

Санкт-Петербургский государственный университет

Минаяев Роман Русланович

Выпускная квалификационная работа

Разработка декомпилятора для
сопровождения продуктов с утерянным
ИСХОДНЫМ КОДОМ

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование
информационных систем»*

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

Научный руководитель:
старший преподаватель кафедры системного программирования, к.т.н., Ю.В. Литвинов

Рецензент:
эксперт по разработке программного обеспечения, А.Ю. Стариковский

Санкт-Петербург
2025

Saint Petersburg State University

Roman Minyazev

Bachelor's Thesis

Development of a decompiler to support products with lost source code

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2021 «Programming Technologies»*

Scientific supervisor:
C.Sc., senior lecturer Y. V. Litvinov

Reviewer:
Software development expert at "YADRO", A.Y. Starikovskiy

Saint Petersburg
2025

Оглавление

Введение	4
1. Постановка задачи	5
2. Обзор	6
2.1. Ghidra	6
2.2. Критерии для отбора инструментов	8
2.3. Использование Ghidra для реализации инструмента . . .	8
2.4. Инструменты для анализа C и C++ кода	9
2.5. Методы и структуры, предоставляемые Clang	10
3. Реализация	13
3.1. Релизованные проверки	13
3.2. Сворачивание развернутых макросов	13
3.3. Алгоритм сворачивания развернутых макросов	14
3.4. Тестирование	18
Заключение	19
Список литературы	20

Введение

Процесс сопровождения продуктов с утерянным исходным кодом зачастую включает в себя обратную разработку программного обеспечения. Для решения этой проблемы существует целый пласт технологий, часть из которых относится к декомпиляторам. Получаемый декомпилирующей исходный код, если и корректен, зачастую достаточно сложен в дальнейшем сопровождении и требует доработки.

Дополнительная информация об исследуемой системе помогает улучшать качество результата декомпиляции. Несмотря на то что современные декомпиляторы поддерживают подобный подход, остаётся потенциал для улучшения результата применением специализированных эвристик, основанных на уникальных знаниях об исходной системе, получаемых из различных источников.

Опираясь на частично доступный исходный код и знания о системных библиотеках, а также используя дополнительную информацию об оригинальной целевой платформе, для которой предназначалась изучаемая система, можно значительно улучшить код на выходе декомпилятора. Например, для интересующих нас систем частично доступны даже исходные коды актуальных или хотя бы прежних версий некоторых процедур, структур данных и макросов. Однако этим не может воспользоваться применённый в проекте декомпилятор.

Улучшение конкретного использованного декомпилятора — одно из возможных направлений исследования. Но по ряду причин, которые обсуждаются далее, особый интерес представляет другой подход, основанный на идее автоматизации преобразований исходного кода, уже полученного в результате работы декомпилятора.

Данная работа посвящена разработке соответствующего инструмента.

1. Постановка задачи

Целью работы является реализация инструмента для автоматизации процесса сопровождения продуктов с утерянным исходным кодом путем поддержки пользовательских авторефакторингов над кодом, сгенерированным декомпилятором.

Для её выполнения были поставлены следующие задачи.

1. Провести обзор существующих решений и технологий.
2. Реализовать детекторы для ситуаций, не соответствующих заданным требованиям заказчика.
3. Реализовать авторефакторинги кода для ситуаций, не соответствующих заданным требованиям заказчика.
4. Протестировать реализованную функциональность.

2. Обзор

В данном разделе будут рассмотрены инструменты и технологии, используемые в предметной области.

2.1. Ghidra

Ghidra¹ — декомпилятор с открытым исходным кодом, используемый в целях обратной разработки в интересующем нас проекте. Ghidra использует механизм SLEIGH² для генерации промежуточного кода (p-code) из машинных инструкций по файлу спецификации процессора, впоследствии проводя различные преобразования на уровне этого промежуточного кода (например, построение SSA формы и устранение мертвого кода) и дальнейшую генерацию кода на языке программирования C.

