Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра информационной безопасности

Телепов Владимир Юрьевич

Распознавание паттернов в декомпилированном коде

Выпускная квалификационная работа

Научный руководитель:
сотрудник кафедры ИБ
М.С.Воронов

Москва, 2020

Аннотация

В данной работе исследуется возможность определения поведения функций в декомпилированном коде. Также в работе обсуждаются подходы по распознаванию семантики кода. Был предложен и практически реализован метод в виде утилиты для Ida Pro и набора скриптов для предсказания вероятностей принадлежности функций в декомпилированном коде одному из 104 классов алгоритмов.

Оглавление

Введение

Исследование поведения функций в декомпилированном коде является важным по нескольким причинам. Во-первых, анализ функций может быть использован для обнаружения вредоносного программного обеспечения. Благодаря постоянному развитию и совершенствованию вредоносных программ компании терпят огромные убытки и трудозатраты для ликвидации последствий причиненного ущерба. Качественный анализ может предотвратить эти нежелательные последствия и сократить вынужденные денежные потери. Во-вторых понимание поведения функций в декомпилированном коде может быть использовано для улучшения качества и скорости обратной разработки. Такое исследование программ применяется с целью получения некоторых закрытых сведений о внутреннем устройстве программы, когда разработчик предоставил приложение без исходных кодов, а также с целью поиска уязвимостей и может быть использовано для написания программных средств защиты информации.

Упомянутая выше задача является частным случаем более общей задачи исследования семантики кода. Решение этой задачи может быть использовано для решения многих практических задач:

- Автоматическая проверка кода - рекомендация названий имен функций может улучшить читаемость и поддержку кода и упростить использование открытого API, также может использоваться для подсказки правильного названия метода при разработке в IDE;
- Автодополнения кода;
- Генерация кода;
- Распознавание дублирования кода;
- Предсказание оптимальных параметров для запуска программы;
- Определение класса алгоритма(ускорение reverse-engineering) и др.

1 Цель работы

Разработать и реализовать метод классификации функций в бинарной программе по 104 классам, определённым в наборе тестовых данных POJ-104[1].

1.1 Постановка задачи

1. Рассмотреть применимость текущих подходов к распознаванию паттернов в исходном коде, основанных на машинном обучении, к распознаванию паттернов в декомпилированном коде.

2. Разработать метод классификации функции в бинарной программе при полном отсутствии отладочной информации.

3. Реализовать и протестировать разработанный подход.

2 Описание предметной области

В дальнейшем тексте будут неоднакратно использоваться различные понятия из машинного обучения и внутреннего представление LLVM IR. В этой секции приводится краткое описание нужных терминов.

2.1 Связь с обработкой естественных языков

Рассмотрим вкратце, как именно машинное обучение может быть применимо к задаче анализа исходного кода. В области машинного обучения и искусственного интелекта есть отдельное ответвление, занимающееся обработкой естественного языка (Natural Language Processing, NLP), которое изучает проблемы компьютерного анализа и синтеза естественных языков. Код на языке программирования является искусственным языком, имеющим строгий синтаксис и семантику, в отличие от естественного языка. Искусственный и естественный языки имеют и другие отличия, однако существует гипотеза, что тексты на языке программирования имеют некоторые статистические особенности, схожие с текстами на естественном языке, и эти особенности могут быть использованы для построения инструментов анализа кода программ. Поэтому существуют методы обработки текстов, которые могут также быть применены к анализу исходного кода программ.

2.2 Векторное представление слов

Во многих методах обработки текстов на естественном языке с помощью машинного обучения используется подход, направленный на получение представление единицы текста(слова, предложения и т.п) в виде вещественного вектора. Это так называемое векторное представление слов (embedding). Основной целью данного подхода является получение признаков, представляющих информацию об исходных текстах в пригодном для метода формате, т.к большинство методов машинного обучения принимают на вход в качестве признаков набор вещественных векторов.

2.3 LLVM

LLVM - проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня, системы оптимизации, интерпретации и компиляции в машинный код. В основе инфраструктуры используется RISC-подобная платформонезависимая система кодирования машинных инструкций - байткод LLVM IR, которая представляет собой высокоуровневый ассемблер, с которым работают различные преобразования. Над промежуточным представлением можно производить трансформации во время компиляции, компоновки и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ как статически, так и динамически.

Проект LLVM написан на C++ и портирован на большинство Unix-подобных систем и Windows. Система имеет модульную структуру, отдельные её модули могут быть

встроены в различные программные комплексы, она может расширяться дополнительными алгоритмами трансформации и кодогенераторами для новых аппаратных платформ.

2.4 LLVM IR

LLVM IR - это промежуточное представление в виде трёхадресного кода в SSA-форме. На практике для хранения кода используется эффективное бинарное представление(bitcode). SSA — это такая форма промежуточного представления кода, в которой любое значение присваивается только один раз.

LLVM IR поддерживает следующие типы данных:

- целые числа произвольной разрядности. Генерация машинного кода для типов очень большой разрядности не поддерживается, но для промежуточного представления никаких ограничений нет;
- числа с плавающей точкой: float, double, а также ряд типов, специфичных для конкретной платформы(например, x86_fp80);
- void — пустое значение;
- указатели;
- массивы;
- структуры;
- функции;
- векторы.

Система типов рекурсивна, поэтому можно использовать многомерные массивы, массивы структур, указатели на структуры и функции, и т. д.

