COMPUTER RESEARCH AND MODELING
2024 VOL. 16 NO. 7 P. 1779–1792
DOI: 10.20537/2076-7633-2024-16-7-1779-1792

Ки&М

SPECIAL ISSUE

UDC: 004.93

Fast and accurate x86 disassembly using a graph convolutional network model

N. A. Strygin[a], N. D. Kudasov[b]

Innopolis University,
1 Universitetskaya st., Innopolis, 420500, Russia

E-mail: [a] n.strygin@innopolis.university, [b] n.kudasov@innopolis.ru

Received 26.10.2024, after completion — 15.11.2024
Accepted for publication 25.11.2024

Disassembly of stripped x86 binaries is an important yet non-trivial task. Disassembly is difficult to perform correctly without debug information, especially on x86 architecture, which has variable-sized instructions interleaved with data. Moreover, the presence of indirect jumps in binary code adds another layer of complexity. Indirect jumps impede the ability of recursive traversal, a common disassembly technique, to successfully identify all instructions within the code. Consequently, disassembling such code becomes even more intricate and demanding, further highlighting the challenges faced in this field. Many tools, including commercial ones such as IDA Pro, struggle with accurate x86 disassembly. As such, there has been some interest in developing a better solution using machine learning (ML) techniques. ML can potentially capture underlying compiler-independent patterns inherent for the compiler-generated assembly. Researchers in this area have shown that it is possible for ML approaches to outperform the classical tools. They also can be less time-consuming to develop compared to manual heuristics, shifting most of the burden onto collecting a big representative dataset of executables with debug information. Following this line of work, we propose an improvement of an existing RGCN-based architecture, which builds control and flow graph on superset disassembly. The enhancement comes from augmenting the graph with data flow information. In particular, in the embedding we add Jump Control Flow and Register Dependency edges, inspired by Probabilistic Disassembly. We also create an open-source x86 instruction identification dataset, based on a combination of ByteWeight dataset and a selection open-source Debian packages. Compared to IDA Pro, a state of the art commercial tool, our approach yields better accuracy, while maintaining great performance on our benchmarks. It also fares well against existing machine learning approaches such as DeepDi.

Keywords: disassembly, machine learning, graph neural network, x86

Citation: *Computer Research and Modeling*, 2024, vol. 16, no. 7, pp. 1779–1792.

КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2024 Т. 16 № 7 С. 1779–1792
DOI: 10.20537/2076-7633-2024-16-7-1779-1792

Ки&М

СПЕЦИАЛЬНЫЙ ВЫПУСК

УДК: 004.93

Графовая сверточная нейронная сеть для быстрого и точного дизассемблирования инструкций x86

Н. А. Стрыгин[a], Н. Д. Кудасов[b]

Университет Иннополис,
Россия, 420500, г. Иннополис, ул. Университетская, д. 1

E-mail: [a] n.strygin@innopolis.university, [b] n.kudasov@innopolis.ru

Получено 26.10.2024, после доработки — 15.11.2024
Принято к публикации 25.11.2024

Дизассемблирование двоичных файлов x86 — важная, но нетривиальная задача. Дизассемблирование трудно выполнить корректно без отладочной информации, особенно на архитектуре x86, в которой инструкции переменного размера чередуются с данными. Более того, наличие непрямых переходов в двоичном коде добавляет еще один уровень сложности. Непрямые переходы препятствуют возможности рекурсивного обхода, распространенного метода дизассемблирования, успешно идентифицировать все инструкции в коде. Следовательно, дизассемблирование такого кода становится еще более сложным и требовательным, что еще больше подчеркивает проблемы, с которыми приходится сталкиваться в этой области. Многие инструменты, включая коммерческие, такие как IDA Pro, с трудом справляются с точным дизассемблированием x86. В связи с этим был проявлен определенный интерес к разработке более совершенного решения с использованием методов машинного обучения, которое потенциально может охватывать базовые, независимые от компилятора паттерны, присущие машинному коду, сгенерированному компилятором. Методы машинного обучения могут превосходить по точности классические инструменты. Их разработка также может занимать меньше времени по сравнению с эвристическими методами, реализуемыми вручную, что позволяет переложить основную нагрузку на сбор большого представительного набора данных исполняемых файлов с отладочной информацией. Мы усовершенствовали существующую архитектуру на основе рекуррентных графовых сверточных нейронных сетей, которая строит граф управления и потоков для дизассемблирования надмножеств инструкций. Мы расширили граф информацией о потоках данных: при кодировании входной программы, мы добавляем ребра потока управления и зависимостей от регистров, вдохновленные вероятностным дизассемблированием. Мы создали открытый набор данных для идентификации инструкций x86, основанный на комбинации набора данных ByteWeight и нескольких пакетов Debian с открытым исходным кодом. По сравнению с IDA Pro, современным коммерческим инструментом, наш подход обеспечивает более высокую точность при сохранении высокой производительности в наших тестах. Он также хорошо себя показывает по сравнению с существующими подходами машинного обучения, такими как DeepDi.