Часть задач, решаемых декомпилятором, требует дополнительную информацию об устройстве исследуемой системы. Например, вывод типов и их дальнейшее распространение по синтаксическим деревьям, основанный на предоставлении пользователем сигнатур функций или прямым указанием типа данных для переменной. Более того, для получения качественных результатов решений некоторого класса задач, возникает необходимость применения специфических знаний об исследуемой системе.

Типичным примером является поиск встроенных (inline expansion) вызовов функций в декомпилированном коде. Общие алгоритмы для распознавания, как правило, вообще не используют эвристики, основанные на семантике конкретных процедур, или известные им процедуры ограничены системными библиотеками языка программирования. Так, в работе FSmell: Recognizing Inline Function in Binary Code [5] рассматриваются проблемы распознавания встроенных вызовов функций в общем виде. Несмотря на прогресс данной работы относительно

¹Репозиторий Ghidra, URL: <https://github.com/NationalSecurityAgency/ghidra> (дата обращения: 24.05.2025)

²Документация SLEIGH, URL: https://ghidra.re/ghidra_docs/languages/html/sleigh.html (дата обращения: 24.05.2025)

других, полнота (recall) данного решения составляет 39.2%, а точность (precision) составляет 67.5% (иначе говоря, количество ложноположительных срабатываний составляет 32.5%).

Код, сгенерированный декомпилятором, проходит несколько итераций ручных рефакторингов вне декомпилятора (в сторонних средах для разработки программного обеспечения) и сопровождается командой разработчиков. Процесс обратной разработки программного обеспечения показан на рис 1.

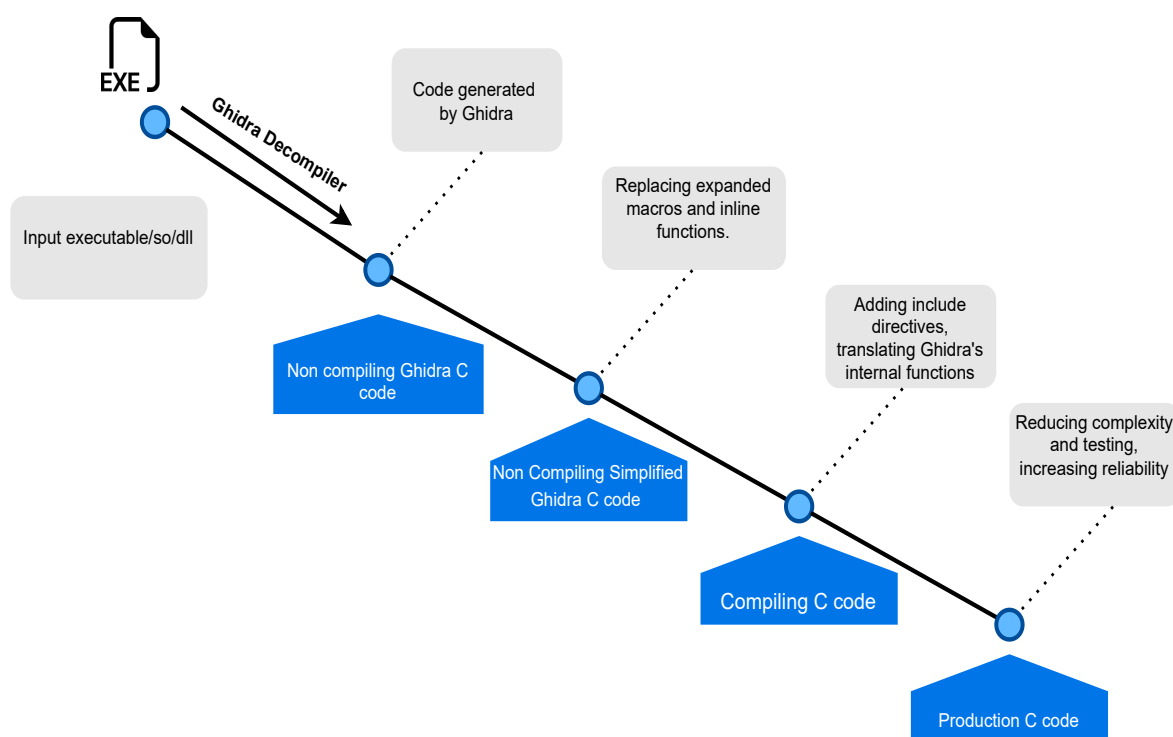


Рис. 1: Процесс обратной разработки программного обеспечения

Стоит отметить, что на основании знаний об исследуемой системе и опыта, полученного при использовании декомпилятора, часть из этих ручных изменений кода можно автоматизировать, обеспечив:

- надежность пользовательских преобразований кода;

- снижение стоимости и сложности разработки программного обеспечения.

Для решения вышеперечисленных проблем необходимо спроектировать и реализовать инструмент, позволяющий автоматизировать процесс применения пользовательских изменений декомпилированного кода для обратной разработки конкретной библиотеки.

2.2. Критерии для отбора инструментов

Для реализации подобного инструмента необходимо иметь программную систему, удовлетворяющую следующим требованиям:

- возможность интеграции в существующий рабочий процесс (мы не рассматриваем исключение из него использование декомпилятора Ghidra, на котором основан текущий рабочий процесс);
- наличие инфраструктуры для анализа кода;
- наличие инфраструктуры для изменения кода;
- расширяемость и плагинизируемость системы для добавления анализаторов, специфичных для исследуемой системы;
- наличие интерактивной системы для анализаторов, подразумевающих наличие ложноположительных срабатываний;
- возможность параметризации анализаторов для сохранения устойчивости при выходе новых версий исследуемой системы;
- возможность интеграции в различные среды для разработки программного обеспечения.

2.3. Использование Ghidra для реализации инструмента

Наиболее интуитивным решением было бы использовать инфраструктуру Ghidra и писать анализаторы на ее промежуточном коде (p-code)

до генерации кода. Однако Ghidra не соответствует большей части вышеперечисленных критериев (за исключением интеграции в различные среды разработки). Если реализация плагинизируемой системы рефакторингов в декомпиляторе является вполне осуществимой задачей, то реализация необходимых для анализа алгоритмов промежуточного кода декомпилятора становится нетривиальной задачей, не говоря о итоговой реализации самих детекторов. Так, в работе «A Formal Semantics for P-Code» [6] рассматриваются проблемы промежуточного кода декомпилятора (например, неоднозначная семантика промежуточного кода декомпилятора с дальнейшим построением формальной семантики на основании экспериментов).

2.4. Инструменты для анализа C и C++ кода

В рамках обзора стоит выделить статические анализаторы общего назначения, которые используются для форматирования и/или проверки соответствия кода определенным стандартам кодирования, например:

- MISRA (Motor Industry Software Reliability Association) C/C++ [7];
- AUTOSAR (Automotive Open System Architecture) [1];
- SEI CERT C/C++ [3].

Неполный список существующих статических анализаторов, реализующих проверки из этих стандартов: Astree, ECLAIR, Axivion Bauhaus Suite, CodeSonar, Helix QAC, Klocwork, LDRA tool suite, PVS-Studio³.

Все эти статические анализаторы предоставляются по коммерческим лицензиям, что накладывает явные ограничения на их использование, тестирование и расширение функциональности для анализа исследуемой системы.

В диссертации А. А. Белеванцева рассмотрена часть из этих инструментов и подходы, используемые в них для анализа кода [2].

³PVS-Studio, URL: <https://pvs-studio.ru/> (дата обращения: 24.05.2025)

Для реализации инструмента было решено использовать Clang, который является фронтендом языков программирования C и C++ для компиляторной инфраструктуры LLVM, а также инструменты из пакета clang-tools-extra, в состав которого входят модули для анализа исходного кода.

Рассмотрим инструменты Clang-Tidy⁴ и Clang Static Analyzer из clang-tools-extra.

Clang-Tidy и Clang Static Analyzer позволяют производить анализ кода на уровне препроцессора, абстрактного синтаксического дерева, предоставляют инфраструктуру для базового анализа графа потока управления, модули для анализа потока данных с абстрактной интерпретацией, SAT решатели, инфраструктуру для анализа методом символического исполнения.

