \{0,\ldots,N-1\} \quad (1)$$

where $D(\cdot)$ disassembles the given bytes and each t_i is an $(Opcode, ModRM, SIB, REX)$ tuple. We call this tuple instruction metadata. We feed 15 consecutive bytes (as shown in Equation 1) because an instruction is composed of up to 15 bytes. If the rest of the bytes are less than 15, we will pad them with `0x90` (`nop`). A decoded instruction may have

prefixes, Opcode, ModRM, SIB, Displacement, and Immediate [4], but we only use REX prefix, Opcode, ModRM and SIB as its semantic representation, because displacement and immediate contain arbitrary values and do not affect the semantic meaning. More details are introduced in Section A in the Appendix.

Figure 1 (c) shows an example of t_k and how the tuple is represented. Note that although an instruction often takes more than one byte, superset disassembly will still disassemble from its next byte to obtain all possible instructions, which forms a superset of instructions.

Since disassembling any instruction is independent, this process can be easily parallelized on GPU: given n raw bytes, we simply create n GPU threads, and thread i disassembles from b_i [34]. A modern GPU can schedule over one billion threads, so doing so will not cause performance issues.



(a) Time 0 (b) Time 1

Figure 2: GPU Disassembly State at Different Time

Figure 2 illustrates an example of data parallelism on GPU. Assuming the address of the first instruction byte is 0, we assign thread 0 to 31 (a warp) to disassemble instructions starting at memory location 0 to 31. At time 0, all threads consume one byte at location 0 to 31 accordingly at the same time. At time 1, some threads may turn inactive because they encounter 1-byte instructions and remain inactive until all threads in this warp finish disassembling their instructions. Other threads consume the next bytes, which are memory location 1 for thread 0, 2 for thread 1, and so on. The number of threads we create is the same as the number of bytes in the code section, and each thread will output one instruction.

We make each thread in a warp disassemble a consecutive memory location because of GPU global memory coalescing. When threads in a warp access an aligned and consecutive memory location, this is a coalesced access and GPU can fetch up to 32 words in one memory transaction. If the memory accesses were strided (for example, greater than 31 words), each memory transaction would fetch only one word, wasting almost 97% of memory bandwidth.

When GPU is not available, we can perform this task on CPU, which is very straightforward. We just need to go over one byte at a time and disassemble one instruction starting from it. We can exploit multi-threading on CPU by creating multiple threads, each of which sweeps through one chunk of the input binary.

3.2 Instruction Embedding

After we get the superset of instructions, we would like to use the R-GCN model to infer the true instructions. First, we need to decide what kind of representation of each instruction should be fed into the R-GCN model (the representations are used as the node features in the R-GCN model). In this section, we introduce how we construct instruction representations from their metadata t_i, as shown in Figure 1 (d).

The metadata t_i, i.e., the $(Opcode, ModRM, SIB, REX)$ tuple, is integer-encoded, so we first convert it to a fixed-dimensional embedding via a learnable embedding layer, then incorporate the embeddings of an instruction and its *following instructions* into the instruction representation (feature vector) via a recurrent neural network (RNN). Note that Figure 1 (c) shows the original values of Opcode, ModRM, SIB, and REX extracted from instructions. However, their value ranges may overlap (e.g., the range of ModRM and SIB is $\{0,\ldots,255\}$) and it will confuse the embedding layer. So we add a constant value to Opcode, ModRM, SIB, and REX to make their ranges non-overlapping. In total we have 1,025 distinct opcodes, 257 ModRM, 257 SIB, and 17 REX. Each field has a reserved value which is used when the corresponding field is not presented. This makes the overall input size of the embedding layer 1,556. We use an instruction sequence instead of a single instruction because one instruction carries too little information to tell if it is valid. Take Figure 1 (b) for example, instruction 4 alone looks valid. However, if we also consider its following instruction, instruction 6 where `ebx` is used as a base register, the modification of `bh` in instruction 4 becomes suspicious. In this way, the same instruction in different execution paths can have different semantic representations, and the context-aware representations can help improve the classification accuracy. In our experiment, two following instructions can give enough information and will not cause much runtime penalty.

Formally, we define the instruction i's feature vector as follows:

$$x_i^{(n)} = f(x_i^{(n-1)}, u_{i\oplus(n-1)}), n = 1,\ldots,M \tag{2}$$

where f is the vanilla RNN's recurrent function [49], $x_i^{(n)} \in \mathbb{R}^{d_2}$ is the hidden state of the RNN network ($x_i^{(0)}$ is an all-zero vector, which is the initial hidden state of the RNN). M is the sequence length, which is three in this paper. $u_i \in \mathbb{R}^{4\cdot d_1}$ is the embedding of t_i generated by a learnable embedding layer. Each item in the tuple is treated as a word index and the embedding layer convert it to a d_1-dimensional vector. u_i is the concatenation of the embeddings of the four items (Opcode, ModRM, SIB, REX). For an instruction i in the superset of instructions, we define that the operation $i \oplus j$ represents finding j-th non-overlap following instruction of i. Take Figure 1 (d) for example, for instruction 0, $0 \oplus 1 = 3$, $0 \oplus 2 = 5$, etc. If $i \oplus (k+1)$ does not exist (out of bound or instruction $i \oplus k$ being invalid), we define $i \oplus (k+1) = i \oplus k$.

Since we define $M = 3$ in this paper, a simpler unrolled RNN equation of length three can be defined as follows:

$$x_i^{(3)} = f_{unrolled}(u_i, u_{i\oplus 1}, u_{i\oplus 2}) \tag{3}$$

Since only the RNN steps cannot be parallelized, a small sequence length means it would not be particularly more expensive. That is why our approach can still achieve high efficiency even though an RNN is used.

After the RNN module, we can use $x_i^{(M)}$ (in this paper, $M = 3$) as the representation of instruction i and then feed this representation as the node feature into the R-GCN model for graph inference (see Section 3.4 for more details). We chose the vanilla RNN over GRU or LSTM for better efficiency.

3.3 Instruction Flow Graph

Since we are exhaustively disassembling binaries, there exist many false instructions. Even worse, instructions are variable-length, thus the model cannot easily determine where the true instructions are. To help the model better understand the contexts, we propose to model different relations between instructions using a graph called Instruction Flow Graph (IFG), which is used with the Graph Inference phase to propagate information of each instruction to its neighbors and to classify true instructions.

