На правах рукописи

**Чан Ти Тхиен**

# Разработка нового метода автоматизированного тестирования программных библиотек

Специальность 2.3.5 —
«Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Автореферат
диссертации на соискание учёной
степени кандидата технических наук

Москва — 2023

Работа выполнена в Федеральном государственном автономном образовательном учреждении высшего образования «Московский физико-технический институт (национальный исследовательский университет)».

Научный руководитель: д.ф.-м.н., академик РАН
**Аветисян Арутюн Ишханович**

Официальные оппоненты: **Шабанов Борис Михайлович**, член-корреспондент РАН, доктор технических наук, доцент, директор Межведомственного Суперкомпьютерного Центра РАН, заместитель директора по научной работе Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

**Маркин Дмитрий Олегович**, кандидат технических наук, сотрудник Федерального государственного казенного военного образовательного учреждения высшего образования «Академия Федеральной службы охраны Российской Федерации»

Ведущая организация: Федеральное государственное бюджетное образовательное учреждение высшего образования «МИРЭА – Российский технологический университет»

Защита состоится 19 октября 2023 г. в 17 часов на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В. П. Иванникова РАН по адресу: 109004, г. Москва, ул. А. Солженицына, дом 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования им. В. П. Иванникова РАН.

Автореферат разослан “____” сентября 2023 г.

Ученый секретарь
диссертационного совета 24.1.120.01,
кандидат физико-математических наук Зеленов С.В.

**Общая характеристика работы**

**Актуальность темы**. Индустрия создания программного обеспечения (далее - ПО) растет быстрыми темпами. Согласно отчету компании Gartner, в 2022 году глобальный рынок ПО оценивался в 4.534 миллиарда долларов США, что на 3% больше, чем в 2021 году. Ожидается, что рост индустрии будет продолжаться и в ближайшие годы. Рост индустрии ПО обусловлен многими факторами. Сегодня ПО используется в различных отраслях, от банковского и финансового секторов до здравоохранения и государственного управления. При этом появляются новые технологии и требования, что приводит к созданию новых программных продуктов и услуг. Также следует отметить, что развитие интернета, мобильных устройств и облачных технологий создает новые возможности для создания и использования ПО.

При разработке ПО широко применяются программные библиотеки, которые предоставляют собой набор функций, решающих конкретную задачу в программе. Использование библиотек позволяет значительно сократить время разработки ПО, так как разработчики могут использовать готовые решения вместо написания кода, что, в частности, позволяет уменьшить количество ошибок. Вторым значимым преимуществом использования библиотек является возможность стандартизации написания программного кода, что помогает избежать дублирования кода и повышает эффективность труда программистов. Оба описанных выше фактора позволяют повысить качество ПО, но вместе с тем появляется необходимость проведения всестороннего тестирования программных библиотек, так как наличие ошибки в библиотеке вносит ошибку во всё программное обеспечение, которое эту библиотеку использует.

Одним из наиболее эффективных методов тестирования ПО является фаззинг. Впервые он был предложен группой под руководством Б. Миллера. Фаззинг позволяет обнаруживать ошибки, которые могут привести к аварийному завершению программы, утечке памяти, доступу к конфиденциальной информации и к другим проблемам безопасности. Исследователи уделяют большое внимание данному методу тестирования. С 2007 в известных изданиях (ACM digital library, Elsevier ScienceDirect, IEEEXplore digital library и т.д.) было опубликовано более 150 работ по теме фаззинга. С помощью технологии фаззинга Google обнаружила более 16.000 ошибок в браузере Chrome и более 11.000 ошибок в более чем 160 проектах ПО с открытым исходным кодом. Microsoft рассматривает фаззинг как один из этапов жизненного цикла разработки программного обеспечения, нацеленный на поиск уязвимостей и повышения стабильности своих продуктов (Microsoft SDL). Большинство опубликованных работ посвящено улучшению покрытия выполнения программы, разработке новых инструментов фаззинга и оптимизации генерации тестовых данных, которая базируется на следующих техниках:

- решении ограничений (constraint-based);

- грамматических спецификациях (grammar-based);

- символьном выполнении (symbolic execution);

- потоке помеченных данных (taint based);

- композициях перечисленных методов.

