Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б10-мм

Рекомпиляция с помощью Ghidra: заголовочные файлы

Сарапулов Василий Алексеевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
старший преподаватель кафедры информационно-аналитических систем, К. К. Смирнов

Санкт-Петербург
2024

Оглавление

Введение

В последнее время исследование и анализ исполняемых файлов становятся все более актуальными в области информационной безопасности и обратного проектирования программного обеспечения. Ghidra — это фреймворк для обратного инжиниринга программного обеспечения, включающий набор инструментов для анализа скомпилированного кода. Декомпиляция является важной частью этого процесса, поскольку позволяет преобразовать исполняемые файлы в человекочитаемый вид, что упрощает анализ. Помимо Ghidra существует множество других программ для обратной разработки, например, IDA. Ее функциональность шире, но большая ее часть доступна только в версии pro.

Рекомпиляция отличается от декомпиляции тем, что полученный код может быть повторно скомпилирован, и полученный исполняемый файл будет делать то же самое, что и исходный. Рекомпиляция необходима для поднятия исходного кода программ, скомпилированных под одну архитектуру, и последующей компиляции на процессорах другой архитектуры. Поднятие кода — перевод кода, скомпилированного под конкретную архитектуру, в код высокоуровневого языка. Также рекомпиляция позволяет внести изменения в исходный исполняемый файл и заново его скомпилировать.

Хотя Ghidra предоставляет инструменты для декомпиляции, анализа исполняемого файла и обратного инжиниринга, полученный декомпилированный код не всегда может быть повторно скомпилирован. И так как Ghidra предполагается как инструмент для анализа исполняемых файлов, а не для рекомпиляции, разработчики не ведут работу в этом направлении. Поэтому в полученном декомпилированном коде некоторые сложные синтаксические конструкции заменяются на более простые человекочитаемые, но не соответствующие стандартам языка C.

Таким образом, для успешной рекомпиляции необходимо внести изменения и доработки в существующие методы анализа исполняемых файлов. Ранее в рамках летней школы был разработан постобработчик

декомпилированного кода, который разрешал особенности полученного кода. Целью данной учебной практики является разработка и внедрение новых инструментов, которые помогут улучшить методы анализа, а также коррекция текущих инструментов для обеспечения возможности успешной рекомпиляции декомпилированного кода.

1. Постановка задачи

Целью работы является улучшение и создание новых инструментов декомпиляции Ghidra, для достижения рекомпиляции. Для её выполнения были поставлены следующие задачи:

1. Проанализировать особенности декомпилированного кода.
2. Спроектировать и реализовать недостающие инструменты декомпиляции, исправить существующие.
3. Выполнить замеры количества ошибок компиляции в исходном варианте и после внесения изменений.

2. Обзор

2.1. Обзор Ghidra

Ghidra — фреймворк для обратного инжиниринга программного обеспечения, включающий в себя набор инструментов анализа программного обеспечения, которые позволяют анализировать код, скомпилированный на различных платформах. Для автоматического анализа Ghidra использует набор анализаторов, обрабатывающих бинарный файл.

2.1.1. Декомпилятор

Основным анализатором является декомпилятор, который поднимает двоичный код в язык ассемблера, затем в pcode и в язык C. Для этого используется библиотека Decompiler Analysis Engine. Дизассемблинг языков машинного кода и последующая трансляция в pcode составляют основную подсистему декомпилятора. Библиотека представляет язык передачи регистров — pcode и инструмент SLEIGH.

Pcode — это язык передачи регистров, предназначенный для реверс-инжиниринга. Язык обладает достаточной универсальностью, чтобы моделировать поведение множества различных процессоров. В основе работы pcode лежит преобразование отдельных инструкций процессора в последовательность операций. Набор операций pcode представляет собой набор арифметических и логических действий, выполняемых процессорами общего назначения.

SLEIGH — язык спецификации процессоров и набор инструментов для генерации кода ассемблера и pcode. SLEIGH основан на языке спецификации SLED[7] (Specification Language for Encoding and Decoding) и расширяет его, предоставляя семантические описания машинных инструкций и другие усовершенствования для реверс-инжиниринга.