Ключевые слова: дизассемблер, машинное обучение, графовые нейронные сети, x86

Introduction

Disassembly is the process of converting machine code into a human- or machine-readable form. It is one of the cornerstones of binary analysis tasks and an active research area.

One of the potential applications of disassembly is binary translation. This intricate process involves converting computer code originally written for a specific *instruction set architecture* (ISA) into code that can execute on a different ISA. This enables, for instance, the execution of legacy x86 applications on an ARM processor without the need for access to the original source code. Other important tasks that often require disassembly include Vulnerability Analysis (finding vulnerabilities in the program) and Malware Analysis (determining if the program is malicious).

Unfortunately, disassembly is not easy to perform correctly without debug information, especially on x86 architecture [Andriesse et al., 2016], which has variable-sized instructions interleaved with data. Moreover, the presence of indirect jumps in binary code adds another layer of complexity. Indirect jumps impede the ability of recursive traversal, a common disassembly technique, to successfully identify all instructions within the code. Consequently, disassembling such code becomes even more intricate and demanding, further highlighting the challenges faced in this field.

Thus, the main challenge for x86 disassembly is accurate identification of x86 instructions. To understand what we mean by "accurate" in this paper, we need to consider the following scenarios:

1. Missing an instruction (making a false negative error) may lead to execution jumping to an untranslated instruction in runtime. Handling this is possible (for example, by including an x86 interpreter as part of the translated binary), but has a non-trivial performance cost for the translated binary.

2. False positives, on the other hand, are less problematic. While they are certainly undesirable, they just add overhead to the translation state. Even though they will never be executed, they still will be translated, wasting translator's CPU time and bloating the translated binary.

Therefore, while it is a trade-off (translator performance and bloat vs runtime performance), it is (arguably) more important to have higher runtime performance than translation performance. As such, we believe it is better to focus on having less false negatives (that is, high recall).

Classical disassembly

Most commercially available tools use recursive traversal accompanied by heuristics to identify instructions and functions. Perhaps, the most prominent tools to date are IDA Pro [IDA Pro Development Team, 1990] and Ghidra [Ghidra Development Team, 2019]. These tools are interactive reverse-engineering tools in which the user is expected to be able to fix the incorrect disassembly manually. These tools have existed for a long time and are widely used by reverse-engineers because of their advanced features, analysis capability, wide architecture support, and an extensive ecosystem with a lot of user-developed plugins.

However, due to their interactive nature, they are not easy to use as a library in another software (e. g. a binary translator). IDA Pro also has a non-free prohibitively expensive license, imposing restrictions on the users of any tool built on top of it.

Additionally, heuristics used in those tools are brittle [Yu et al., 2022; Bao et al., 2014; Pei et al., 2020], as they depend on exact form of the output produced by certain compilers and can be unable to detect certain constructs or output generated by an unknown compiler.

Disassembly with machine learning

If done correctly, Machine Learning (ML) can potentially capture underlying compiler-independent patterns inherent for the compiler-generated assembly. Prior work [Yu et al., 2022; Bao et al., 2014; Pei et al., 2020] has shown that it is possible for ML approaches to outperform these tools. They also can be less time-consuming to develop than manual heuristics, shifting most of the burden onto collecting a big representative dataset of executables with debug information.

However, some of the solutions like XDA [Pei et al., 2020] are not readily available, with DeepDi [Yu et al., 2022], being developed by a commercial company, releasing its model under very restrictive license and not publishing the source code.

Related work

Bauman et al. [Bauman, Lin, Hamlen, 2018] analyzed the instruction identification task and concluded: generally, one cannot identify all instructions in the binary without either missing some of them or marking some of the non-code regions as instructions. With this, they defined a notion of "Superset Disassembly" — a set of instructions that *definitely* includes all true instructions but may include some false ones.

It is an especially useful notion for binary translation, as it cannot tolerate false negative instructions (instructions that were not identified). False positives may be a problem but do not lead to a catastrophic failure, only some performance issues. The authors did not attempt much to remove false positive instructions, basically disassembling all possible addresses. An example of such a disassembly snippet can be seen in Fig. 1.

```
004415d4: 55                   push   ebp
004415d5: 8b ec                mov    ebp, esp
004415d6: ec                   in     al, dx
004415d7: 8b 45 08             mov    eax, DWORD PTR [ebp+0x8]
004415d8: 45                   inc    ebp
004415d9: 08 8b 48 1c 85 c9    or     BYTE PTR [ebx-0x367ae3b8], cl
004415da: 8b 48 1c             mov    ecx, DWORD PTR [eax+0x1c]
004415db: 48                   dec    eax
004415dc: 1c 85                sbb    al, 0x85
004415dd: 85 c9                test   ecx, ecx
004415de: c9                   leave
004415df: 75 10                jne    0x4415f1
004415e0: 10 ff                adc    bh, bh
004415e1: ff 75 10             push   DWORD PTR [ebp+0x10]
004415e2: 75 10                jne    0x4415f4
004415e3: 10 8b 48 18 ff 75    adc    BYTE PTR [ebx+0x75ff1848], cl
004415e4: 8b 48 18             mov    ecx, DWORD PTR [eax+0x18]
004415e5: 48                   dec    eax
004415e6: 18 ff                sbb    bh, bh
004415e7: ff 75 0c             push   DWORD PTR [ebp+0xc]
004415e8: 75 0c                jne    0x4415f6
004415e9: 0c e8                or     al, 0xe8
004415ea: e8 6f 26 fd ff       call   0x413c5e
004415eb: 6f                   outs   dx, DWORD PTR ds:[esi]
004415ec: 26 fd                es std
004415ed: fd                   std
004415ee: ff 5d c3             call   FWORD PTR [ebp-0x3d]
004415ef: 5d                   pop    ebp
004415f0: c3                   ret
004415f1: 80 79 04 05          cmp    BYTE PTR [ecx+0x4], 0x5
```