Formally, we define an instruction flow graph as a directed graph $G = (V, E, R)$. For each node $v_i \in V$, there is a feature vector x_i, a semantic representation of the instruction obtained from Section 3.2. Each edge $(v_i, r, v_j) \in E$ is labeled with a relation $r \in R$ denoting the edge type. $R = \{f, b, o\}$ represents three types: **f**orward, **b**ackward, and **o**verlap, respectively. If the label r in (v_i, r, v_j) is a forward relation, it means the next instruction of i can be j, either i falls through to j, i calls j, or i jumps to j. For example, if the instruction i is a conditional jump which may fall through to j or jump to k, there is a forward edge from i to j and a forward edge from i to k. If instruction i is a return instruction or an indirect jump/call, no forward edge from i is created since the transfer target is unknown. A forward edge from i to j is the same as a backward edge from j to i. If r is an overlap relation, it means instruction i and j overlap with each other. That is, the starting point of instruction j is inside instruction i, or vice versa. These different relations can help the model propagate different kinds of information.

Figure 1 (e) shows an example of an Instruction Flow Graph. For instance, Node 3 has two forward relations because Instruction 3 is a conditional jump and thus has two potential targets. Likewise, Node 0 has two overlap relations with Node 1 and 2 because the length of Instruction 0 is three.

3.4 Graph Inference

For our graph inference, we use a Relational-GCN (R-GCN) [50] to propagate information of each instruction to its neighbors. In this network, nodes can have different kinds of relations so that we can pass different messages along different relations. Recall that a valid instruction makes its successors valid, but not vice versa because it can have multiple predecessors, and only one of them or even none of them is valid. R-GCN is capable of modeling this and increases the likelihood of valid instructions while decreases the likelihood of invalid instructions.

As defined in Section 3.3, an instruction flow graph is denoted as (V, E, R). We use the following propagation model to update the hidden state of each node v_i in each layer:

$$h_i^{(l+1)} = ReLU\left(\sum_{r \in R}\sum_{j \in N_i^r} \frac{1}{|N_i^r|} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)}\right) \quad (4)$$

where $h_i^{(l)} \in \mathbb{R}^{d_2}$ is the d_2-dimensional hidden state of the node v_i in the l-th layer. N_i^r denotes the set of neighboring indices of node v_i under relation $r \in R$. $|N_i^r|$ denotes the number of nodes in N_i^r. $W_r^{(l)} \in \mathbb{R}^{d_2 \times d_2}$ is the weight matrix for relation $r \in R$ in the l-th layer. $W_0^{(l)} \in \mathbb{R}^{d_2 \times d_2}$ is the weight matrix for the node itself in layer l (self-connection). Initially, $h_i^{(0)} = x_i$, the feature vector associated with node v_i (see Section 3.2). The final output of R-GCN with L layers is the hidden state of the last layer $h_i^{(L)}$. Figure 3 illustrates the propagation process at layer l.

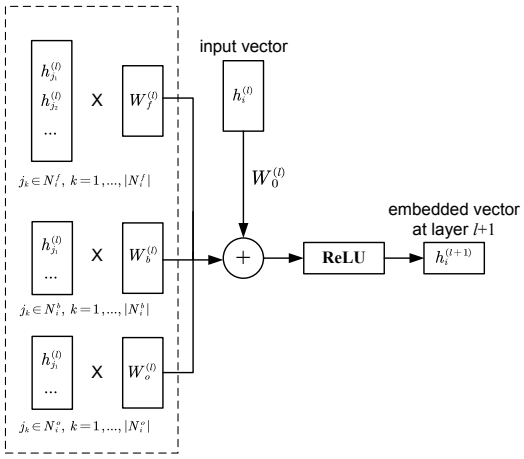


Figure 3: Embedding propagation at layer l of R-GCN

During training, each instruction embedding is propagated and updated L times via different relations: forward, backward, and overlap to capture information from neighboring nodes. The final output $h_i^{(L)}$ is fed into a classifier: a fully-connected layer to reduce the dimension to one, and then activated by *sigmoid* to generate a probability p. We try to minimize the Binary Cross Entropy loss function:

$$J(\Theta, p, y) = \sum(-(y \cdot \log(p) + (1-y) \cdot \log(1-p))) \quad (5)$$

where Θ denotes the model parameters and y is the true label. As shown in Figure 1, all the trainable modules of DEEPDI are linked together and trained in an end-to-end fashion.

3.5 Function Entrypoint Recovery

To recover function entrypoints, we first identify a set of function entrypoint candidates, and then feed each candidate and its surrounding instructions into a classifier. To identify the candidates, we first obtain the metadata of valid instructions, and exclude instructions that are `int3`, `jmp`, `ret`, `nop`, or are reachable via instruction fallthrough or conditional jump because these instructions will not be function entrypoints. We also assume the targets of `call` instructions are function entrypoints. This not only reduces false positives, but also greatly reduces the number of candidates to evaluate.

We then stack each candidate instruction with three preceding instructions and three following instructions into our function entrypoint recovery model. The model has a learnable embedding layer followed a GRU layer and a two-layer perceptron classifier. This will determine if this candidate instruction is indeed a function entrypoint. Let the valid instruction metadata be $\{t_0, t_1, \ldots, t_k\}$, we define the function entrypoint recovery model as follows:

$$g_i = f(u_{i-3}, u_{i-2}, u_{i-1}, u_i, u_{i+1}, u_{i+2}, u_{i+3}) \tag{6}$$

where f is the GRU's recurrent function, and $u_i \in \mathbb{R}^{4 \cdot d_1}$ is the embedding of t_i generated by a learnable embedding layer (not the same embedding layer in Section 3.2). g_i is the hidden state of the GRU layer and is then fed into a classification layer.

During evaluation, we only feed function entrypoint candidates into our model. Since the number of function candidates is very limited compared to the number of superset instructions (about 1:30), this model has almost no impact on runtime performance. Our experiment shows that it helps achieve the average F1 score of function recovery 98.6%.

Guo et al. [27] show that RNN-based function identification tends to learn specific bit patterns, such as `push ebp`. However, we identify function entrypoints based on high-level features learned by the neural network model and accurate instructions, which can likely lead to higher robustness. The drawback of this approach is that we will miss tail jumps and functions with unseen prologues. To identify tail jumps, we can use the same heuristics in other works [45, 46]. If the jump target address is larger than the next function start or smaller than the current function start, it is considered as a tail jump. For unseen prologues, we are able to find many of them via call targets.

4 Evaluation

In this section, we evaluate DEEPDI's performance. Our experiments aim to answer the following Research Questions (RQs).

RQ1 How does it perform on regular binaries?
RQ2 How does it perform on unseen binaries?
RQ3 How does it perform on obfuscated binaries?
RQ4 How resilient is it against adversarial attacks?