Основываясь на файле спецификации процессора, SLEIGH сопоставляет машинные инструкции в двоичной кодировке с последовательностями операций pcode. Операции pcode генерируются для одной машин-

ной инструкции по определенному адресу. Также по pcode отслеживается поток управления.

На основе pcode генерируются базовые блоки и отслеживается их граф потока управления. Запускается основной цикл упрощения, в котором генерируются переменные в форме SSA, устраняется мертвый код, выводится информация о типах переменных, основываясь на инструкциях, в которых они используются, переписываются выражения, настраивается граф потока управления и восстанавливаются структуры потока управления. На этапе переписывания выражений происходят наибольшие изменения дерева синтаксического разбора. В соответствии со списком правил определяется последовательность операций редактирования дерева. При восстановлении структуры потока управления декомпилятор восстанавливает высокоуровневые объекты, такие как циклы, блоки if/else и операторы switch.

После завершения основного цикла упрощения выполняется окончательное преобразование pcode, так как во время цикла многие операции нормализуются особым образом для процесса переписывания выражений. Выводятся высокоуровневые переменные, путем преобразования из формы SSA и слияния низкоуровневых переменных, выбираются их имена на основе таблицы символов и их низкоуровневых элементов. Добавляются преобразования типов, чтобы конечный результат был синтаксически корректным. Выполняется финальное преобразование потока управления: упорядочиваются компоненты, определяются, какие неструктурированные переходы являются break, устанавливаются метки для оставшихся неструктурированных переходов. После восстановления прототипа функции, структуры потока управления и выражений, генерируются финальные C-токены. Большинство лексем аннотируются адресом машинной инструкции, с которой они наиболее тесно связаны. Токены проходят через стандартный алгоритм Oppen pretty-printing[6] для определения окончательных переносов строк и отступов.

2.1.2. Анализатор типов данных

Важным для рекомпиляции является анализатор типов данных. С его помощью Ghidra восстанавливает высокоуровневые типы данных языка C и анализирует сигнатуры функций. Для его работы необходимы архивы типов данных (файлы с расширением .gdt).

2.1.3. C Parser

Ghidra имеет инструмент C Parser для создания архивов типов данных. C Parser принимает на вход список заголовочных файлов, директории их поиска и параметры парсинга. Для обработки заголовочного файла используются два парсера: парсер директив препроцессора и парсер кода на языке C. Парсер препроцессора сохраняет макроопределения в архив и выполняет макроподстановки. Также он обрабатывает подключения других заголовочных файлов в исходном. Парсер C сохраняет в архив сигнатуры функций, структуры и определения типов данных.

2.2. Обзор существующих решений

2.2.1. IDA

Самым известным инструментом обратной разработки является IDA (Interactive DisAssembler). Дизассемблер IDA поддерживает более 60 процессоров: ARM, MIPS, RISC-V, x86/x64 и другие[5]. Также имеется поддержка большого количества видов исполняемых файлов: PE, ELF, Mach-O и другие[4]. Для декомпиляции IDA использует разные декомпиляторы для разных архитектур. Но большая часть функциональности доступна только в версии про. IDA FREE предоставляет дизассемблер и декомпилятор для архитектур x86/x64. Также в бесплатной версии отсутствует возможность экспорта декомпилированного кода на языке C. Поэтому невозможно полностью оценить возможность рекомпиляции декомпилированного кода.

2.2.2. Binary Ninja

Binary Ninja — это интерактивный декомпилятор, дизассемблер, отладчик и платформа для анализа бинарных файлов. Binary Ninja поддерживает дизассемблирование файлов различных архитектур, включая x86/64, ARM, PowerPC, MIPS, RISC-V, MSP430, TriCore, C-SKY и другие. Имеется поддержка основных исполняемых файлов (PE, Mach-O, ELF). При декомпиляции Binary Ninja использует язык промежуточного представления BNIL, который позволяет просмотреть код на разных этапах декомпиляции. Также Binary Ninja умеет компилировать декомпилированный код на C, некоторых архитектур[3], используя встроенный компилятор SCC[2]. Но разработчики не гарантируют полную рекомпиляцию декомпилированного кода[1]. Так же как и в IDA в Binary Ninja Free доступна только ограниченная часть функциональности: дизассемблинг и декомпиляция на некоторых архитектурах, просмотр кода на языке IL доступен только в высокоуровневом представлении, компиляция с помощью SCC недоступна.