Figure 1. A machine code snippet that underwent an x86 superset disassembly. Green denotes true instructions, red denotes false instructions

Probabilistic Disassembly [Miller et al., 2019] attempts to reduce the amount of false positive instructions detected. To do that, the authors have built a Naive Bayes Classifier that uses several hand-crafted features to predict the probability of the instruction being true. Then they set quite a low probability threshold of 0.05 and marked all instructions with predicted probability higher than this as true instructions. It is much in the spirit of Superset Disassembly: it predicts all instructions that *might* be true ones.

With this method, they claim to have achieved no False Negatives on their test set and a much lower false positive rate compared to Superset Disassembly [Bauman, Lin, Hamlen, 2018].

Of interest is also the set of features used:

1. Control Flow Convergence: if multiple instructions have the same control flow target, they are both more likely to be true instructions.

2. Control Flow Crossing: if multiple control flow edges cross, which often happens with loops.

3. If instruction i is a jump and its' target does not occlude with the instruction sequence starting with i, it is likely a true instruction.

4. Register Define-Use Relation: if one instruction first sets some register (or flag) and later another instruction uses this register, they are both likely to be true instructions.

The choice of these features is supported by calculations showing that the probability of instructions having them just by chance is relatively low.

DeepDi [Yu et al., 2022] approaches disassembly similarly to Probabilistic Disassembly [Miller et al., 2019]. However, in contrast to Probabilistic Disassembly, DeepDi also aims to be fast. To achieve this, the authors design a neural network which, for each instruction in the superset disassembly, predicts whether this instruction is true or false.

The input to the neural network is metadata that is extracted from instructions after disassembly. Inputs are fed through a fully-connected layer to reduce dimensionality and then through a 3-layer RNN [Hopfield, 1982] with neighboring instructions to get local instruction embeddings. The authors specifically note that using decoded metadata is superior to using raw bytes, as the latter merely shifts the decoding work onto the network.

Then the instructions are passed through an R-GCN [Schlichtkrull et al., 2017] to obtain neighborhood information. The graph that is fed into R-GCN has instructions as nodes and contains the following types of edges:

1. Forward Edge: connects instruction to the instruction which will be executed after the current one.

2. Backward Edge: connects instruction to the instruction that would have been executed before the current one.

3. Overlap Edge: connects two instructions if their machine codes overlap.

Finally, the outputs of R-GCN are fed into a classification layer that predicts whether the instruction is true.

Comparing DeepDi with the model used in Probabilistic Assembly [Miller et al., 2019], one can see that some features used in the latter (mostly related to data flow) cannot be constructed using the information fed into the neural network alone. The authors probably did not feed this information into the model, as it is computationally expensive to calculate, and DeepDi is explicitly focused on being *fast*. Changing the representation to include this information might improve model accuracy.

Contribution

Out contribution consists of `identify-x86`, an open-source ML model for instruction identification, as well an x86 instruction identification dataset. As such, this work will focus on building instruction identification dataset from open sources, as well as `identify-x86`: an open-source Machine Learning model for instruction identification trained on this dataset.

Our implementation is open-source and is available at https://github.com/DCNick3/identify-x86.

The rest of the paper is structured as follows:

- In Methodology, we define the model architecture and describe the methodology for its evaluation.
- In Data Collection, we describe our dataset collection procedure.
- In Evaluation, we provide and interpret benchmark results.

Methodology

As the superset disassembly contains all possible instructions to be found in the binary, instruction identification is now just a matter of assigning a true/false label to each instruction (true meaning the instruction was generated by the compiler and false meaning it was not). As no model is perfect, some of its labels will be mis-assigned. Two kinds of these errors are possible:

1. **False positives** (corresponding to lower precision) — the model classified a false instruction as a true one, meaning an instruction not generated by the compiler was identified. While this is not desirable, it still allows binary translation to work. It just reduces performance of the translator, as resources will have to be spent translating an instruction that will never be executed.
2. **False negatives** (corresponding to lower recall) — the model classified a true instruction as a false one, meaning a compiler-generated instruction was **not** identified. This kind of error is much worse for binary translation, because such an instruction *could* be attempted to be executed by the program and the runtime will then have to either crash or attempt to interpret the instruction. This will have considerable runtime overhead, especially if the undetected instructions are hot.

It is much more important to have fewer false negative errors (that is, high recall) rather than fewer false positives. Ideally, the model should not produce false negative errors at all.