Большинство инструкций в LLVM принимают два аргумента и возвращают одно значение. Значения определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные — @. Локальные значения также называют регистрами, а LLVM — виртуальной машиной с бесконечным числом регистров.

Тип операндов всегда указывается явно, и однозначно определяет тип результата. Операнды арифметических инструкций должны иметь одинаковый тип, но сами инструкции «перегружены» для любых числовых типов и векторов. Всего существует 52 инструкции, среди них:

- набор арифметических операций;
- набор побитовых логических операций и операций сдвига;
- специальные инструкции для работы с векторами;
- инструкции для обращения к памяти;

- операции приведения типа.

Из этого краткого обзора видно, что промежуточное представление LLVM достаточно близко соответствует коду на низкоуровневых процедурных языках наподобие Си. При трансляции высокоуровневых языков - объектно-ориентированных, функциональных, динамических - придётся выполнить гораздо больше промежуточных преобразований, а также написать специализированный интерпретатор. Но и в этом случае LLVM снимает с разработчика компилятора проблемы кодогенерации для конкретной платформы, берёт на себя большинство независимых от языка оптимизаций — и делает их качественно. Кроме этого, с помощью LLVM разработчик получает готовую инфраструктуру для динамической компиляции и возможность оптимизации времени связывания между различными языками, компилируемыми в LLVM.

3 Описание существующих подходов

Задача автоматизированного выявления паттернов в бинарных программах известна достаточно давно (одной из первых работ в этой области является [4]). Существующие способы решения используют в основном статический и динамический анализ, основанный на различных эвристиках. Но сравнительно недавно стали очень активно развиваться методы анализа, основанные на машинном обучении. В этом разделе рассматриваются как раз два существующих на данный момент подхода по распознаванию семантики исходного кода с анализом их плюсов и минусов в контексте применения для выявления паттернов в скомпилированных бинарных программах.

3.1 Code2Vec

Основная идея code2vec[2] состоит в получении векторного представления для кода функции на некотором языке программирования. Авторы решали задачу получения имен функций на языке Java по их исходному коду. Решение выглядело следующим образом:

- для функции строилось абстрактное синтаксическое дерево, из него извлекались пути между листовыми вершинами;
- каждый путь заменялся на свое векторное представление;
- каждый полученный вектор проходил через один и тот же полносвязный слой с нелинейным преобразованием;
- для каждого полученного вектора с помощью attention mechanism рассчитывался коэффициент, с которым данный вектор войдет в взвешенную сумму;
- считалась взвешенная сумма всех векторов, которая считалась векторным представлением функции;
- для каждого названия функции встреченного в выборке считалось скалярное произведение векторного представления названия функции и вектора значимости, и от полученного набора векторов вычислялась функция активации softmax;
- для полученного распределения вероятностей меток функций считалась функция ошибки cross-entropy loss и с помощью методов оптимизации производилась настройка обучаемых параметров: матрица векторных представлений для путей, матрица полносвязного слоя, вектор значимости и матрица векторных представлений названий функций.

Схема модели code2vec представлена на рисунке [1].

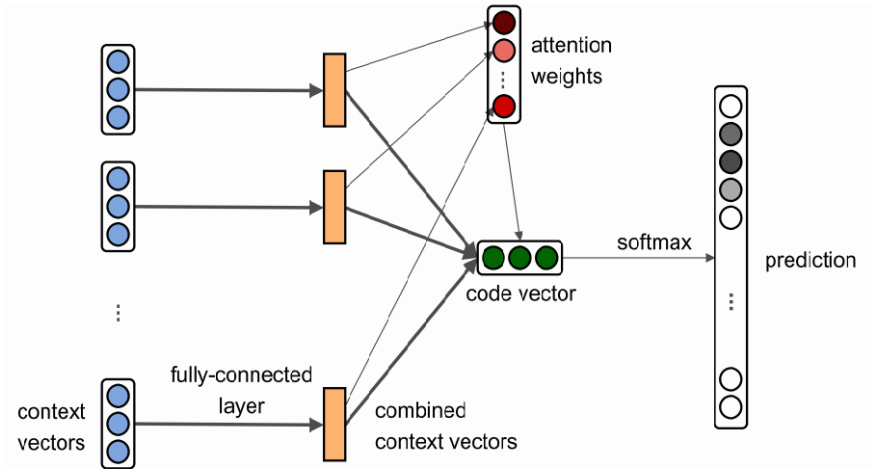

Рис. 1: Схема модели code2vec

После предварительного изучения было выявлено, что основным недостатком данной модели является большая зависимость от имён переменных и аргументов функции, а также их количества. Т.к. обучение производится на AST-путях между аргументами и локальными переменными. Code2vec может быть применён к рассматриваемой задаче посредством анализа декомпилированного кода, полученного с помощью декомпиляторов (например, с помощью Hex-Rays из Ida Pro). Но ввиду упомнутых недостатков данный подход не сможет функционировать с хорошей точностью, т.к. в декомпилированном представлении бинарного файла без отладочной информации не сохраняются оригинальные имена переменных и аргументов.