Более того, Clang-Tidy имеет отдельный модуль для изменения исходного кода, интерактивную систему рефакторинга, а также интеграцию в большинство популярных сред для разработки исходного кода.

Например, в статье [4] описывается процесс разработки анализатора Svace в институте системного программирования им. В. П. Иванникова РАН на основе Clang-Tidy и Clang Static Analyzer, который реализует 95% проверок MISRA C 2012.

2.5. Методы и структуры, предоставляемые Clang

В данном разделе описаны методы статического анализа кода, предоставляемые Clang и подходящие для реализации определенных эвристик.

2.5.1. Clang AST Matcher и Clang AST Visitor

Часть из накладываемых эвристик может быть проверена с помощью обхода абстрактного синтаксического дерева при помощи паттерна Visitor. Также, Clang предоставляет механизм (декларативный язык

⁴Документация Clang-Tidy, URL: <https://clang.llvm.org/extra/clang-tidy/> (дата обращения: 24.05.2025)

для сопоставления узлов абстрактного синтаксического дерева), называемый Clang AST Matcher. Простой пример кода, использующий Clang AST Matcher:

```
1 Finder->addMatcher(  
2     functionDecl(isDefinition(),  
3         unless(anyOf(isDefaulted(), isDeleted(), isWeak()))  
4         .bind('functionDecl'),  
5     this);
```

Данный код нужен для того, чтобы в заранее зарегистрированную функцию обратного вызова отправлять узлы AST, которые являются определением функции, соответствующее данным требованиям:

- функция не является функцией по умолчанию (`func() = default`);
- не является слабым символом (`void func() __attribute__((weak))`);
- объявление функции не является удаленной функцией (`void func() = delete`).

2.5.2. Clang Control Flow Graph

Часть эвристик может быть проверена с помощью наложения эвристик на граф потока управления. Некоторые модули:

- `clang::CFGReverseBlockReachabilityAnalysis`, анализ достижимости вершин графа потока управления;
- `clang::CFGDominatorTreeImpl`, деревья доминаторов и постдоминаторов;
- `clang::LiveVariables`, анализ времени жизни переменных.

2.5.3. Остальные методы

- Clang dataflow analysis
 - Часть из накладываемых эвристик может быть проверена путем анализа потока данных. В Clang анализ потока данных основан на решетках (lattice) и реализуется путем предоставления монотонных передаточных (transfer) функций для вычисления абстрактных состояний после выполнения инструкций clang::CFGElement (представление инструкции в расширенном графе потока управления данных clang::AdornedCFG) и join/widen операторов (clang::dataflow::DataflowAnalysis);
- SAT-решатель
 - clang::dataflow::WatchedLiteralsSolver, реализация DPLL.

3. Реализация

На текущий момент реализация подразумевает под собой расширение функциональности Clang-Tidy внедрением требуемых проверок.

3.1. Релизованные проверки

В ходе работы было реализовано несколько требуемых детекторов, часть из которых является параметризованными по нескольким причинам:

- возможность удаления/добавления определенных эвристик для сохранения устойчивости системы при выходе новых версий исследуемой системы (с возможным увеличением ложноположительных срабатываний);
- возможность управления процессом рефакторинга установлением определенных параметров.

3.2. Сворачивание развернутых макросов

Опишем процесс реализации на наиболее актуальной из поставленных задач. Одной из задач, рассматриваемых в ходе работы была автоматизация сворачивания раскрытых макросов, имеющих доступное определение:

- раскрытые макросы могут занимать больше половины сгенерированного исходного кода;
- сложный для рефакторинга человеком код, получаемый в ходе генерации
 - значительное увеличение уровня вложенности кода;
 - в ходе оптимизаций компилятора, декомпилятора и последующей кодогенерации раскрытый макрос может быть ”размазан” по всему файлу с множеством переходов на его отдельные части;

- сложность сгенерированного кода из-за раскрытых макросов во встроенных функциях.