This may be accomplished by using a suitably crafted loss function for the model. We are using weighted binary cross-entry loss with false-negative having a weight of $W_n = 8.0$. It also helps to account for imbalance in the dataset: false instructions are far more common than true instructions.

We impose the following desired properties on the designed model:

1. **Recall close to** 100 % (almost no false-negatives). Instructions being omitted hurts performance for the binary translated code.
2. **High precision** (small amount of false-positives). While not critical as recall, it is also a metric to optimize for. Lower precision increases amount of instructions that have to be translated, hurting the performance of the binary translator and bloating the translated binaries with unreachable code.
3. **Portability.** We plan to embed the model into a binary translator, which should be able to run in environments not having GPU (and not requiring fiddly setup). Ideally, it should be exportable into ML-framework-independent format like ONNX [Bai, Lu, Zhang, 2019].
4. **Reasonable performance.** There are no hard limits on performance, but the model should no be much slower that alternative solutions like IDA Pro [IDA Pro Development Team, 1990] or DeepDi [Yu et al., 2022]. Note that only CPU performance is relevant, as, per the previous point, the model is intended to run in varying environments.

Model

The devised model is heavily inspired by DeepDi [Yu et al., 2022]. It is based on a relational graph convolutional neural network. It is designed to have high GPU performance, so all the steps, including the superset disassembly, are done on GPU (with CPU being available as a fallback). A visualization of the model can be seen in Fig. 2.

As a first step, same as DeepDi, our model constructs instruction embeddings from some metadata. Unlike DeepDi, which uses metadata gathered from internals of x86 disassembler, our model includes only instruction opcode and length. This is because it is difficult to extract the "internal" metadata without re-implementing an x86 disassembler. This metadata is passed through a simple fully-connected layer to reduce the embedding dimensionality.

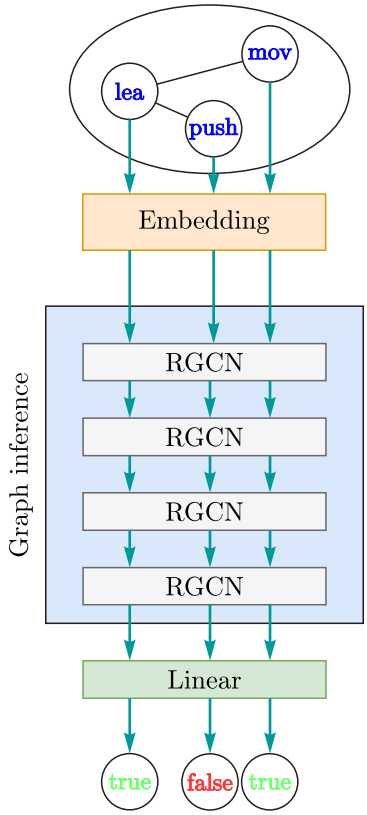

Figure 2. Visualization of the model

Next, DeepDi uses an RNN layer to incorporate neighboring instruction information. We believe that it is unnecessary: this information can be easily incorporated at the graph inference stage. Hence, our implementation omits the RNN layer.

After the embedding layer the R-GCN layer is located. It operates on a graph built from the superset disassembly that has instructions as nodes and edges carrying control and data flow information.

The Graph. The following are the kinds of edges used in our graph (see Fig. 3 for an example):

1. **Backward/Forward Control Flow edges.** Directed. Shows the "default" control flow, as the processor would execute it.

2. **Jump Control Flow edges.** Directed. Shows the "conditional" control flow for JCC family of instructions.

3. **Register Dependency edges.** Directed. Shows data dependencies. Connects instruction using the register to the instructions that set it.

4. **Overlap edges.** Undirected. Connects instructions that share some bytes (this is possible because I am using Superset Disassembly representation).

The main additions compared to DeepDi [Yu et al., 2022] are the Jump Control Flow and Register Dependency edges, inspired by Probabilistic Disassembly [Miller et al., 2019].

The R-GCN layer propagates the node embeddings through the differently-typed edges to gather information from neighbouring edges. This results each instruction having a vector describing the instruction itself and its relationship to the neighbours. This vector is then fed to the final fully-connected layer to predict whether the instruction is a true or a false one.

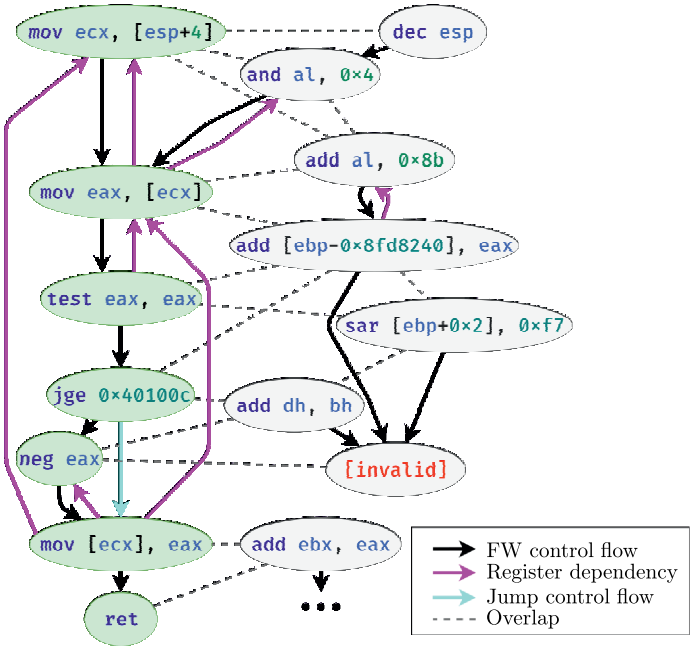

Figure 3. An example of graph. Note: true instructions are highlighted in green

Computing Register Dependency edges. All of the edges are quite straight-forward to compute from the Superset Disassembly, except the Register Dependency edges. This paragraph describes the algorithm to compute them.