4.1 Implementation and Setup

We use PyTorch [41] to implement our model and write a plugin to disassemble raw bytes and return instruction metadata and an IFG as PyTorch Tensors. To disassemble instructions on GPU, we used a header-only library LDasm[4] and modified the code so that it can run on GPU, and its look-up tables are properly cached and shared among GPU threads. The IFG is represented as a set of sparse adjacency matrices, and we used the PyTorch Sparse[5] library to avoid expensive memory coalescing operations. We ran all the experiments on a dedicated server with a Ryzen 3900X CPU @ 3.80 GHz×12, one GTX 2080Ti GPU, 16 GB memory, and 500 GB SSD.

Baseline. We select the following disassemblers for baseline comparison: Binary Ninja 2.2 [1], IDA Pro 7.2 [3], Ghidra 9.1.2 [2], Datalog Disassembly [24], and XDA [43]. IDA Pro, Ghidra, and Binary Ninja are widely used in reverse engineering and binary analysis practices, and their results are considered high-quality. Datalog is a recently proposed binary rewriting approach. XDA is the state-of-the-art machine learning-based approach. This selection covers the state-of-the-art commercial disassembler tools and the most recent research prototypes.

We used the default settings when evaluating IDA Pro and Binary Ninja. For Ghidra, we disabled its decompiler, ASCII string analyzer, x86 exception handling, and constant reference analyzer to boost its efficiency. We finetuned two XDA models, one for instruction and one for function entrypoints, both based on the pre-trained model that XDA provided. We kept the same hyperparameters as in their paper and finetuned each model for five epochs.

Dataset. We conducted experiments on BAP corpora [17], LLVM 11 for Windows[6], SPEC CPU2006 [6], and SPEC CPU2017 [7]. The BAP corpora contain 1,032 x86 and x64 ELF binaries compiled by `GCC` with optimization levels O0 to O3. Though these corpora also come with ELF binaries compiled by `Intel C++ Compiler` (`ICC`) and PE files, these binaries are not used in experiments due to the existence of jump tables in the code section. LLVM 11 is compiled by `Microsoft Visual Studio 2019` with optimization levels Od, O1, O2, Ox for both x86 and x64 architectures. SPEC CPU2006 is compiled by `GCC-4.8.4` and `MSVC 2008` for x86 and x64 architectures and with four optimization levels. SPEC CPU2017 is also compiled on the two ISAs with four optimization levels by using `GCC-7.5` and `MSVC 2019`. To reduce the training time for XDA, we excluded files larger than 5MB.

In total, we have 1,032 ELF files (268 MB) from BAP, 266 PE files (322 MB) from LLVM, 152 PE files (152 MB)

[4] https://github.com/Rprop/LDasm
[5] https://github.com/rusty1s/pytorch_sparse
[6] https://github.com/llvm/llvm-project

and 190 ELF files (79 MB) from SPEC CPU2006, and 270 PE files (287 MB) and 218 ELF files (120 MB) from SPEC CPU2017. Note that we only count code section size.

It is straightforward to extract the ground truth from ELF files, since there is no data in the code section according to Andriesse et al. [10]. We get instruction boundaries by linearly disassembling the code section. We use pyelftools[7] to get function entrypoints come from the symbol table where the symbol type is "STT_FUNC" and the symbol index is not "SHN_UNDEF" (to exclude external functions). To obtain the ground truth for PE files, we modified DIA2Dump, an example that comes with Visual Studio, to dump all functions, data, and label addresses from pdb files. We can only find data addresses but no data lengths in pdb files, so to estimate data ranges, we first find the label where the data belongs, then treat the data address to the end of that label as data. When creating the labels, we set the label to one if the corresponding byte is the starting point of an instruction or a function.

Evaluation Metrics. For the accuracy evaluation, we use *F1* scores to measure the performance because both precision and recall are pretty high for almost all disassemblers. For generalizability and obfuscation evaluation, we use *Precision* (P) and *Recall* (R) to measure the performance.

Deep Learning Model Settings. We use the Adam optimization algorithm [32] and a default learning rate 10^{-3}. As introduced in Section 3.4, we use the Binary Cross-Entropy Loss to calculate the loss. We choose the following hyperparameters through an informal parameter sweep process: $d_1 = 8$, $d_2 = 16$, $L = 2$, $M = 3$, and the batch size is 1,048,576. If a code section is larger than the batch size, we obtain an Instruction Flow Graph for each batch, and edges outside of this graph are dropped. We apply the same strategy to keep the graph small and fit in the GPU memory during the inference. In each batch, the average valid-to-invalid instruction ratio is about 1:1 because compilers tend to insert sufficient padding instructions to align instructions. If we count the paddings as invalid, the ratio becomes 1:4. The graph size is roughly five times the batch size: almost all instructions have only one forward and one backward relation (fallthrough), each of which overlaps with three instructions on average. We also apply a row normalization to make each node in a similar range [50]. As for the function model, the output length of the embedding layer is 8, the hidden size of GRU is 64, and the hidden layer size of the two-layer perceptron is 64, 1, respectively. In total, our model only has 49,889 trainable parameters.

4.2 Accuracy and Efficiency

In this section, we evaluate the accuracy and the efficiency of DEEPDI and other baseline tools. First, we introduce some details and settings of the experiments, then report and discuss experimental results.

[7] https://github.com/eliben/pyelftools

Training and Testing Details. We randomly shuffled the dataset and did a 90-10% split (90% of binaries are used for training, 10% for testing). Both XDA and DEEPDI are trained for five epochs because XDA converges after five epochs according to their paper. We feed code sections (raw bytes) to XDA and binary files to DEEPDI.

4.2.1 Accuracy

To answer RQ 1, we measure F1 scores of DEEPDI and baseline models at instruction and function levels, as shown in Table 2.

When evaluating instruction level results, we treat `nop`, `int3`, `hlt` and `jmp` instructions, and `lea` instructions whose source and destination registers are the same as padding instructions, thus they do not count towards positive or negative instructions. Similarly, for the function entrypoint evaluation, if the first instruction of a function is `jmp`, this function does not count towards positive or negative functions.

Datalog only supports x64 ELF files, so its evaluation on LLVM binaries is not available, and the corresponding cells show "N/As" in Table 2. From the table, we observe that most disassemblers struggle to identify function entrypoints on SPEC datasets. By looking into the datasets, we find that functions from the BAP and the LLVM dataset are mostly aligned, meaning padding instructions can be found between functions. These padding instructions are a strong indicator of function boundaries. However, functions from SPEC datasets are not aligned. To make it worse, many functions end with non-return calls, and frame pointers are often omitted on high optimization levels. With frame pointers omitted, the first instruction of a function is not `push ebp/rbp`, but `xor`, `cmp`, `mov`, etc. These are normal instructions after a `call` instruction, and this explains why many disassemblers struggle to recover function entrypoints. IDA Pro treats many small functions as error handling code, or "`__unwind`". That is why IDA Pro misses many functions in the LLVM dataset. Note that DEEPDI is not the best performer, but is comparable with the other disassemblers. We are unable to evaluate Shingled Graph Disassembly [55] on our dataset because it is not open source. Still, according to their paper, the accuracy of Shingled Disassembly is comparable to IDA Pro, meaning its instruction-level accuracy is similar to DEEPDI.