2.3. Особенности декомпилированного кода

2.3.1. Неопределенные типы данных

При декомпиляции, если Ghidra не может сопоставить высокоуровневый тип данных переменной, то создает тип данных вида undefined<размер в байтах>. Также в процессе декомпиляции создаются другие типы данных. Но при экспорте декомпилированного кода созданные типы данных не определяются.

2.3.2. Защита стека

При компиляции по умолчанию в компиляторе включена опция защиты стека (в gcc для ее отключения используется флаг -fno-stack-protector). Таким образом, при декомпиляции добавляется код защиты стека, вызывающий функцию runtime.

Листинг 1: Пример декомпиляции функции с защитой стека

```
undefined8 main(void)
{
  long in_FS_OFFSET;
  long local_10;
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  ...
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return 0;
}
```

2.3.3. Служебные функции

Служебные функции — функции, такие как plt и runtime, добавляемые транслятором для выполнения своих задач. Так как данные функции добавляются компилятором, в декомпилированном коде они не нужны.

2.3.4. Некорректные выражения с приведениями типов

Декомпилятор, после присвоения имен переменным, добавляет приведения типов, но иногда некорректные. Например: `*pauVar1 = (undefined [16])0x0;`

2.3.5. Внутренние функции декомпилятора

В силу использования языка передачи регистров pcode, в декомпилированном коде используются функции такие как ZEXT, SUB, CONCAT. Но декомпилятор их не создает.

2.3.6. Некорректные обращения к массивам

Для доступа к элементам массива декомпилятор генерирует конструкцию вида ._<отступ в байтах>_<количество байт>_. Такая кон-

струкция используется для упрощения ручного анализа кода, но синтаксически некорректна.

Листинг 2: Пример некорректного обращения к массиву в деомилированном коде

```
undefined auVar5 [16];
auVar5._8_8_ = 0;
auVar5._0_8_ = pbVar3;
```

2.3.7. Отсутствие выгрузки глобальных переменных

В экспортированном коде на языке C могут использоваться глобальные переменные, но Ghidra их не выгружает.

2.3.8. Отсутствие выгрузки заголовочных файлов, использующихся в декомпилированном коде

Анализатор типов данных сопоставляет типы данных в декомпилированном коде с типами данных из архивов типов данных. Но в экспортированном коде на языке C заголовочные файлы, содержащие используемые типы данных, не подключаются.

3. Реализация

Работа по данной теме началась в рамках летней проектной школы в команде с Вячеславом Кочергиным. Были изучены особенности декомпилированного кода с помощью Ghidra и создан постобработчик[1], исправляющий их. В данной работе было принято решение создать форк от основного репозитория и уже в нем вносить изменения, исправляющие выявленные затруднения.

Одной из важных проблем для рекомпиляции является автоматическое добавление заголовочных файлов при экспорте, так как в декомпилированном коде используются функции и типы данных из них. При экспорте можно выбрать опцию имитации типов данных, но используемые функции из заголовочных файлов не будут выгружены.