В настоящее время уже разработано несколько инструментов и платформ для фаззинга, наиболее известные среди них следующие: AFL, LibFuzzer, ClusterFuzz, Peach, Sulley, Powerfuzzer, ISP-Fuzzer, Avalanche.

Обзор источников показывает, что в большом круге вопросов автоматизации фаззинга практически не исследована задача автоматизации построения тестового окружения тестируемой программной системы. В литературе такое окружение называется фаззинг-обертками. Фаззинг-обертка – это программа, которая обращается к фаззеру (программе-генератору псевдослучайных данных), получает от него мутационные данные и передает их как входные параметры в тестируемую функцию, то есть фаззинг-обертка является представлением тестового варианта, содержащего сами тестовые входные данные или способ их получения посредством псевдослучайной генерации (мутации) и вызов тестируемой функции.

Близкой темой занимались ученые ИСП РАН, которые в 2008 году предложили технологию массового создания тестов работоспособности Азов. Данная технология использует имеющуюся информацию об интерфейсных операциях, о типах параметров и результатов операций. Технология использовалась для массового создания небольших тестов на работоспособность программных библиотек, реализующих множество интерфейсов стандарта Linux Standard Base (LSB). Предложенная технология была успешной в случае, когда отсутствует информация о приложениях, использующих интерфейсы, но она не учитывает такую информацию, если она становится доступной. Кроме того, в этой технологии не предусмотрено взаимодействие с фаззерами.

Разработка фаззинг-оберток является необходимой частью работ по фаззингу. В случае, когда число и сложность тестовых сценариев становится большим, трудоемкость разработки фаззинг-оберток является критическим звеном, определяющим возможность и эффективность фаззинга в условиях промышленного применения этой технологии. Таким образом, методы автоматизации генерации фаззинг-оберток, которые позволяют сократить трудоемкость фаззинга, являются важной актуальной задачей.

**Целью** данной работы является автоматизация тестирования программных библиотек при помощи генерации фаззинг-оберток в

условиях наличия и отсутствия данных о контексте использования функций библиотек.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Исследовать совокупность программных сущностей и их взаимосвязей в программном коде, необходимых для конструирования вызовов функций, подлежащих фаззинг-тестированию.
2. Разработать метод автоматизированной генерации фаззинг-оберток для функций библиотеки в условиях отсутствия и с учетом контекста использования библиотеки.
3. Разработать программные средства для автоматизированного анализа кода библиотеки, автоматизированной генерации фаззинг-оберток для функций библиотек и сбора и анализа результатов тестирования.

**Соответствие паспорту специальности.** Цели и задачи диссертационного исследования соответствуют направлениям исследований, предусмотренным паспортом специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей». Область исследования настоящей диссертации включает в себя методы и алгоритмы анализа программ (пункт 1 паспорта специальности), языки программирования и семантику программ (пункт 2), методы, архитектуры, алгоритмы, языки и программные инструменты организации взаимодействия программ и программных систем (пункт 3).

**Научная новизна:**

1. Предложен метод генерации фаззинг-оберток для функций библиотеки, который позволяет генерировать нацеленные тесты в условиях отсутствия информации о контексте использования библиотеки.
2. Предложен метод генерации фаззинг-оберток для функций библиотеки с учетом контекста использования библиотеки в пользовательской программе, который позволяет нацелить генератор только на используемые интерфейсы библиотеки.

**Практическая значимость.** Практическая значимость результатов диссертации заключается в разработке метода, позволяющего генерировать фаззинг-обертки на основе анализа исходного кода и контекстов использования функций.

**Методология и методы исследования.** Результаты диссертационной работы получены на базе использования методов статического анализа исходного кода библиотек; автоматизации создания фаззинг-оберток для

функций в библиотеке. При решении задачи нахождения контекстов вызовов функций используется теория компиляторов, в том числе анализ потока управления и данных.