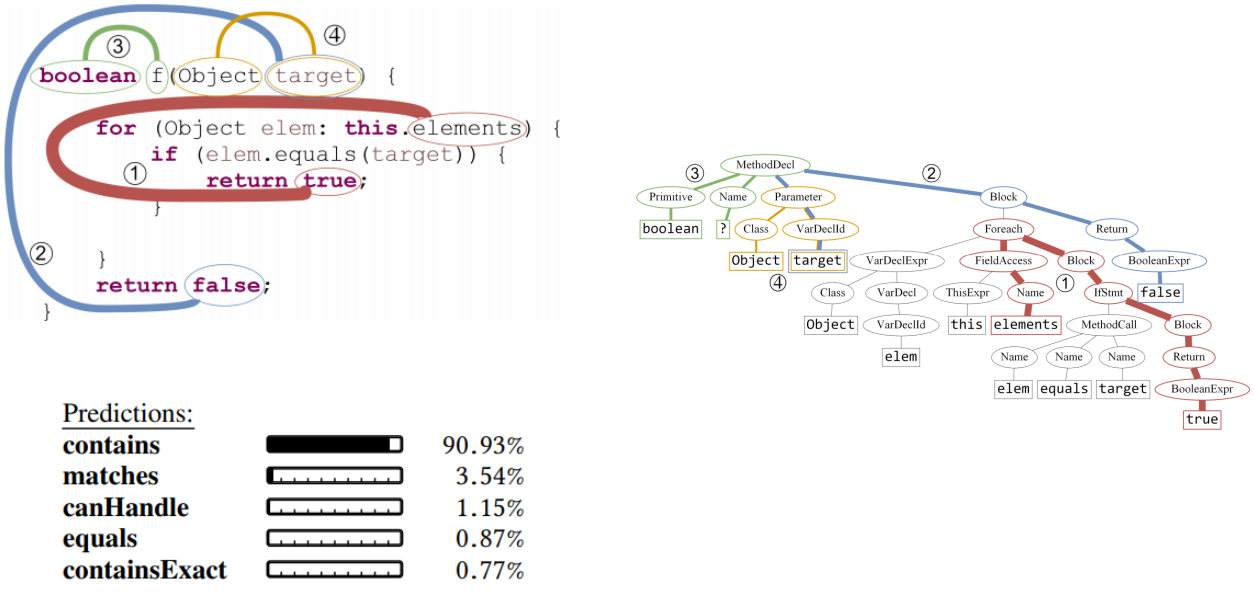

Рис. 2: Код функции на языке Java с выделенными AST-путями и ее AST дерево.

3.2 Inst2vec

Используя метод inst2vec[3] авторы предложили решение сразу 3 задач. Первая задача состояла в определении класса алгоритма. Во второй задаче надо было определить, на каком вычислительном устройстве(центральный процессор или видеокарта) выгоднее запустить программу, чтобы ее время исполнения было меньше. Третья задача заключалась в определении максимального количества нитей при запуске кода на видеокарте для минимизации скорости работы. Для решения этих задача использовался подход с обучением векторного представления инструкций.

Т.к исходный код для этих задач может быть написан на разных высокоуровневых языках, то с целью генерализации подхода авторы выбрали представление llvm ir, в которое может быть скомпилирован код на большом множестве языков, например, на C/C++, FORTRAN, Python, Java, CUDA, OpenCL и др. Было сделано предположение, что инструкции, которые находятся "близко"друг к другу, имеют похожий смысл. Поэтому для обучения векторных представлений были выбраны модели Word2vec и Skip-Gram, использование которых предполагает наличие понятия контекста слова(в данном случае инструкции).

Авторы определяют контекст инструкции, как множество инструкций, от которых напрямую зависит выполнение данной инструкции. Таким образом учитывается зависимость как по передаче управления так и зависимость по данным. Инструкции с вышеупомянутой зависимостью не могут быть напрямую извлечены из исходного кода, поэтому сначала строится XFG(contextual-flow graph) граф. Это ориентированный мультиграф, который определяется следующим образом: вершинами являются переменные или метки, например, базовые блоки, функции; ребра являются либо зависимостью по данным либо зависимостью по управлению; если у инструкции нет зависимости по данным, она соединяется ребром с корневой вершиной. Для того, чтобы словарь не был очень большим данные подвергались предобработке. Убирались комментарии и метаданные, константы заменялись на фиксированные токены, производилась подстановка структур.

После обучения векторных представлений для решения поставленных задач обучалась рекуррентная нейронная сеть, которая принимала на вход преобразованную в векторное представление последовательность инструкций llvm ir (рисунок 3).

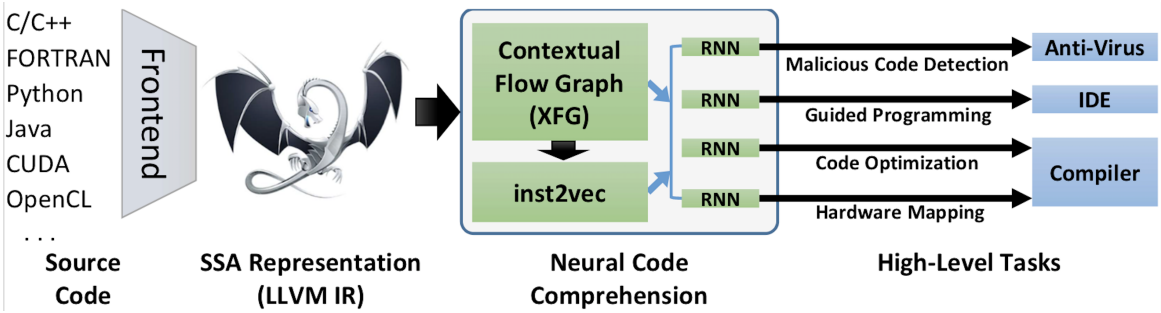

Рис. 3: Принципиальная схема модели inst2vec.