4.2.2 Efficiency

Figure 4 shows the correlations between code section size and disassembly time for our approach, IDA Pro, Binary Ninja, Ghidra, Datalog, and XDA. The y-axis of this figure is log-scaled. For IDA Pro, Binary Ninja, and Ghidra, we run them in console/headless mode to avoid unnecessary GUI costs. For Datalog Disassembly, we take the numbers reported from the tool directly. When disassemblers are tested on CPU, only one CPU core is used to ensure fairness.

DEEPDI on GPU clearly stands out in this experiment. Its throughput is about 24.5 MB/s, about 170 times faster than

Table 2: Instruction and Function Level Accuracy

Dataset	Opt.	Instruction F1 (%)						Function Entrypoint F1 (%)					
		DEEPDI	XDA	Datalog	IDA Pro	Binary Ninja	Ghidra	DEEPDI	XDA	Datalog	IDA Pro	Binary Ninja	Ghidra
BAP	O0	99.9	99.9	100	99.9	99.9	100	99.9	99.9	100	100	99.9	100
	O1	99.8	99.9	100	99.9	99.8	99.9	99.3	99.5	100	99.9	99.8	99.9
	O2	99.7	99.9	99.9	99.9	99.8	99.9	98.6	99.4	100	99.9	99.8	99.9
	O3	99.7	99.9	100	99.9	99.7	99.9	99.0	99.5	100	99.9	99.7	99.9
LLVM	Od	99.8	99.9	N/A	99.9	99.8	99.9	99.8	99.9	N/A	99.9	97.1	99.9
	O1	99.8	99.9	N/A	99.9	99.7	99.9	99.8	99.9	N/A	99.8	96.8	99.9
	O2	99.8	99.9	N/A	99.9	99.6	99.9	99.8	99.9	N/A	99.8	89.7	99.7
	Ox	99.7	99.9	N/A	99.9	99.7	99.9	99.8	99.9	N/A	99.8	84.9	99.7
SPEC 2006	O0	99.9	99.9	100	99.9	99.6	98.9	98.4	99.9	99.9	88.8	88.7	97.3
	O1	99.7	99.8	100	99.8	99.3	97.2	97.0	99.3	100	86.3	88.7	93.4
	O2	99.9	99.9	100	99.9	99.2	97.6	96.4	99.5	100	85.2	91.5	92.7
	O3	99.9	99.9	100	99.9	98.9	98.0	98.6	99.5	100	93.3	96.0	99.9
	Os/Ox	99.8	99.9	100	99.9	99.4	97.5	95.3	98.3	100	85.6	87.1	91.6
SPEC 2017	O0	99.9	99.9	99.9	99.9	99.7	94.2	99.0	99.8	100	89.4	93.6	86.7
	O1	99.9	99.9	100	99.9	99.5	95.9	99.7	99.8	100	80.8	95.6	76.8
	O2	99.8	99.9	100	99.9	99.4	95.1	99.5	99.9	100	79.4	96.7	75.5
	O3	99.6	99.9	100	99.9	98.9	90.1	98.9	99.4	100	88.4	93.5	85.0
	Os/Ox	99.7	99.8	100	99.9	99.6	96.5	96.3	99.5	100	72.5	92.1	68.7

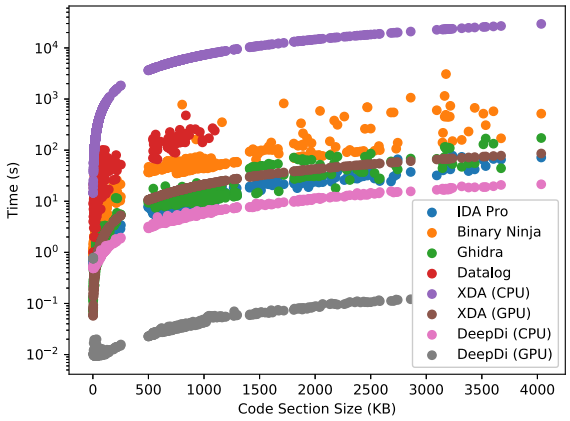


Figure 4: Efficiency Evaluation

DEEPDI on CPU, 146 KB/s. The latter still is noticeably faster than the remaining disassemblers: IDA Pro 72 KB/s, XDA (GPU) 47 KB/s, Binary Ninja 11 KB/s, Ghidra 10 KB/s, Datalog 5 KB/s (for files around 1 MB), and XDA (CPU) 140 B/s. Shingled Graph Disassembly, according to their paper, is two to three times faster than IDA Pro, making it comparable to our CPU approach.

In contrast, XDA is several orders of magnitude slower than the other disassemblers when running on CPU, and its GPU version is merely comparable to the other CPU disassemblers. It is worth noting that we obtained XDA source code from their GitHub repository, but we could not reproduce their reported efficiency. One possible reason is that they used three GPUs [43] whereas we only used one.

The answer to RQ 1: DEEPDI is very accurate on regular binaries. Its accuracy is comparable to all the commercial tools and recent research prototypes. Moreover, DEEPDI is significantly more efficient.

4.3 Generalizability

To answer RQ 2, we conduct two experiments. First, we train our model on the BAP corpora and test it on the LLVM dataset, and then compare it with another machine learning-based model – XDA [43]. We did not do it in the opposite way (i.e., training on the LLVM and testing on the BAP corpora) because XDA is pre-trained on the BAP corpora [43] and this dataset should not be considered unseen for XDA. DEEPDI and XDA are trained on each optimization level of BAP corpora for five epochs and tested on the LLVM binaries. This experiment shows disassemblers' performance on unseen binaries of different compilers (`GCC` vs `MSVC`), platforms (Linux vs Windows), and optimization levels. Second, we evaluate our model and XDA's model from Section 4.2 on the same unseen real-world software used by XDA. This experiment uses unseen real-world software to show the performance in real-world scenarios.