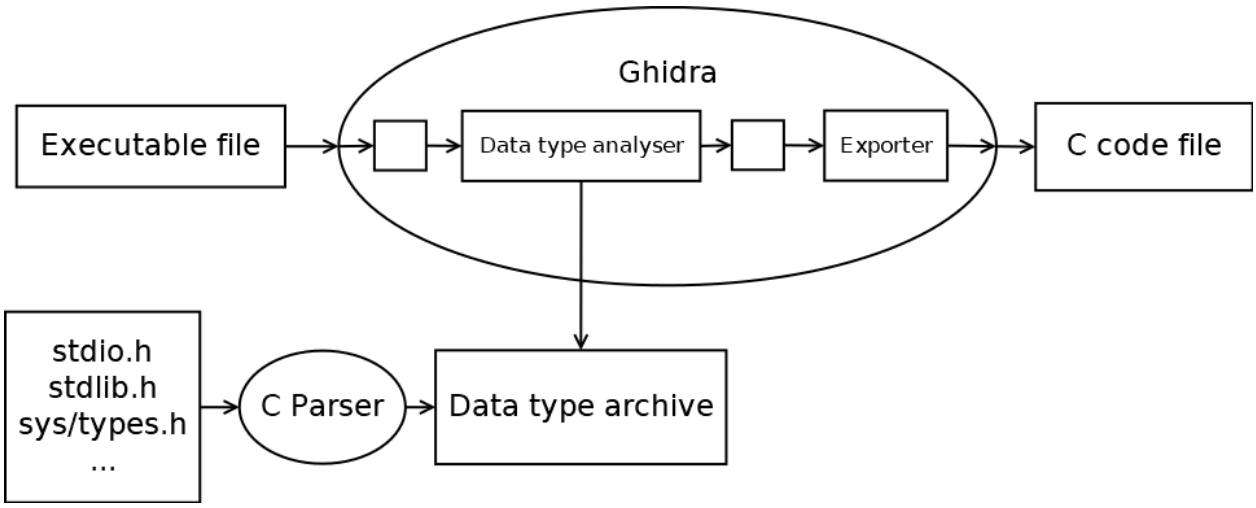

Рис. 1: Архитектура Ghidra: анализатор типов данных

В класс экспортера декомпилированного кода на языке C был добавлен метод для выгрузки заголовочных файлов. При тестировании этого метода оказалось, что в стандартных архивах типов данных у типов данных не указаны пути к их заголовочным файлам. То есть, например, заголовочный файл types.h, находящийся в каталоге sys, в архиве сохранен как types.h, а не sys/types.h. Таким образом, при экспорте кода и его компиляции транслятор не может найти заголовочный файл в стандартных директориях. Поэтому возникла необходимость создания

[1] https://github.com/VyacheslavIurevich/recompilation_postprocessor

нового архива типов данных с исправленными путями к заголовочным файлам.

В C Parser был исправлен способ присвоения путей заголовочных файлов типам данных. Но при создании нового архива типов данных были обнаружены и исправлены следующие ошибки. На этапе парсинга директив препроцессора отсутствовала поддержка символьных литералов, и при обработке файла bits/wchar.h возникала ошибка. Данная проблема описана в issue в официальном репозитории[2]. Также парсером препроцессора неправильно выполнялись макроподстановки, и парсер кода выдавал синтаксическую ошибку.

После исправления данных ошибок оказалось, что в силу принципа работы анализатора типов данных, при экспорте добавляются неправильные заголовочные файлы. Например, если при декомпиляции была обнаружена функция fopen, возвращающая указатель на FILE, то при экспорте будет добавлен заголовочный файл bits/types/FILE.h, а не stdio.h. В стандартном архиве типов данных generic_clib_64 данной проблемы нет, так как тип данных FILE в нем относится к заголовочному файлу stdio.h. Таким образом, алгоритм добавления типов данных в создаваемый архив был переделан. В результате был создан архив типов данных, включающий заголовочные файлы из glibc. Также в анализатор типов данных была добавлена выгрузка сигнатур распознанных функций в локальный архив типов данных декомпилируемой программы.

[2] https://github.com/NationalSecurityAgency/ghidra/issues/4740

4. Эксперимент

4.1. Условия эксперимента

Для проведения эксперимента был собран набор тестовых программ из домашних заданий по программированию на языке C, бенчмарков и системных утилит linux:

- hello_world. Программа, выводящая ”hello World!”.
- bmp[3]. Программа чтения заголовков bmp файлов.
- avl. Программа, реализующая авл дерево.
- linpack[4]. Бенчмарк для измерения вычислительной производительности компьютеров при обработке чисел с плавающей запятой.
- dhrystone[5]. Бенчмарк для измерения целочисленной производительности компьютерных систем.
- echo. Системная утилита linux для вывода текста.
- cat. Системная утилита linux для вывода содержимого файла.