Данный подход имеет свои плюсы, важнейшими из которых является обощённость и возможность повторного использования векторных представлений. Однако использование рекуррентных нейронных сетей может вызывать необходимость использования большого количества вычислительных ресурсов.

4 Описание предлагаемого подхода

В этом разделе описывается предлагаемое решение.

4.1 Алгоритм классификации функций в бинарном коде

Программный комплекс, решающий задачу определения поведения функций, состоит из фреймворков McSema и NCC.

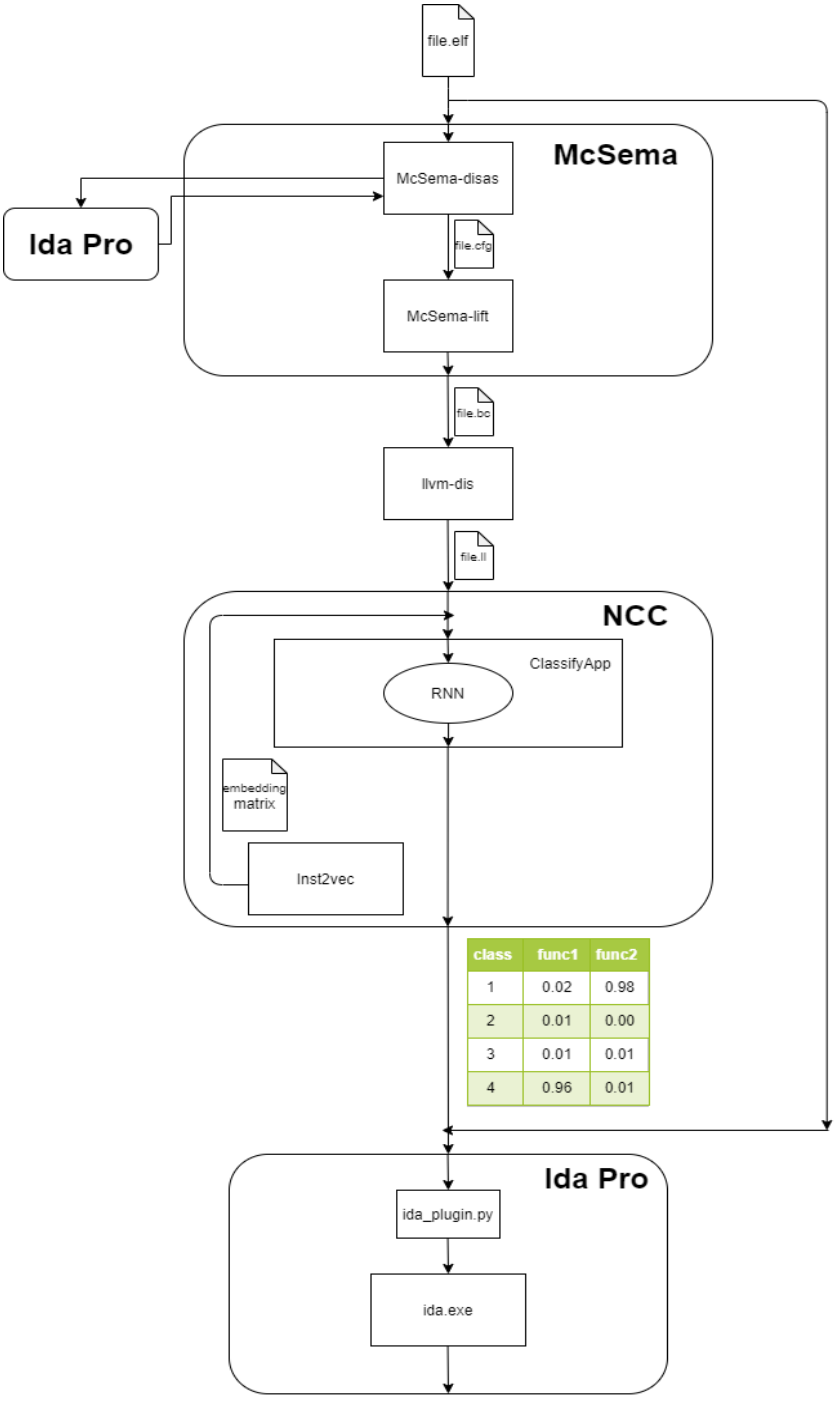

Рис. 4: Предлагаемый алгоритм классификации функций в бинарном файле.

Предлагаемый алгоритм получения меток для каждой функции в бинарном исполняемом файле выглядит следующим образом:

1. На первом этапе исполняемый бинарный файл подается в программу mcsema-disas из фреймворка McSema, из него извлекается граф потока управления;
2. Полученный на предыдущем шаге файл подается в программу mcsema-lift из фреймворка McSema, которая переводит файл в llvm биткод;
3. Файл с llvm биткодом переводится программой llvm-dis из фреймворка LLVM в представление llvm ir;
4. Полученный файл подается на вход программы classifyapp. Программа выполняет следующие действия:
   (a) Файл с llvm ir разбивается на функции, каждая функция сохраняется в отдельном файле;
   (b) Все инструкции в файлах заменяются на их векторные представления;
   (c) Файлы поступают на вход рекуррентной нейронной LSTM сети, которая возвращает распределение вероятностей принадлежности функции 104 классам;
   (d) Из полученного распределения сохраняются в отдельный файл в json формате 3 меток классов с максимальными вероятностями
5. На последнем шаге с помощью плагина полученный json файл загружается в Ida Pro и для каждой функции создаётся комментарий с 3 полученными меткам классов и распределением вероятностей.