Table 3 lists the evaluation results on instruction and function recoveries. Even though DEEPDI has not seen LLVM binaries before, it still reaches 97.1%+ precision and recall on recovering instruction boundaries. However, XDA only obtains a high precision while recall is constantly below 50%. One possible explanation is that XDA's attention header is too conservative, and does not perform well when instruction patterns are unseen. The function entrypoint recovery evaluation shows a greater degradation when analyzing unseen binaries of unseen compilers. As the optimization level increases, function prologues become less obvious and differ

Table 3: Precision and Recall on Unseen Binaries from an Unseen Compiler

Model	Train \ Test	Instruction								Function							
		Od		O1		O2		Ox		Od		O1		O2		Ox	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DEEPDI	O0	98.6	**99.1**	**98.1**	**97.6**	**98.0**	**97.6**	**98.2**	**97.7**	**94.5**	**42.3**	**95.9**	**38.4**	**74.8**	**26.2**	**73.1**	**26.0**
	O1	98.6	**98.9**	**97.2**	**96.6**	97.9	**97.1**	98.0	**97.1**	**94.9**	**60.5**	**93.3**	**76.8**	**72.2**	**72.1**	**69.5**	**71.9**
	O2	98.9	**99.7**	**98.3**	**98.6**	**98.3**	**98.5**	98.2	**98.6**	**89.4**	**47.3**	**86.7**	**61.6**	**82.6**	**55.0**	**83.1**	**53.7**
	O3	98.2	**99.0**	**97.7**	**96.9**	**98.1**	**97.3**	98.1	**97.4**	**80.4**	**21.0**	**78.7**	**39.5**	**72.9**	**30.9**	**74.3**	**32.5**
XDA	O0	**98.7**	38.9	96.1	43.9	97.1	42.1	97.5	42.6	56.9	0.1	77.6	0.7	5.3	0.03	45.5	0.6
	O1	**99.0**	37.5	**97.2**	44.2	**98.1**	42.5	**98.4**	43.0	2.6	0.4	8.9	1.2	2.3	0.9	3.6	1.4
	O2	**99.1**	38.7	97.2	46.5	98.2	44.2	**98.5**	44.6	16.8	0.5	57.6	3.8	29.5	2.9	34.1	3.9
	O3	**98.9**	39.8	97.3	47.6	**98.1**	44.8	**98.4**	45.1	8.7	0.2	40.4	1.4	7.6	0.4	20.5	1.4

Table 4: Precision and Recall of Function Entrypoint Recovery on Real-world Software

Model	Opt.	curl		diffutils		GMP		ImageMagick		libmicrohttpd		libtomcrypt		OpenSSL		PuTTy		SQLite		zlib	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DEEPDI	O0	99.9	99.9	99.4	99.2	97.7	**97.2**	99.6	99.9	**99.5**	99.5	97.7	94.2	99.7	**100**	99.9	99.8	99.8	99.9	**100**	99.3
	O1	**98.5**	**99.4**	94.6	94.8	**96.9**	85.6	98.2	94.9	**93.6**	89.5	**97.9**	77.1	**97.3**	93.5	**99.4**	91.6	**97.7**	96.9	**98.3**	85.6
	O2	**96.2**	**96.6**	94.4	**96.5**	95.7	90.7	94.1	95.3	91.7	92.7	97.8	95.6	**92.6**	**95.5**	**98.5**	95.6	94.9	**95.5**	**99.1**	84.3
	O3	**96.7**	**97.4**	88.9	**97.9**	96.0	91.3	94.1	95.1	88.7	93.4	**97.9**	95.1	**92.8**	96.0	**99.0**	**96.2**	94.7	**95.9**	98.0	84.2
XDA	O0	**100**	**100**	**100**	**100**	**99.2**	96.7	**99.9**	**100**	**99.5**	**100**	**99.6**	**95.6**	**100**	**100**	**100**	**99.9**	**100**	**100**	**100**	**100**
	O1	91.6	96.0	**96.1**	**96.6**	94.1	**94.2**	**98.9**	**98.7**	89.8	**92.8**	91.4	**95.8**	93.0	**96.1**	95.4	**97.3**	92.9	**97.0**	94.8	**92.7**
	O2	88.9	95.6	**95.9**	95.4	**95.9**	**91.9**	**97.9**	**98.4**	**93.9**	**95.5**	**98.0**	**96.0**	89.6	95.1	96.1	**95.9**	**95.9**	94.0	**99.1**	**90.9**
	O3	88.9	96.1	**94.1**	95.7	**96.3**	**94.7**	**97.1**	**97.8**	**96.6**	**95.8**	97.0	**96.6**	83.8	**97.3**	96.2	94.3	**95.3**	94.9	**98.2**	**93.3**

a lot from compilers to compilers, making function identification much harder. Despite that, DEEPDI outperforms XDA by a large margin.

Table 4 shows the precision and recall of function entrypoint recovery on each software and optimization. We find that DEEPDI is on par with XDA. The F1 scores of both XDA and DEEPDI are close to 100 on instruction recovery, and their performance is almost identical, so we omit the table for instruction recovery.

The first experiment shows that DEEPDI can generalize function entrypoint recovery to some extend when analyzing binaries from unseen compilers and optimization levels. The second experiment shows DEEPDI can generalize pretty well when compilers and optimization levels are already known. This indicates that each compiler has its function patterns on each optimization level, so for DEEPDI, training the model on binaries compiled by gcc and MSVC with different optimization levels is good enough for most general software.

> ***The answer to RQ 2***: For unseen binaries, DEEPDI is still able to achieve high precision and recall. It outperforms another machine learning-based model, XDA, by a large margin for unseen compilers and optimization levels, and is on par with XDA for unseen real-world binaries. These results suggest that DEEPDI has good generalizability.

4.4 Obfuscation Evaluation

To answer RQ 3, we used two different obfuscators to evaluate whether our approach is resilient to obfuscations, and how it compares with the disassemblers with sophisticated heuristics. The first obfuscator was developed by Linn and Debray [36]. In that paper, the authors proposed to insert junk code to confuse both linear and recursive disassembly. Moreover, unconditional jumps are redirected to a universal function that modifies its return address based on callers. This nonstandard behavior hides jump targets and breaks common heuristics. We used the models trained in Section 4.2 and the ground truth provided by Linn and Debray [36]. They provided 11 obfuscated x86 ELF binaries of the SPECint 2000 benchmark suite that have been obfuscated by their tool. Evaluation results of these binaries are shown in Table 5.

We excluded Datalog Disassembly and Binary Ninja because Datalog Disassembly does not support x86 ELF files, and Binary Ninja consumed all memory resources and was killed by the OS. We can observe from Table 5 that DEEPDI is the best performer with respect to precision, recall, and runtime efficiency. In contrast, Ghidra took almost three hours to analyze these binaries and achieved low precision and recall. XDA is slightly worse than DEEPDI in terms of precision and recall, but 235 times slower than DEEPDI on GPU.