3.3. Алгоритм сворачивания развернутых макросов

Стоит отметить, что задача по восстановлению конкретного макроса может быть неразрешимой без дополнительного контекста. Рассмотрим макрос:

```
// macro.h
#define MACRO(x) \
    if (x < 0) { \
        do_negative(x); \
    } else { \
        do_positive(x); \
    }
// macro.cpp
#include macro.h
static constexpr int x = 1;
MACRO(X)
```

В ходе компиляции при помощи Clang, на уровне LLVM IR будет построено SSA представление, на котором при помощи абстрактной интерпретации будет проведено распространение условных констант (SCCP), вследствие чего из-за недостижимости части раскрытого макроса для него не будет произведено последующей кодогенерации.

Пример декомпиляции для этого кода:

```
// compiled with clang macro.cpp -Werror -Wpedantic -O2
do_positive(1);
```

Однако если мы имеем дополнительную информацию о процедуре `do_positive`, например, что она не используется вне заголовочного

файла и контекст ее вызова для этого макроса отличается от контекста остальных вызовов в этом заголовочном файле, то при анализе этой версии библиотеки, мы можем сделать вывод о том, что этот код был частью MACRO(1).

Основными проблемами рефакторинга раскрытых макросов являются:

- Адресное пространство инструкций раскрытого макроса может быть прерывистым вследствие оптимизаций, примененных компилятором, что усложняет распознавание границ макроса
- Наличие инструкций в высокоуровневной структуре управления макроса, не относящихся к представлению макроса вследствие агрессивных оптимизаций, примененных компилятором (например, Code Motion)

Подход, применяемый в данной работе, заключается в наложении достаточно строгих эвристик на граф потока управления, который будет кандидатом на раскрытый макрос, а также дальнейшее распознавание его границ, основанных на семантике самого макроса и анализе его высокоуровневой структуры управления.

Рассмотрим алгоритм на примере одного из макросов:

- обход синтаксического дерева при помощи паттерна посетитель (`clang::RecursiveASTVisitor`) для сбора поддеревьев с полной структурой управления (под этим подразумевается наличие полной структуры оператора управления `if`);
- наложение эвристик на узел оператора `if` путем обхода абстрактного синтаксического дерева или сопоставления узлов при помощи `AST Matching`;
- построение графа потока управления по полученному поддереву;
- построение дерева доминаторов и постдоминаторов по полученному графу потока управления;

- проверка эвристик на графе потока управления при помощи доступного инструментария:

- данный подграф должен иметь одну точку входа;
- подграф, получаемый исследованием истинной ветки оператора `if` должен иметь одну точку выхода (формально, любой подграф будет иметь одну точку выхода, являющуюся общим выходным блоком графа управления объемлющей процедуры, имеется в виду один из предшественников этого выходного блока);

$$\exists E \in \text{preds}(E_e) : \text{PostDomTree.dominates}(E, E_1)$$

E_e — выходной блок графа потока управления

E_0 — входной блок графа потока управления

E_1 — successor-блок E_0

- анализ графа потока управления:
 - путем экспериментов выявляются значимые инструкции макроса (условные переходы, вызов конкретных функций, присваивание переменным конкретных значений), что актуально для декомпилированного кода, содержащего множество артефактов декомпиляции;
 - на основании выделенных инструкций проводится проверка существования основных путей выполнения у графа (например, должен быть путь из начальной вершины, который содержит инструкцию условного перехода с выражением X , последующий вызов функции Y и инструкцию `return`);
 - сбор информации для восстановления параметров для генерации кода с использованием макроса;
- использование дерева доминаторов для удаления блоков, относящихся к макросу (проблема использования общих базовых блоков

несколькими раскрытыми макросами). Те базовые блоки раскрытого макроса, что не были удалены по причине использования другими макросами путем безусловного перехода на них (`goto`), становятся недостижимыми из точки подстановки обнаруженного макроса (это связано с семантикой макроса). Для макросов другого вида может генерироваться дополнительная метка и оператор `goto` на последующие инструкции для сохранения семантики исходного кода.