Данный алгоритм изображён на рисунке 4. Таким образом правильность полученных меток зависит от обученной нейронной рекуррентной сети и качества векторных представлений. Ниже представлен результат работы данного алгоритма, который для функции sort, представляющей из себя сортировку пузырьком, выдает максимальную вероятность для 102 класса. В 102 классе как раз содержатся программы решающие задачу сортировки структур по ключу.

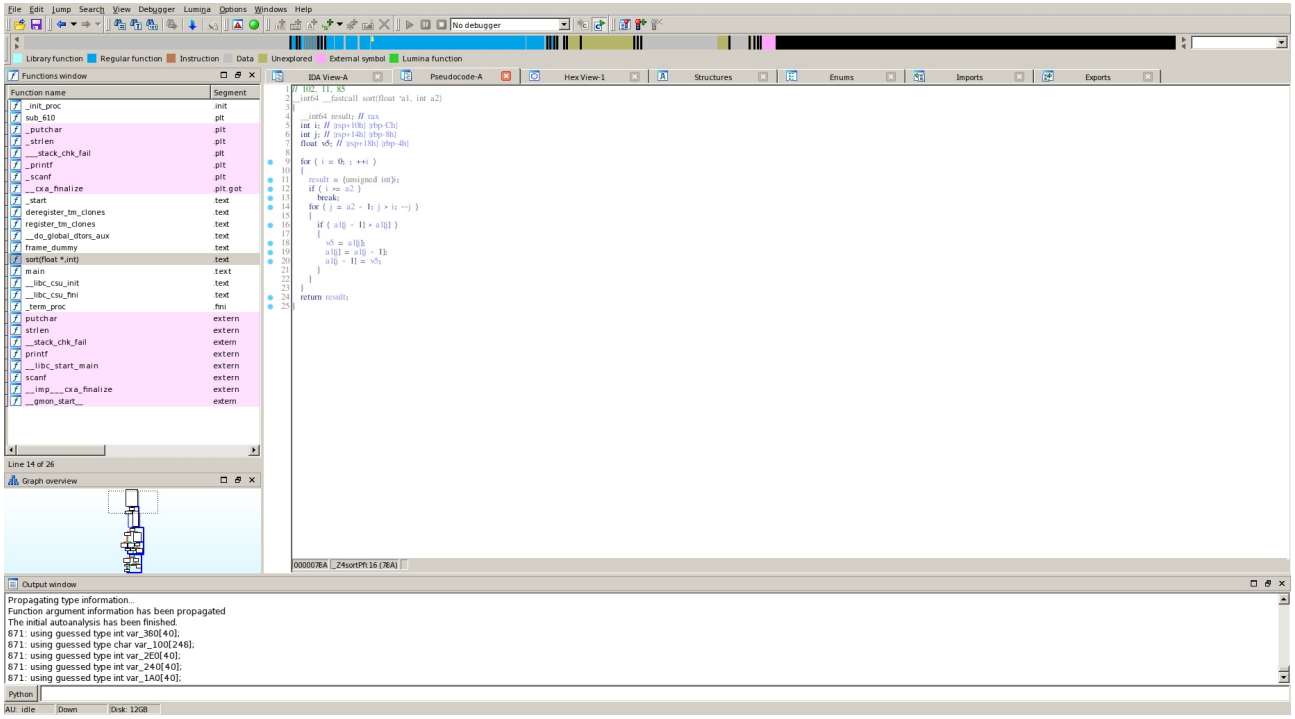

Рис. 5: Предсказание алгоритмом меток класса в Ida.

4.2 Обучение векторных представлений

Для обучения векторных представлений использовались в качестве датасета скомпилированная в llvm ir библиотека musllvm, которая является аналогом стандартной библиотеки языка C. Для корректной обработки файлов фреймворком inst2vec была произведена модификация нескольких файлов на языке Python. Также из датасета были удалены файлы, содержащие только объявления функций и константы. Модель Word2Vec состояла из двух полносвязных слоев и функции активации softmax. В качестве функции ошибки использовалась функция cross-entropy loss и метод оптимизации Adam. На рисунке 5 представлен график зависимости функции ошибки от количества итераций. Данная зависимость похожа на экспоненциальную, что говорит о сходимости процесса оптимизации.