Table 5: Obfuscation Test Results

Disassembler	Precision	Recall	Time
DEEPDI (GPU)	**84.1**	**95.2**	**1.2s**
XDA (GPU)	80.2	95.1	282s
IDA Pro	75.8	44.8	262s
Ghidra	69.1	47.0	10,240s

We also evaluated another obfuscator called Hikari [58]. It is an improvement over Obfuscator-LLVM [29], and it can generate hard-to-read code to provide tamper-proofing and increase software security. We used five obfuscation

Table 6: Function Entrypoint Recovery on Obfuscated Unseen Binaries, P: Precision, R: Recall, T: Time

Obfuscation	DEEPDI			XDA			IDA Pro			Binary Ninja			Ghidra			Datalog		
	P	R	T	P	R	T	P	R	T	P	R	T	P	R	T	P	R	T
bcfobf	98.9	98.9	1.6s	99.6	99.4	396s	99.5	100	129s	86.1	100	621s	35.9	33.1	208s	99.7	100	783s
cffobf	99.4	97.9	0.7s	99.6	99.1	342s	99.9	100	112s	98.6	100	593s	39.8	33.0	920s	99.7	100	1,231s
indibran	99.8	98.0	0.5s	99.8	99.0	229s	20.5	100	842s	75.5	99.9	248s	39.7	33.3	230s	98.8	100	905s
splitobf	99.7	98.6	0.6s	99.8	99.3	312s	100	100	117s	98.5	100	539s	42.4	33.2	198s	99.7	100	480s
subobf	99.7	97.9	0.5s	99.8	98.8	187s	100	100	63s	98.6	100	409s	50.6	33.3	105s	99.7	100	284s

strategies, namely bogus control flow (bcf), control flow flattening (cff), basic block splitting (splitobf), instruction substitution (subobf), and register-based indirect branching (indibran) to obfuscate seven popular open-source projects, including `curl-7.74.0`, `diffutils-3.7`, `gmp-6.2.1`, `ImageMagick-7.0.10`, `libmicrohttpd-0.9.72`, `SQLite-3.34.0`, and `zlib-1.2.11`. We also turned off optimizations as instructed by Hikari [58]. The function entrypoint evaluation results are shown in Table 6. In this experiment, IDA Pro has low precision when files are obfuscated by Indirect Branching. It fails to resolve some indirect jump instructions and treats these jump targets as function entrypoints. Ghidra misidentifies many function entrypoints, indicating that signature-based function identification is not very resilient to unseen patterns. IDA Pro, Binary Ninja, Ghidra, and Datalog Disassembly show increased analysis time due to the increased control flow complexity. In contrast, machine learning-based approaches like DEEPDI and XDA are not affected by this.

Based on the results in Table 5 and Table 6, we can see that the two machine learning-based approaches, DEEPDI, and XDA, are superior in accuracy when dealing with obfuscated binaries, but DEEPDI is hundreds of times faster than XDA on GPU.

> ***The answer to RQ 3***: For obfuscated binaries, DEEPDI is superior in accuracy and its efficiency is not affected by the increased code complexity.

4.5 Adversarial Evaluation

An extensive answer to RQ 4 would deserve a separate investigation. In this section, we conduct a preliminary evaluation. Since our model relies on jump relations to recognize true instructions, one possible adversarial attack would be replacing some of these jumps with computed jumps. In this experiment, we trained our model on O3 BAP corpora. In evaluation, we use O0 BAP corpora and randomly drop 50% and 90% of jump edges.

The evaluation results show that if 50% of the jumps are removed, the false positive rate (FPR) increases slightly from 0.0473% to 0.0524%, and the false negative rate (FNR) from 0.24% to 0.51%. If 90% are removed, the FPR is 0.0575%, and the FNR is 0.81%. By analyzing false-negative cases, we find most false negatives are the first instruction of a short basic block, or `nop` instructions at the beginning of a basic block. This makes sense because the first instruction of a basic block, especially a short one, has the least context information if it is not a jump target.

We also evaluate the function entrypoint accuracy. When all jump edges are removed, precision drops to 93.8% and recall to 98%. Precision drops a lot because `GCC` may align basic blocks and insert `nop`s between them. If a function has multiple exits, we can find code patterns like `return` - `nop` - `mov reg, [reg]`. The third instruction looks like a function entrypoint even to humans, and thus confuses the model.

We speculate that the high resiliency of DEEPDI against this jump-obfuscation attack is attributed to graph inference, which takes into account several kinds of relations between instructions. Context information still exists in adjacent instructions and overlapping instructions. Destroying only a part of these relations (in this case, jump relations) does not cause a drastic impact on the overall graph inference task.

> ***The answer to RQ 4***: Through a preliminary evaluation on jump-obfuscation attacks, we show that DEEPDI has good resilience.

5 Downstream Application

In this section, we showcase how DEEPDI can support downstream applications. Particularly, we choose malware classification in this demonstration. We leave more extensive evaluations on downstream applications as future work.

We use the malware dataset from Microsoft Malware Classification Challenge [48]. This dataset contains nine malware families, and is split into 10,868 malware training samples and 10,873 testing samples. Each malware sample comes with IDA Pro disassembly results and raw bytes (represented as hexadecimal values) of the code sections. Some raw bytes are represented as "??", so we removed such bytes and converted other hex strings back to bytes. For all the following experiments, we use 10-fold cross-validation on the training data and report mean accuracy as well as standard deviation. The ground truth of the test dataset is not released to the public, and the only evaluation metric returned from the online judge system is logloss, so we report logloss instead of accu-

racy on the test dataset. As a reference, the logloss of random guessing on the test dataset is *2.19722*.

The top models in this challenge used both disassembly and raw bytes to extract high-level features such as N-gram and strings [48]. These features are expensive and can take hours or even days to extract [8, 59]. Although they could achieve over 99.7% training accuracy and 0.0063 in loss, those models are impractical for real-time analysis.

To demonstrate how the high-level features benefit malware classifiers, we conduct two experiments. First, we compare MalConv [47] with Gemini [57] to compare the performance of classifiers that take raw bytes and high-level features. Second, we compare the original EMBER [9] with a modified version where high-level disassembly features are added.

For the first experiment, we extend Gemini [57] which takes attributed control-flow graph (ACFG) as input, generates embeddings for all basic blocks, and finally outputs an embedding for each function by summing up all basic-block embeddings. To build a malware classifier, instead of generating function embeddings, we concatenate min- and max-pooling of all basic-block embeddings of the program, and then feed them into a 2-layer perceptron followed by a `tanh` activation function. It finally outputs 9-dimensional vectors for classification. We can then use softmax to get a probability for each class. We expect that a classifier based on high-level features can achieve good accuracy and generalizability.

We use Adam optimizer with the default learning rate 10^{-3} and Cross Entropy Loss to train the model. At the input layer, we added a fully connected layer to increase the vector size to 32 to allow more information to pass through ACFGs. We also set the output embedding size 32, and information propagates five hops. In this simple case study, we did not attempt to find the optimal hyperparameters or explore different network architectures, so there is certainly room for improvement.