For the purposes of tracking dependencies, only general purpose registers (`EAX`, `EBX`, `ECX`, `EDX`, `ESP`, `EBP`, `ESI`, `EDI`) and status flags (`CF`, `PF`, `AF`, `ZF`, `SF`) are considered. For simplicity, flags and registers are treated the same way (flags could be thought of as 1-bit registers).

Each instruction has a well-defined list of registers it reads from and writes to. To compute the Register Dependency edges, the algorithm needs to know, for each instruction, which instructions have previously written to each register. This can be achieved with a data structure storing, for each instruction, possible data sources for each register (data source being the address of an instruction that writes to the register).

What is not trivial, however, is the order of instruction nodes traversal. Ideally, if control flow graph (CFG) had a definable topological order, or, in other words, was acyclic, the topological order would be the way to go: before visiting an instruction, visit all instructions that pass execution to it. Unfortunately, normally the CFG has cycles in it, representing loops in the original program. This actually poses some challenge, as it is not possible to compute all data dependency edges in one

pass. One possible way to solve this is to use a quasi-topological sort. This will miss some Register Dependency edges, but may be acceptable for this task.

While some sophisticated algorithms for finding an approximate topological sort exist, simply going from lower instruction addresses to higher is a reasonable approximation: most of the CFG edges are pointing "downwards" (to the next instruction). Backward edges will not be considered this way.

The latter approach is the one that we employ. Possible future explorations may include a data dependency computation algorithm that does multiple passes to propagate the dependency information until no changes are observed (a fixed point). It is, however, unclear how many passes are needed to successfully propagate all data dependencies and, therefore, how well such an algorithm would scale.

Data collection

Extracting data from executables

PE and ELF are executable formats used on Windows and Linux, respectively. While the details are wildly different, the core information they carry is the same: process' initial memory image. This image is represented as a collection of memory regions with defined content, each of which is loaded at a particular address in memory. This essential information is all that is needed for the basic executable disassembler to work.

As such, most of the `identify-x86` code deals with this simplified model of an executable file. At the stage of dataset collection, executables are converted to this model, serialized using Rust's `bincode`[1] crate and compressed with `zstd`[2]. This greatly simplifies storage and handling of the dataset.

Another part required from the dataset is the ground truth disassembly. It is extracted using different methods from the PE and ELF files and stored alongside the memory image.

PE ground truth. To get ground-truth disassembly from PE files a PDB debug information is used. One part of PDB is symbols. There are a lot of different types (over 100), but here are the relevant ones that contain information about code and data:

1. `S_LDATA32`, `S_LDATA32_ST`, `S_GDATA32` and `S_GDATA32_ST`, which denote static data.
2. `S_GPROC32`, `S_GPROC32_ST`, `S_LPROC32`, `S_LPROC32_ST`, `S_LPROC32_DPC`, `S_GPROC32_ID`, `S_LPROC32_ID`, `S_LPROC32_DPC_ID` and `S_TRAMPOLINE`, which denote code.

These symbols can provide us with the ground-truth disassembly, however there are a couple of problems:

1. The "Data" symbols do not have a reliable way to get their sizes. We can guess it by assuming the next code symbol stops the data.
2. Jump tables for `switch`-case statements generated by MSVC are reported as being part of the function. This can be handled by making data have priority over the code.

The overall algorithm for finding the ground-truth disassembly in PE files is as follows:

1. Find the sizes for data symbols by assuming they span from their beginning to the next code symbol.
2. Mark the regions specified by code symbols as being code.

[1] https://docs.rs/bincode/latest/bincode/
[2] https://docs.rs/zstd/latest/zstd/

3. Unmark all data regions as not being code to handle the MSVC jump tables.

4. Linearly disassemble each code region to yield the true instruction offsets.

Besides handling the PDB files, there is another roadblock with getting a dataset for PE files: lack of good source of PE executables accompanied by full debug information. One good source might have been the Windows itself, as Microsoft publishes PDB files for Windows components to aid in debugging, those PDB files are known to be incomplete, making them useless for the task of full executable disassembly.

This, unfortunately, led to a considerable under-representation of PE files in the dataset: there are only 56 (compared to 909 ELFs).

ELF ground truth. Like the PDB, ELF also has a notion of symbols, which have an associated type. The interesting ones are `STT_OBJECT` and `STT_COMMON` (which denote data) and `STT_FUNC` (which denotes code).