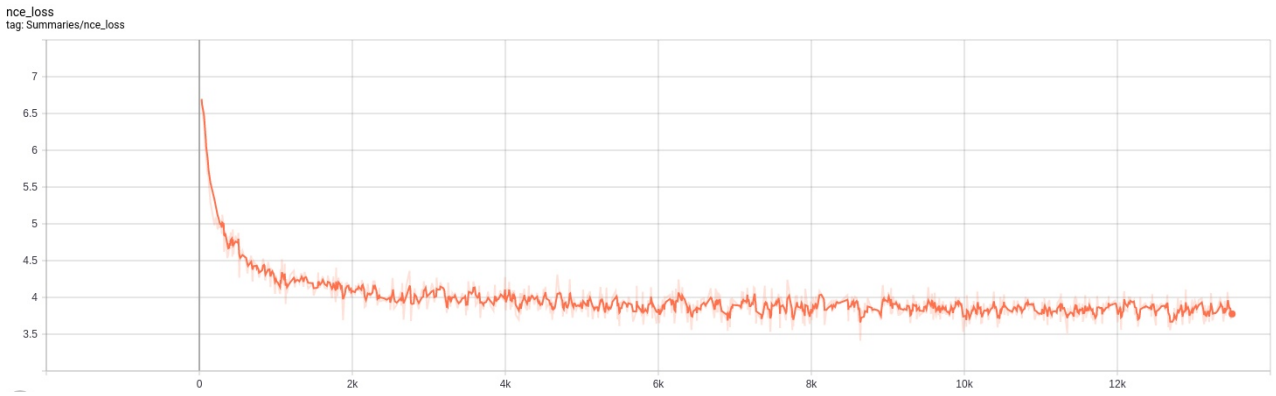

Рис. 6: График зависимости функции ошибки от количества итераций.

Однако в дальнейшем полученные векторные представления не использовались, а были взяты обученные создателями фреймворка NCC, т.к их датасет содержал большее количество файлов и содержал большее разнообразие функций, и соответственно имел большее качество.

4.3 Обучение рекуррентной нейронной сети

Рекуррентная нейронная сеть состояла из двух LSTM блоков, слоя батч-нормализации и двух полносвязных слоев с функциями активации ReLU и сигмоидой соответственно (рисунок 6).

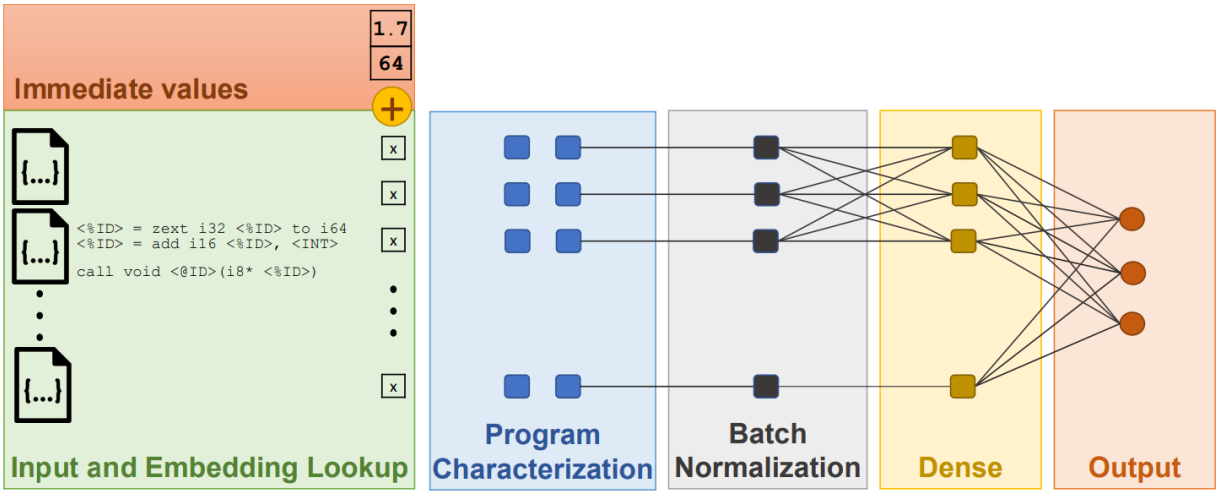

Рис. 7: Схема рекуррентной нейронной сети

В качестве функции ошибки использовалась функция cross-entropy loss и метод оптимизации Adam. Нейронная сеть создателей фреймворка обучалась на скомпилированных в llvm ir Файлах. Использовать ее для предсказания классов не представляется возможным, потому что файлы, прошедшие через Mcsema и llvm-dis очень отличаются от просто скомпилированных в llvm ir наличием большого числа метаданных и появившимися при компиляции вспомогательными функциями.

В качестве датасета был взят использованный авторами датасет POJ-104, в исходных файлах на языках C/C++ были добавлены заголовочные файлы библиотек и заменено обьявление функции main добавлением типа возвращаемого значения int для возможности компиляции всех файлов компилятором clang. После этого все файлы были скомпилированы 8 раз со случайным выбором уровня оптимизации и случайным выбором компилятора(g++/clang).

Затем файлы подавались на вход фреймворку mcsema, а после llvm-dis. Было замечено, что код функций, который был исходных файлах, в .ll файлах содержался только в функциях, имеющих такое же имя или имеющих такое же имя и префикс 'sub', поэтому был удален код всех не удовлетворяющих этим названиям функций, чтобы уменьшить размер датасета и улучшить качество предсказаний.

Однако первом запуске обучения на полученном датасете программа завершилась сообщением об ошибке, фреймворк tensorflow превысил ограничение по памяти, заняв всю имеющуюся на сервере оперативную память. Скорее всего это происходило из-за использования некоторых функций фреймворка, в которых была утечка памяти. Было заменено API для обучения модели на другой эквивалентный, после этого процесс обучения не завершался с ошибкой, однако был слишком долгий и все равно потреблял большую часть оперативной памяти на сервере. Было замечено, что максимальная длина последовательности в нашем датасете больше чем в 10 раз превышала длину на датасете создателей фреймворка, а так как обучение проводилось в мини-батчевом режиме, все последовательности дополнялись до максимальной длины. Также было замечено что больше 90% последовательностей в нашем датасете имели длину меньше чем 3000 инструкций.