Системные утилиты были взяты из Ubuntu 24.04. Компиляция проводилась с помощью компилятора gcc версии 14.2.0.

4.2. Исследовательские вопросы

Основной задачей данного эксперимента является проверка уменьшения количества ошибок компиляции при компиляции декомпилированного кода с автоматической выгрузкой заголовочных файлов. Для сравнения результатов экспорт кода проводился трижды: без опций имитации типов данных и подключения заголовочных файлов, с опцией имитации типов данных и с опцией выгрузки заголовочных файлов.

[3] https://github.com/Sarapulov-Vas/BMP
[4] https://github.com/ereyes01/linpack
[5] https://github.com/sifive/benchmark-dhrystone

Таблица 1: Сравнение количества ошибок компиляции при экспорте кода с разными опциями.

Программа	Без дополнительных опций экспорта	С имитацией типов данных	С подключением заголовочных файлов
hello_world	11	9	12
bmp	64	43	25
avl	57	29	54
linpack	93	30	84
dhrystone	56	56	54
echo	416	295	427
cat	582	384	545

4.3. Результаты

Результаты замеров приведены в таблице 1. По ним можно сделать вывод, что на большинстве тестовых программ при подключении заголовочных файлов происходило уменьшение количества ошибок компиляции, но на некоторых произошло небольшое увеличение. Почти во всех тестах можно заметить увеличение ошибок компиляции с подключением заголовочных файлов по сравнению с тестовыми запусками с имитацией типов данных. Это связано с тем, что в декомпилированном коде присутствуют нестандартные типы данных, которые генерируются при декомпиляции и выгружаются только с опцией имитации типов данных. Также при подключении заголовочных файлов появляются ошибки из-за конфликта функций, содержащихся в них, и декомпилированных plt функций. Если после экспорта исключить служебные функции (автоматизацией исключения служебных функций занимается Вячеслав Кочергин в рамках семестровой практики) и добавить генерируемые типы данных, то результат значительно улучшится. Например, если применить вышеописанные изменения к hello_world, то компиляция пройдет без ошибок. Если эти изменения применить к linpack, то возникает 20 ошибок компиляции.

Заключение

В рамках данной учебной практики были достигнуты следующие результаты:

- Выполнен обзор особенностей декомпилированного кода.
- Реализован новый инструмент декомпиляции, исправлены существующие:
  - Реализована опция подключения заголовочных файлов при экспорте.
  - В C Parser добавлена поддержка символьных литералов препроцессором, исправлен алгоритм выполнения макроподстановок, исправлен алгоритм присвоения типам данных путей к заголовочным файлам.
- Выполнены замеры количества ошибок компиляции.

Внесенные изменения расположены в пул-реквесте: https://github.com/VyacheslavIurevich/recompilation-ghidra/pull/2

Список литературы

[1] Binary Ninja Pseudo C. — URL: https://docs.binary.ninja/guide/index.html#pseudo-c (дата обращения: 17 ноября 2024 г.).

[2] Binary Ninja Shellcode Compiler. — URL: https://scc.binary.ninja/ (дата обращения: 17 ноября 2024 г.).

[3] Binary Ninja supported processors. — URL: https://binary.ninja/faq/#supported-architectures (дата обращения: 17 ноября 2024 г.).

[4] IDA supported file formats. — URL: hhttps://docs.hex-rays.com/user-guide/disassembler/supported-file-formats (дата обращения: 17 ноября 2024 г.).

[5] IDA supported processors. — URL: https://docs.hex-rays.com/user-guide/disassembler/supported-processors (дата обращения: 17 ноября 2024 г.).

[6] Oppen Dereck C. Pretty printing // ACM Trans. Program. Lang. Syst. — 1980. — . — Vol. 2, no. 4. — P. 465–483. — URL: https://doi.org/10.1145/357114.357115.

[7] Ramsey Norman, Fernández Mary F. Specifying representations of machine instructions // ACM Trans. Program. Lang. Syst. — 1997. — . — Vol. 19, no. 3. — P. 492–524. — URL: https://doi.org/10.1145/256167.256225.