Пример кода, сгенерированный Ghidra, который нуждается в рефакторинге:

```
1 uVar7 = execute_anything();
2 if (uVar7 != 0) {
3     if (DATA[1] < 6) {
4         uStack_e40 = (_T *)0x7e59d34;
5         uVar38 = transfer(CONST_ENUM_VALUE);
6         uVar20 = 0x53e8;
7         uVar38 = uVar38 & 0xff;
8         psStack_e48 = (Y *)0xda1dd50;
9         uStack_e40 = (_T *)pcVar41;
10        goto LAB_ADDRESS1; // +6 lines
11    }
12    goto LAB_ADDRESS2; // +12 lines;
13 }
```

Желаемый вид:

```
1 // run: clang-tidy --checks=expanded-macro-name-refactor
2 //--extra-arg='-includeheader_with_macro_define.h' --fix ./source.c
3 #include "header_with_macro_define.h"
4 ...
5 uVar7 = execute_anything();
6 // From the source code lines: lineX-lineY
7 if(MACRO1(uVar7)) {
8     // From the source code lines: lineX-lineY
9     MACRO2(CONST_ENUM_VALUE, ..., uVar7);
10 }
11
```

3.4. Тестирование

Для тестирования реализованной функциональности использовалась внутренняя инфраструктура Clang (`clang::tidy::test`) и фреймворк для модульного тестирования Google Test. Были написаны тесты, идентичные принятым в исходном проекте и проверяющие эквивалентность заранее подготовленных исходных файлов с теми, что получаются в ходе автоматического рефакторинга. Для проверки процента покрытия тестами исходного кода использовался инструмент `llvm-cov`. Для этого использовались флаги компилятора Clang `-fprofile-instr-generate`, `-fcoverage-mapping`, которые добавляют дополнительную информацию для последующего анализа покрытия кода при запуске тестов. После запуска тестов генерируется бинарный файл с данными о покрытии, который впоследствии используется `llvm-cov` для генерации человеко-читаемого отчета о покрытии. Итоговое покрытие исходного кода составляет 79,267%.

Заключение

В результате проделанной работы были выполнены следующие задачи.

- Проведен обзор существующих решений и технологий.
- Реализованы детекторы для ситуаций, не соответствующих заданным требованиям заказчика.
- Реализованы авторефакторинги кода для ситуаций, не соответствующих требованиям заказчика.
- Протестирована реализованная функциональность.

Реализация:

<https://github.com/MinyazevR/auto-refactorings-clang-tidy-module>

Разработка ведется под аккаунтом [MinyazevR](#)

Список литературы

- [1] AUTOSAR. AUTOSAR Coding Standards. — URL: <https://www.autosar.org/standards> (online; accessed: 2024-03-07).
- [2] Belevantsev A. A. Multilevel static analysis for improving program quality. — URL: <https://www.ispras.ru/dcouncil/docs/diss/2017/belevancev/dissertacija-belevancev.pdf> (online; accessed: 2024-03-07).
- [3] Center CERT Coordination. CERT C Secure Coding Standard. — URL: <https://wiki.sei.cmu.edu/confluence/display/c> (online; accessed: 2024-03-07).
- [4] Checking programs for compliance with MISRA C standard using the Clang framework / R. A. Buchatskiy, Y. A. Churkin, K. A. Chibisov et al. — URL: https://www.ispras.ru/proceedings/docs/2023/35/5/isp_35_2023_5_169.pdf (online; accessed: 2024-03-07).
- [5] FSmell: Recognizing Inline Function in Binary Code / Wei Lin, Qingli Guo, Jiawei Yin et al. — URL: https://link.springer.com/chapter/10.1007/978-3-031-51476-0_24 (online; accessed: 2025-05-14).
- [6] A Formal Semantics for P-Code / Nico Naus, Freek Verbeek, Dale Walker, Binoy Ravindran. — URL: https://link.springer.com/chapter/10.1007/978-3-031-25803-9_7 (online; accessed: 2025-05-14).
- [7] MISRA. MISRA C STANDARDS. — URL: <https://www.misra.org.uk/> (online; accessed: 2024-03-07).