После этого все последовательности дополнялись до указанной длины или обрезались, если их длина была больше. После этих модификаций процесс обучения стал занимать приемлемое количество времени и памяти (одна эпоха обучения порядка 80 минут).

4.4 Результаты обучения рекуррентной нейронной сети

Была получена точность на 78.77% тренировочной выборке, 67.82% на валидационной выборке и 68.02% на тестовой выборке, графики зависимостей от количества эпох можно увидеть на рисунках 7 и 8. На обоих рисунках синяя линия соответствует тренировочной выборке, желтая - валидационной.

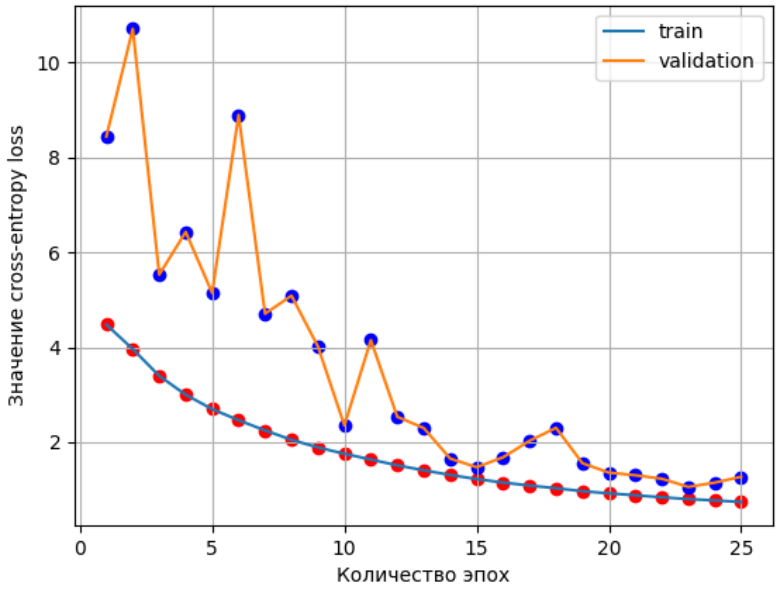

Рис. 8: График зависимости функции ошибки от количества эпох.

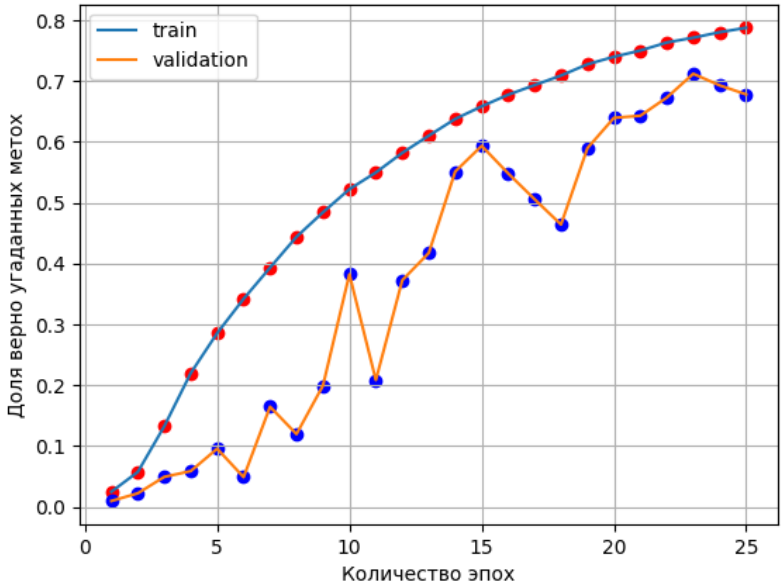

Рис. 9: График зависимости доли точно предсказанных классов от количества эпох.

4.5 Эксперименты

Чтобы убедиться в применимости метода были произведены эксперименты по получению точности предсказания, указанной в статье. При первой попытке обучения на полученном датасете программа завершилась из-за сообщения фреймворка tensorflow об очень большом графе вычислений, расположенном на видеокарте. Чтобы уменьшить количество вычислений было уменьшено количество примеров в тренировочной выборке и из исходной модели была убрана сигмоидная функция активации, т.к при больших значениях аргументов градиент сигмоидной функции становится маленьким, что может привести к очень долгому обучению или вообще невозможности работы оптимизационного процесса. Однако попытка обучения не удалась т.к значение функции ошибки притерпевало осцилляции около некоторого значения и в среднем не уменьшалось с количеством итераций.

После этой попытки в модель была обратно добавлена сигмоида и произведено обучение. Результаты тоже оказались неудовлетворительными, т.к на тренировочной выборке функция ошибки и доля правильно предсказанных меток росли, а на валидационной выборке эти значения притерпевали большие осцилляции. Была получена точность 94% на тренировочной выборке, 23% на валидационной и 5% на тестовой. Это означало, что модель сильно переобучилась. Дальнейшие эксперименты с заменой частей модели на уменьшенном датасете тоже давали либо совсем не обученные, либо переобученные модели.

Было сделано предположение, что размер датасета оказался слишком мал, поэтому происходит переобучение, а "недообучение"может происходить из-за неудачно выбранной модели. Была произведена замена кода, чтобы перейти на новую версию tensorflow, в которой разработчиками была решена данная проблема, также были удален код, который ограничивал использование памяти на видеокарте. После этого была обучена модель на полном датасете, получена 95% точность на тестовой выборке.