There are, however, some common inaccuracies in the symbols: functions defined in the CRT startup code (`crt1.o`, `Scrt1.o`, `crti.o`, `crtn.o`, `crtbegin.o`, `crtbeginS.o`) do not have the size set correctly, probably due to them being generated in some roundabout way, being a low-level part of `glibc`. Parts of this code are included in all `gcc`-generated executables, so it was quite important to know their size to correctly mark the code region they represent.

The following symbols are matched manually: `_start`, `__x86.get_pc_thunk.ax`, `__x86.get_pc_thunk.bx`, `__x86.get_pc_thunk.cx`, `__x86.get_pc_thunk.dx`, `__x86.get_pc_thunk.bp`, `__x86.get_pc_thunk.si`, `__x86.get_pc_thunk.di`, `__i686.get_pc_thunk.ax`, `__i686.get_pc_thunk.bx`, `__i686.get_pc_thunk.cx`, `__i686.get_pc_thunk.dx`, `__i686.get_pc_thunk.bp`, `__i686.get_pc_thunk.si`, `__i686.get_pc_thunk.di`, `deregister_tm_clones`, `register_tm_clones`, `__do_global_dtors_aux`, `frame_dummy`, `_init`, `_fini`. To find the size of functions denoted by these symbols we utilize the fact that the functions themselves are do not change for the fixed version of compiler. The only part that differs is absolute offsets in some jump and call instructions in these functions. As such, they can be matched by a form of regular expression on bytes:

```
\x31\xED \x5E \x89\xE1 \x83\xE4\xF0 \x50 \x54 \x52
\x68.... \x68.... \x51 \x56 \x68.... \xE8.... \xF4
|
\x31\xED \x5E \x89\xE1 \x83\xE4\xF0 \x50 \x54 \x52
\xE8\x22\x00\x00\x00 \x81\xC3.... \x8D\x83....
\x50 \x8D\x83 .... \x50 \x51 \x56 \xFF\xB3....
\xE8.... \xF4
\x8B\x1C\x24\xC3
```

This example pattern matches two versions of the `_start` function. `\xXX` match the `XX` byte (in hex), while `.` matches any byte. `|` separates different variants of the function. Different instructions are separated by spaces for marginally better readability.

By fully matching the function code, it is possible to figure out the length of the function, filling in the unspecified length of these symbols.

Remaining handling of this ELF symbol information is straightforward: symbols marked as code are marked as code regions, data symbols are marked as data. As some executables were wound to contain incomplete symbol information, the coverage metric is used to discard those executables. The coverage is just a percentage of the memory marked as either code or data. If there is too much unmarked bytes (more than 25 %) it is not included in the dataset.

Unlike Windows, Linux distributions have a habit of publishing full debug information for the distributed executables. A big part of the dataset uses Debian executables, utilizing the debug info infrastructure (see next paragraph).

Dataset acquisition

Our dataset is a combination of ByteWeight dataset [Bao et al., 2014] and select Debian packages, all converted to the sample format described in the previous paragraph. Our dataset is published at https://huggingface.co/datasets/DCNick3/identify-x86.

The acquisition of ByteWeight data is straightforward, one just needs to convert the executables to the custom sample format.

The part that consists of Debian packages is a bit more involved. Debian packages do not include debug information by themselves, but rather Debian provides separate packages containing split debug information. Both the executable and the debug info file are ELF files, complementing each other. They are matched using build-id embedded into the executable.

Debian Jessie doesn't have a general way to handle debug info packages. Instead, by convention, some packages have companion packages ending in `-dbg`. The amount of such packages is limited. Debian buster has an additional mirror hosting the debug packages: http://debug.mirrors.debian.org/debian-debug. It has debug info packages for most of the Debian packages, with `-dbgsym` being appended to the name.

The fetching program uses `debian-packaging` [Szorc, 2023] crate to parse and download data from the Debian repositories. It scans the repository, downloads requested packages and their debug information, locates ELF files in each package, finds debug info file corresponding to that ELF file and converts the ELF file to the sample file described above.

After fetching the samples, another program is ran to convert the sample files into graph representation, suitable for the neural network to train on. Graphs are stored using a standard NPY format used by `numpy` [Harris et al., 2020], albeit using Zstd [Collet, Kucherawy, 2018] compression for better space utilization.

Pre-processing scaling

Initially, we planned to do the pre-processing step in Python to reduce inter-language friction. It turned out, however, that Python is not suitable for this task. Graphs for some executable files turned out quite big, having around 2M nodes and around 18M edges. As one might guess, Python handled it slowly. However, the biggest problem was memory consumption. As each Python object has a really high overhead (each integer, for example, takes 28 bytes!), constructing such a huge graph required some tricks to not lead to out-of-memory condition.

Hence, we use Rust for all the pre-processing, which greatly helps with pre-processing scalability, both in time requirements and memory usage. This language switch decreased processing time by about 10x.