We also evaluated MalConv [47], a convolutional neural network model that takes raw bytes as input for malware classification. We used the same training strategy described above to train a MalConv model.

Table 7: Malware Classification Results

Model	Training Accuracy	Testing Loss	Time (GPU)
Gemini	96.52% ± 0.595	0.134974 ± 0.036	7m
MalConv	97.81% ± 0.659	0.159165 ± 0.048	48.6s

Table 7 lists the results of this experiment. We can see that although MalConv has better training accuracy, Gemini can better generalize with 0.13 logloss. This result substantiates that a malware classifier based on high-level features tends to be more accurate on unseen samples. In terms of efficiency, MalConv only takes 48.6 seconds to process all testing samples (5.2 GB in total) on GPU, because it takes raw bytes as input. Gemini takes 7 minutes to process the same amount of samples on GPU. This is still a notable achievement, given that DEEPDI has to disassemble the malware samples and extract ACFG as high-level features and then hand them over to Gemini to perform classification.

For the second experiment, we evaluate EMBER which uses static features such as byte code histogram and imported functions to train a gradient-boosted decision tree (GBDT) model. We first train the original EMBER model with the default parameters except changing the objective from binary to multiclass. Later, we add high-level features: code histogram and code entropy histogram to the static features to show how they benefit classification. Code histogram and entropy histogram are extracted from instruction metadata mentioned in Section 3.2, similar to how byte histogram and byte entropy histogram are extracted.

Table 8: EMBER Classification Results

Model	Training Accuracy	Testing Loss	Time
EMBER	99.13% ± 0.1747	0.041541 ± 0.0022	21m
EMBER w/ code	99.40% ± 0.2465	0.024391 ± 0.0018	24m

Table 8 shows that we can lift the training accuracy from 99.1% to 99.4%, and almost halve the testing loss while adding minor overhead (3 minutes).

This case study shows that DEEPDI opens up a lot of opportunities for fast and accurate binary analysis. It will be interesting to explore other machine-learning and deep-learning models that take disassembly results and high-level features as input to produce even more accurate classification results and conduct other binary analysis tasks.

6 Discussion

In this section, we have more discussions about our evaluation results.

Learning-based vs. Rule-based Approaches. In this work, we demonstrate that a learning-based approach outperforms rule-based approaches used in the commercial disassemblers with respect to accuracy (especially on obfuscated binaries) and efficiency. This result might be surprising to many people, as binaries are generated by the compilers following a well-understood compilation process. So experts should be able to develop good rules and heuristics to correctly disassemble the binaries. However, much of higher-level information is lost during the compilation process, and ambiguities start to emerge. The situation is further exacerbated by deliberate obfuscations that aim to break these rules and heuristics, as demonstrated by our obfuscation evaluation in Section 4.4. A learning-based approach, if done right, can automatically learn from a large number of real data on how to resolve the ambiguities and tolerate certain obfuscation attempts. We also demonstrate that a learning-based approach (particularly, a neural network-based approach) can be more efficient than rule-based approaches. A deep neural network model can better leverage the parallelism in modern processors to perform

vector and matrix computation very efficiently. In contrast, a rule-based approach may not be easily parallelized.

Generalizability. A common problem for a machine learning model is overfitting, meaning that the model only learns superficial features existing in the training dataset and cannot generalize on unseen dataset. Our evaluation in Section 4.3 shows that our model is able to learn intrinsic features from the training set, and perform well on a completely different dataset containing a different set of programs generated by a different compiler for a different operating system. We speculate that this excellent generalizability mainly comes from how we make use of Relational-GCN, as it captures a number of important relations between instructions. These relations generally hold true across programs, compilers, and OS.

Adversarial Attacks. A machine-learning system is known to be vulnerable to adversarial attacks. DEEPDI is no exception. However, the disassemblers we evaluated face the same problem, and perform even worse than DEEPDI on obfuscated binaries. Section 4.5 shows that DEEPDI at least is able to counter attacks that simply hide direct jumps. A strong adversary may be able to perform in-depth analysis on our model (e.g., based on the gradients), to construct adversarial examples. This problem deserves a separate investigation, and we leave it as future work. Nevertheless, our evaluation in Section 4.4 and Section 4.5 shows that DEEPDI is already more robust than the existing commercial disassemblers.

7 Related Work

We have discussed existing disassembly techniques in Section 2. In this section, we briefly discuss other related works.

Function Identification. Function identification in stripped binaries is a fundamental challenge in reverse engineering and binary analysis. Nevertheless, many security solutions, such as binary rewriting and control flow integrity, rely on accurate function identification. There exist many machine-learning-based solutions, such as ByteWeight [12] and Shin et al.'s work [53]. ByteWeight extracts features from code (raw bytes or linearly disassembled instructions) and builds a prefix tree to evaluate the probability of a sequence of instructions or raw bytes being function boundary. Shin et al. builds a multi-layer RNN network and feeds one raw byte a time to the network [53]. The output is whether this byte is function boundary or not. Some machine-learning-based models turned out to capture specific patterns, such as `push ebp` [27], as function entrypoint signature, and are likely to miss functions if the first instructions in the function are rarely used (e.g., frame pointer omitted). Others are rule-based solutions such as Nucleus [11] and Qiao et al.'s work [46]. Fundamentally, they rely on various heuristics or program analysis. The problem of function identification is that a precise identification result does not guarantee a precise disassembly because the function body may not be contiguous and may contain data. Another problem is that the runtime performance of function identification is not good.

Differentiating Code and Data. This is another way of thinking disassembly. If we know which part is data, linear sweep disassembly can give us the correct result. Wartel et al. [56] uses a compression model to estimate the probability of a sequence, but its efficiency is not evaluated.

Dynamic Disassembly. Many researchers have made great contributions [14–16, 40, 42, 54] to this direction. Dynamic disassembly can achieve better accuracy on the code path that is actually executed compared to static disassembly, and is resilient to obfuscation and packing, but imposes extra runtime overhead and limited code coverage.

Deep Learning for Binary Analysis. There has been a surge of research efforts on applying deep learning techniques to solve binary analysis problems. A prominent one is binary code similarity analysis and search. Its central theme is to generate an embedding for a piece of code (function or basic block), and then use the generated embedding to search similar code snippets [37, 38, 57, 60]. Researchers also leverage deep learning to perform other sophisticated binary analysis tasks, such as inferring function type signatures [18], and conducting coarse-grained value set analysis [26]. All of these schemes except αDiff [37] require disassembly code or features extracted from disassembly code as input. As a result, no matter how efficient these schemes are, the end-to-end system performance is bounded by the disassembler. By integrating DEEPDI with these downstream tasks, the end-to-end system performance can be improved substantially.