4.6 Ограничения подхода

Из предыдущей секции можно сделать вывод о том, что предложенный подход применим для анализа больших бинарных программ и сложных алгоритмов. Однако у данного подхода есть следующие ограничения:

1. Mcsema не всегда может перевести в llvm ir бинарный файл и результат ее работы сильно зависит от дизассемблера. Также нужно учесть, что результат работы диззассемблера для обфусцированных бинарных файлов может быть неудовлетворительным и, следовательно, данный подход не может быть применён.;
2. Структура полученного .ll файла очень сильно зависит от работы mcsema, поэтому при значительных изменениях ее ядра необходимо заново переобучать векторные представления и рекуррентную нейронную сеть.

5 Описание реализации

В рамках данной работы были написаны программы, необхимые для обучения рекуррентной нейронной сети и получения меток классов функций в Ida Pro. Программы написаны на скриптовых языках Python и bash.

5.1 Описание работы реализации

compile_data.py - скрипт, который компилирует исходный код на языках C/C++ в исполняемый бинарный файл, и с помощью фреймворка mcsema и программы llvm-dis переводит его в llvm ir. Данный скрипт принимает на вход путь до директории, в которой находится датасет, количество файлов, которое будет скомпилированно со случайным выбором уровня оптимизаций, путь до ida.exe, путь до llvm-dis и путь до mcsema.

get_functions.py - скрипт, который принимает путь до .ll файла, создает в текущей директории поддиректорию tmp, и создает файл с кодом каждой функции, которая есть в указанном .ll файле.

del_functions.py - скрипт, который принимает путь до директории с датасетом, состоящим из .ll файлов и директории, в которой находятся файлы с именами функций в исходных файлах на языке C/C++. Скрипт производит удаление вспомогательных функций, которые появились во время компиляции.

get_functions_from_c.py - скрипт, который создавал директорию, в которую для каждого файла на языке C/C++ создается файл, в котором находятся имена функций, которые были в исходном файле. Скрипт принимает на вход путь до датасета и путь, по которому будет расположена директория с такой же структурой и файлами, содержащими названия функций.

script.sh - скрипт, который запускает декомпиляцию бинарного файла в Ida Pro и подписывает как комментарий к функции вероятности принадлежностей функции к одному из 104 классов алгоритмов. Скрипт принимает на вход путь до ida.exe, путь до llvm-dis, путь до mcsema, путь до classifyapp.py, и путь до бинарного файла.

Компиляция датасета производилась на компьютере с процессором Amd Ryzen 5 2600x в операционной системе Ubuntu 18.04.4. При запуске программы с 10 потоками время компиляции составило примерно 20 часов.

5.2 Обучение векторных представлений

При использовании фреймворка NCC было выявлено, что словарь, составленный по поданному на вход датасету сравнивается со словарем, составленным авторами, и при остутствии какой-либо инструкции в собранном словаре в их словаре, производится принудительное завершение работы программы. С целью произвести обучение все такие инструкции были заменены на специальную инструкцию 'unknown'. Обучение производилось на домашнем компьютере с процессором Amd Ryzen 5 2600x в операционной системе Ubuntu 18.04.4. Для обучения потребовалось 35 эпох.

5.3 Обучение рекуррентной нейронной сети

Обучение производилось на сервере с операционной системой Ubuntu 18.04.4, процессором Intel Xeon Gold 6151 и видеокартой Nvidia Tesla V100(16GB). Для тренировочной выборки брались 400 случайных файлов из каждого класса. Одна эпоха занимала 1 час 20 минут. Для обучения потребовалось 25 эпох.

Заключение

Основные результаты данной работы заключаются в следующем:

1. Была исследована применимость подходов[2, 3] по распознаванию семантики кода, использующие в своей основе машинное обучение, были выявлены плюсы и минусы каждого подхода. Было выявлено, что первый подход не применим в рассматриваемой задаче из-за того, что в качестве признаков в том числе он использует названия аргументов и переменных функции.
2. Адаптирован inst2vec подход для классификации функции в скомпилированном бинарном файле.
3. Данный подход реализован практически в виде утилиты для Ida Pro и набора скриптов для обучения весов рекуррентной нейронной сети https://github.com/vladimirtelepov/course-work

В целом можно отметить применимость данного подхода для анализа бинарных файлов. Полученная точность в 68% на самом деле является совокупной точностью по всем классам, но если сранивать наличие нужного класса в первых трёх (а именно столько классов с большой вероятностью будет учитываться человеком при изучении результатов работы) то точность будет больше.

В качестве направления дальнейших исследований предлагается дальнейшая адаптация подхода inst2vec для распознавания классификации функций в скомпилированном бинарном файле, в частности можно выделить следующие задачи:

1. Изменение задачи классификации на multi-label классификацию. Это связано с тем, что из-за, например, инлайнинга одна функция в скомпилированном файле может принадлежать сразу к нескольким классам. Для решения этой задачи при обучении необходимо предсказывать принадлежность функции сразу нескольким классам.
2. Использование междпроцедурного анализа для улучшения точности.
3. Увеличение набора тестовых данных.

Список литературы

[1] *POJ-104 dataset.*

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[3] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3588–3600. Curran Associates, Inc., 2018.

[4] Dawson Engler. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Advances in Neural Information Processing Systems 31*. ACM, 2001.