Dataset split procedure

To minimize overlap between train and test sets, the similarity between executables was taken into account when performing the splitting. First of all, each executable was converted into a multi-set of 4-grams. Then, pairwise Jaccard Similarity was computed using the following formula:

$$\frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A}| + |\mathbf{B}| - |\mathbf{A} \cap \mathbf{B}|},$$

where $\mathbf{A}$ and $\mathbf{B}$ are n-gram multi-sets being compared and $|\mathbf{X}|$ is the size of multi-set $\mathbf{X}$. Afterwards, a graph is built, where two executables have an edge between them iff Jaccard Similarity exceeded the

threshold of 0.3. The connected components of the graph are then treated as inseparable groups of the dataset.

After this splitting was just a matter of splitting the inseparable groups into test/train bins. To do this, a greedy approach was employed: starting from the largest group, the most underfull bin was selected for each group. With necessarily diverse group sizes, which was achieved in practice, this algorithm achieved the desired test/train ratio, while keeping similar executables in the same bin.

Evaluation

Accuracy

The binary classification task that is being solved has a serious class imbalance: there are about 5 times more false instructions than true ones. As such, the model accuracy is evaluated using precision, recall and F1-score (combining the precision and recall).

Table 1 shows comparison of these metrics between DeepDi [Yu et al., 2022], IDA Pro [IDA Pro Development Team, 1990], and `identify-x86`, run on the test set of `identify-x86`.

Table 1. Overall test set metrics

	Precision (%)	Recall (%)	F1 Score (%)
DeepDi	94.290	99.917	97.008
IDA Pro	98.044	96.660	97.341
identify-x86	**99.661**	**99.935**	**99.798**

`identify-x86` demonstrates significantly better precision than DeepDi, while maintaining the same recall. It also outperforms IDA Pro on both of those metrics. This may be explained by the additional graph features used in `identify-x86` model.

Curiously, there is a difference in accuracy on executables produced by different compilers. Table 2 shows the metrics for the same models, but separate for each compiler.

Table 2. Per-compiler test set metrics

	Precision (%)		Recall (%)		F1 Score (%)	
	gcc	msvc	gcc	msvc	gcc	msvc
DeepDi	94.300	93.754	99.946	98.323	97.028	95.950
IDA Pro	98.051	97.658	96.632	98.210	97.332	97.869
identify-x86	99.685	98.329	99.946	99.322	99.815	98.817

It can seen that IDA Pro has better recall on MSVC-generated binaries, probably due to having more heuristics for this compiler: IDA Pro was historically focused on Windows binaries, adding support for other platforms later.

It is also notable that both DeepDi and `identify-x86` perform worse on MSVC binaries. For `identify-x86`, this can be attributed to the fact that Windows binaries are heavily underrepresented in the dataset (56 out of total 965 executables). This is due to difficulty of obtaining PE executables with full debug information, detailed in Data Collection section above. Such imbalance is much less pronounced in DeepDi's dataset, but it has less PE files nonetheless.

Other possible explanation might that MSVC-generated executables are simply harder to disassemble. The fact that MSVC generates jump tables for switch-case statements inside the `.text` data, disrupting otherwise instruction-filled paragraph, supports that fact.

Performance

To measure the disassembly performance, we measure the time to process a sample across the entire dataset (train and test sets combined). The goal for `identify-x86` is to make it easily embeddable, so all the measurements were done on CPU rather than GPU.

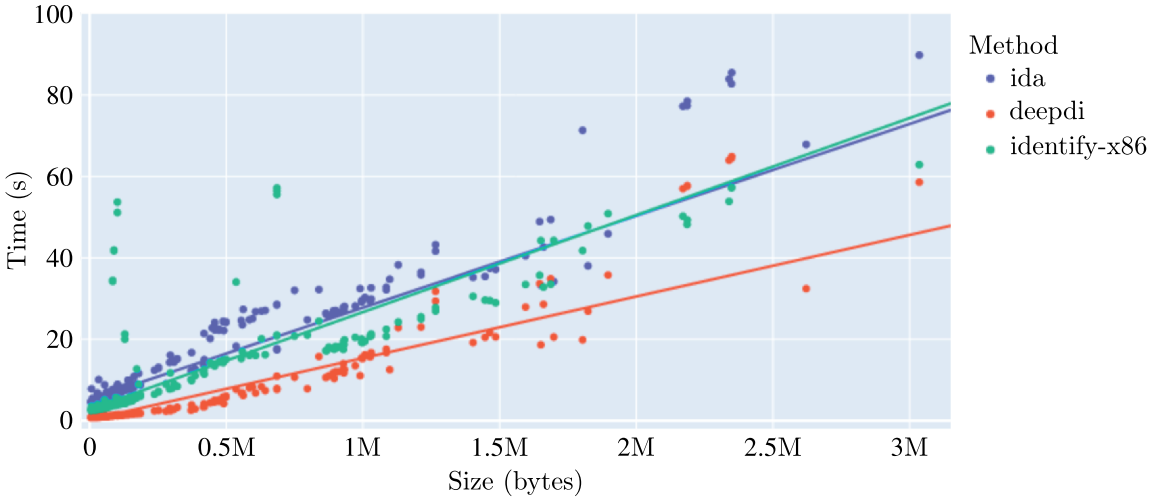

Figure 4. Processing time Least Squares Fit

While there were some outliers, the time scaled linearly with the size of sample (in bytes). The plot of this relationship, along with linear least squares fit for a line can be seen at Fig. 4.