Decompilation. Decompilation takes one step further to recover source code from binaries, and is very useful in understanding or analyzing binaries when their source code is not available. [30] uses an encoder-decoder model to translate raw bytes to pseudo C code, [25, 33] translate instructions to AST. There are also some commercial decompilers such as Hex-Rays Decompiler and Binary Ninja. However, compilers may generate different machine code from the same source code, or the same machine code from different source code. It is hard to evaluate the quality of decompilers.

8 Conclusion

In this paper, we have proposed DEEPDI, a novel deep learning based technique for disassembly that achieves both accuracy and efficiency. Our experimental results have shown that DEEPDI's accuracy is comparable to the state-of-the-art commercial tools and research prototypes, and it is two times faster than IDA Pro, and its GPU version is 350 times faster. DEEPDI is able to generalize to unseen binaries, and counter obfuscations and certain adversarial attacks. When used with EMBER [9] for malware classification involving 5.2 GB testing samples, we are able to increase training accuracy to 99.4% and only add 3 minutes to feature extraction

time, showing its capacity of classifying malware accurately and efficiently.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported in part by National Science Foundation under Grant No. 1719175, and Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Binary ninja, a new type of reversing platform. https://binary.ninja/.

[2] Ghidra – software reverse engineering framework. https://www.nsa.gov/resources/everyone/ghidra/.

[3] The ida disassembler and debugger. https://www.hex-rays.com/products/ida/.

[4] Intel® 64 and ia-32 architectures software developer's manual, volume 2. http://tiny.cc/vskytz.

[5] Researchers easily trick cylance's ai-based antivirus into thinking malware is 'goodware'. http://tiny.cc/qnijuz.

[6] Standard performance evaluation corporation. spec cpu2006 benchmark, 2006.

[7] Standard performance evaluation corporation. spec cpu2017 benchmark, 2017.

[8] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 183–194, 2016.

[9] H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, April 2018.

[10] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[11] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[12] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[13] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[14] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011.

[15] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

[16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[18] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 99–116, 2017.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[20] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[21] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[22] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 2020.

[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[24] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1075–1092, 2020.

[25] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3703–3714, 2019.

[26] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. {DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1787–1804, 2019.

[27] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379, 2018.

[28] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 2009 ACM SIGSAC Conference on Computer and Communications Security*, pages 611–620, 2009.

[29] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[30] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE, 2018.

[31] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[32] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[33] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[34] Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and Georgios Portokalidis. Gpu-disasm: A gpu-based x86 disassembler. In *International Conference on Information Security*, pages 472–489. Springer, 2015.

[35] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118–S126, 2018.

[36] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[37] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678. ACM, 2018.

[38] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[39] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1187–1198, Piscataway, NJ, USA, 2019. IEEE Press.

[40] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

[42] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 81–92. IEEE, 2004.

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

[44] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[45] Rui Qiao and R Sekar. Effective function recovery for cots binaries using interface verification. Technical report, Technical report, Secure Systems Lab, Stony Brook University, 2016.

[46] Rui Qiao and R Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–212. IEEE, 2017.

[47] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[48] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[50] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[51] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In *2011 18th Working Conference on Reverse Engineering*, pages 357–366. IEEE, 2011.

[52] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In *European Symposium on Research in Computer Security*, pages 481–500. Springer, 2008.

[53] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[55] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 273–285. Springer, 2014.

[56] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[57] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[58] Naville Zhang. Hikari – an improvement over obfuscator-llvm. https://github.com/HikariObfuscator/Hikari.

[59] Yunan Zhang, Chenghao Rong, Qingjia Huang, Yang Wu, Zeming Yang, and Jianguo Jiang. Based on multi-features and clustering ensemble method for automatic malware categorization. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 73–82. IEEE, 2017.

[60] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.

A Instruction Representation

In this section, we discuss different design choices of the Pre-processing phase.

Raw Bytes vs. Features. Although some recent studies (e.g., [35]) feed raw bytes to deep learning models and show encouraging results, we argue that raw bytes contain limited semantic information because they are encoded and have variable length. To fully understand the raw bytes, the model has to learn decoding rules, which are already explicitly defined by instruction specifications. Moreover, [5] shows features at raw-byte level are superficial and are vulnerable to adversarial attacks.

Table 9: Instruction Format

Legacy Prefix	REX Prefix	Opcode	ModRM	SIB	Displacement	Immediate
(optional)	(optional)	1-, 2- or 3-byte opcode	1 byte (optional)	1 byte (optional)	1, 2, or 4 bytes (optional)	1, 2, or 4 bytes (optional)

String Representation vs. Metadata The string representation of instructions is very expressive: it has no ambiguity and good readability. Some recent studies [21, 22] are based on the string representation, then utilize NLP models for further analysis. If we see the string representation as *source code*, then the metadata of instructions is similar to *intermediate language*. Table 9 shows instruction format in x86-64 architecture. It essentially shows what each byte in an instruction represents. However, it is still highly encoded, for example, some fields are optional, and some bits in one field can influence the meaning of other fields. Our approach uses metadata because translating from byte code into strings is slow, and relies on our model to learn the meaning of each field.

B Analysis of False Positives and False Negatives

We also dive into the underlying causes of these false results. They are discussed as follows.

First, for false positives, `MSVC` generates jump stubs at the beginning of the code section due to incremental linking. Such patterns do not exist in ELF binaries and it confuses the model when several false instructions look legitimate. Listing 1 shows an example where `00FC100A` and `00FC100F` should be two valid jump instructions, but the model favors instructions starting from `00FC100B`. The `xor` instruction sets the PF flag, and the `jpo` instruction checks the PF flag and does a conditional jump. Both jump targets are legitimate, and it is hard even for humans to decide whether these three instructions are valid or not.

The second outstanding case is the `add esp` instruction. The model favors the opcode `C4`, and all `add esp` instructions become `les` instructions. `les` instructions do not exist in the training set, which might be the reason the model does not perform well.

```
00FC100B  31C2          xor edx,eax
00FC100D  7B 00         jpo 00FC100F
00FC100F  E9 CCC9E400   jmp 01E0D9E0
```

Listing 1: Clang False Positive Example

For false negatives, `MSVC` sometimes generates some very short yet sparse functions. These instructions have very little context information and thus cannot be correctly identified by our model. See Listing 2 for an example.

```
01012F0F    int3
01012F10    mov dowrd ptr ds:[ecx], 021A7014
01012F16    retn
01012F17    int3
01012F18    int3
01012F19    int3
```

Listing 2: Clang False Negative Example