The slopes corresponding to the fit linear least squares functions can be seen in Table 3.

Table 3. Model performances

	Speed (KiB/s)
DeepDi	64.56
IDA Pro	43.25
identify-x86	40.95

The speed of `identify-x86` is just slightly lower than IDA Pro, with DeepDi being significantly faster. This is expected, as, in addition to the features used in DeepDi, `identify-x86` also uses hard-to-compute data dependency information, rendering it slower. The performance is still reasonable, so, when more accuracy is desired, `identify-x86` can be beneficial.

Conclusion

We have created an open-source dataset for ML-based x86 disassembly, as well as tooling for dataset collection, prompting extensions of the dataset. We have also proposed and implemented a graph neural network model for fast and accurate x86 disassembly which we trained and tested on the aforementioned dataset, comparing our implementation against existing state-of-the-art solutions — IDAPro and DeepDi.

Unlike DeepDi, the graph construction is not designed to run on GPU, so its GPU performance would be severely limited by the graph construction. Never being designed with high performance in mind, `identify-x86` still has lower CPU performance than DeepDi.

Future work

There is little variety in the dataset in terms of compilers. Most of the executables are produced by GCC for Linux, with small portion being generated by MSVC. Other compilers, possibly even for different programming languages, should improve robustness of the model.

Some binaries' graphs were too large and caused CUDA out-of-memory errors during training. They were removed from the dataset to let the training proceed. Some procedure to split those graphs into smaller ones should be devised to make it possible to train on those executables.

While there is some tooling for fetching data, generating graphs in bulk and evaluating the model, some common use-cases are not yet covered by it.

Unfortunately, it is not very easy to use the model in programming languages other than python. As outlined in Methodology, some way for improved embeddability should be pursued, be it improving ONNX [Bai, Lu, Zhang, 2019] support in `pytorch_geometric` [Fey, Lenssen, 2019] or re-implementing the model from scratch in other languages.

References

Andriesse D., Chen X., Van Der Veen V., Slowinska A., Bos H. An in-depth analysis of disassembly on full-scale x86/x64 binaries // Proceedings of the 25th USENIX Conference on Security Symposium. — 2016. — P. 583–600.

Andriesse D., Slowinska A., Bos H. Compiler-agnostic function detection in binaries // 2017 IEEE European Symposium on Security and Privacy (EuroS&P).. — 2017. — P. 177–189.

Bai J., Lu F., Zhang K. ONNX: open neural network exchange // GitHub. — 2019. — https://github.com/onnx/onnx

Bao T., Burket J., Woo M., Turner R., Brumley D. BYTEWEIGHT: learning to recognize functions in binary code // 23rd USENIX Security Symposium (USENIX Security 14). — 2014. — P. 845–860.

Bauman E., Lin Z., Hamlen K. Superset disassembly: statically rewriting x86 binaries without heuristics // NDSS. — 2018.

Collet Y., Kucherawy M. Zstandard compression and the application/zstd media type. — 2018.

Fey M., Lenssen J. Fast graph representation learning with PyTorch geometric // ICLR Workshop on Representation Learning on Graphs and Manifolds. — 2019.

Ghidra Development Team. Ghidra. — 2019. — https://ghidra-sre.org/

Harris C., Millman K., Walt S., Gommers R., Virtanen P., Cournapeau D., Wieser E., Taylor J., Berg S., Smith N., Kern R., Picus M., Hoyer S., Kerkwijk M., Brett M., Haldane A., Río J., Wiebe M., Peterson P., Gérard-Marchant P., Sheppard K., Reddy T., Weckesser W., Abbasi H., Gohlke C., Oliphant T. Array programming with NumPy // Nature. — 2020. — Vol. 585. — P. 357–362.

Hopfield J. Neural networks and physical systems with emergent collective computational abilities // Proceedings of the National Academy of Sciences. — 1982. — Vol. 79. — P. 2554–2558.

IDA Pro Development Team. IDA Pro // https://www.hex-rays.com/ida-pro/

Miller K., Kwon Y., Sun Y., Zhang Z., Zhang X., Lin Z. Probabilistic disassembly // 2019 IEEE/ACM 41st International Conference On Software Engineering (ICSE). — 2019. — P. 1187–1198.

Pei K., Guan J., Williams-King D., Yang J., Jana S. XDA: accurate, robust disassembly with transfer learning // arXiv preprint. — 2020. — arXiv:2010.00770

Schlichtkrull M., Kipf T., Bloem P., Berg R., Titov I., Welling M. Modeling relational data with graph convolutional networks // arXiv preprint. — 2017. — arXiv:1703.06103

Szorc G. Debian-packaging: Debian packaging primitives // GitHub. — 2023. — https://github.com/indygreg/linux-packaging-rs.git

Yu S., Qu Y., Hu X., Yin H. DeepDi: learning a relational graph convolutional network model on instructions for fast and accurate disassembly // 31st USENIX Security Symposium (USENIX Security 22). — 2022. — P. 2709–2725.