**Основные положения, выносимые на защиту:**

1. Предложен метод генерации фаззинг-оберток для функций библиотеки в условиях отсутствия контекста использования.
2. Предложен метод генерации фаззинг-оберток для функций библиотеки с учетом контекста использования.
3. Разработано программное средство Futag для реализации предложенных методов.

**Достоверность.** Выводы диссертации подтверждены данными научных экспериментов, полученными с помощью разработанных методов анализа и статического анализатора Clang. Теоретическую и методологическую основу проведенных разработок и исследований составили труды отечественных и зарубежных авторов в области теории компиляторов, а также решения, созданные и опубликованные в российских и зарубежных патентах и свидетельствах на изобретения РФ. Положения и выводы, сформулированные в диссертации, получили квалифицированную апробацию на международных и российских научных конференциях и семинарах. Достоверность также подтверждается публикациями результатов исследования в рецензируемых научных изданиях.

**Апробация работы**. Основные результаты работы докладывались на:

1. Открытая конференция ИСП РАН. Москва. 2020.
2. Международная конференция «Иванниковские чтения». Нижний Новгород 2021.
3. Международная техническая конференция по открытой СУБД PostgreSQL «PGConf.Russia». Москва. 2021.
4. Ломоносовские чтения. Научная конференция. Москва. 2022.
5. IX International Conference «Engineering & Telecommunication - En&T-2022». Москва. 2022.
6. Открытая конференция ИСП РАН. Москва. 2022.

**Личный вклад**. Все представленные в диссертации результаты получены лично автором.

**Публикации автора по теме диссертации**


1. Tran, C. T. Futag: Automated fuzz target generator for testing software libraries / C. T. Tran, S. Kurmangaleev // 2021 Ivannikov Memorial Workshop (IVMEM). — 2021. — P. 80—85. — URL: https://ieeexplore.ieee.org/document/9693749. — (Scopus).
2. Свидетельство о гос. регистрации программы для ЭВМ. Futag / Т. Т. Чан; Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова

Российской Академии Наук (ИСП РАН) (RU). — No 2021662358; заявл. 06.08.2021; опубл. 16.08.2021, 2021663344 (Рос. Федерация). - URL: https://new.fips.ru/registers-doc-view/fips_servlet?DB=EVM&DocNumber=2021663344.
3. Futag - автоматический генератор фаззинг-оберток для программных библиотек / Т. Т. Чан [и др.] // Ломоносовские чтения - 2022. - 2022.
4. Tran, C. T. Application of automatic code generation and fuzzing technology for C / C++ library testing / C. T. Tran // IX International Conference "Engineering & Telecommunication En&T 2022". - 2022.
5. Tran, C. T. Research on automatic generation of fuzz-target for software library functions / C. T. Tran, D. Ponomarev, A. Kuznhesov // 2022 Ivannikov ISPRAS Open Conference (ISPRAS). — 2022. — P. 95—99. — URL: https://ieeexplore.ieee.org/document/10076871. — (Scopus).

**Публикации**. Основные результаты по теме диссертации изложены в 4 печатных изданиях, 2 — в периодических научных журналах, индексируемых Scopus, 2 — в тезисах докладов. Зарегистрирована 1 программа для ЭВМ.

В первой публикации [1] автор описал метод анализа исходного кода библиотеки и метод генерации фаззинг-оберток для функций в условиях отсутствия контекста использования функций библиотек.

Реализация программы Futag [2] выполнена автором полностью.

В публикации [3] автор описал этапы обработки в реализованной программе «Futag»: препроцессирование, генерацию и сбор результата.

В публикации [5] автор описал способы повышения корректности при генерации фаззинг-оберток для функций.

**Объем и структура работы**. Диссертация состоит из введения, 5 глав, заключения и двух приложений. Полный объём диссертации составляет 114 страниц, включая 24 рисунка, 31 листинг и 3 таблицы. Список литературы содержит 74 наименования.

## Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, формулируется цель, ставятся задачи работы, излагается научная новизна и практическая значимость представляемой работы. Представляются выносимые на защиту основные научные положения, приводятся сведения об апробации полученных результатов и публикациях по теме исследования.

**Первая глава** посвящена обзору работ по теме диссертации. В главе рассмотрены методы тестирования библиотек, в том числе технология фаззинга и назначение, структура фаззинг-обертки. Также рассмотрены

свойства характеристики библиотек и обзор инструментов генерации фаззинг-оберток для функций библиотек.

Фаззинг-обертка – это программа, которая обращается к фаззеру (программе-генератору псевдослучайных данных), получает от него мутационные данные и передает их как входные параметры в тестируемую функцию, то есть фаззинг-обертка является представлением тестового варианта, содержащего тестовые входные данные, полученные методом псевдослучайной генерации (мутации) и вызов тестируемой функции.

В результате генерации исходного текста фаззинг-обертки в ней должны появиться части тестового варианта, выполняющие:

- подключение необходимых заголовочных файлов тестируемой библиотеки;
- выделение памяти для мутационных данных и их передачу из буфера фаззера аргументам тестируемой функции;
- конструирование вызовов тестируемой функции;
- освобождение выделенной памяти.

Среди инструментов генерации фаззинг-оберток для функций библиотек самыми распространенными являются FUDGE, FuzzGen и IntelliGen. Все эти инструменты используют статический анализ для исследования совокупности сущностей, их взаимосвязей в программном коде и контекста использования, необходимых для конструирования вызовов функций. В условиях, когда отсутствуют пользовательские программы, которые используют тестируемую библиотеку, FUDGE и FuzzGen не могут генерировать фаззинг-обертки. Доступ к FUDGE и IntelliGen в настоящее время закрыт. FuzzGen хорошо работает с библиотеками Android (при наличии кодовой базы приложений, использующих библиотеки).

Делается вывод о том, что автоматизация генерации фаззинг-оберток является актуальной задачей, и для ее решения необходимо:

- Исследовать техники выявления совокупностей сущностей и их взаимосвязей в программном коде, необходимых для конструирования вызовов функций, подлежащих фаззинг-тестированию.
- Разработать методы автоматизированной генерации фаззинг-оберток для функций библиотеки в условиях отсутствия информации о контексте и в условиях, когда контекст использования библиотеки доступен для анализа.
- Разработать программное средство для автоматизированного анализа кода библиотеки, автоматизированной генерации фаззинг-оберток для функций библиотек и сбора и анализа результатов тестирования.

**<u>Вторая глава</u>** посвящена исследованию вопроса проведения анализа исходного кода библиотек для выделения сущностей и их взаимосвязей в программном коде, необходимых для конструирования вызовов функций, подлежащих фаззинг-тестированию и для подготовки данных для генераторов фаззинг-оберток.

В данной главе описываются:

- метод и инструменты статического анализа, которые используются для анализа программного кода библиотеки и пользовательской программы в обоих случаях (в разделе 2.1);
- процесс конструирования вызовов функций простого и сложного типа в пользовательской программе (в разделе 2.2);
- схема интеграции статического анализа в процесс компиляции и сборки программного продукта (в разделе 2.3).

Для создания фаззинг-обертки необходимо исследовать совокупность сущностей, их взаимосвязей и программного интерфейса в коде тестируемой библиотеки, необходимых для конструирования вызовов функций, подлежащих фаззинг-тестированию. Если имеется доступ к контексту, то необходимо исследовать приложения, использующие тестируемые библиотеки. Для решения этих задач необходим статический анализатор. В связи с этим в работе был проведен поиск, сравнительный анализ статических анализаторов и выбор наиболее подходящего.

Раздел 2.1. посвящен обзору инструментов статического анализа. Инструментов статического анализа достаточно много, но в данной работе к ним предъявляются следующие требования:

- язык программирования анализируемой программы: инструмент должен поддерживать языки Си и Си++;
- интеграция: для упрощения процесса автоматизации генерации фаззинг-оберток инструмент должен легко интегрироваться в процесс компиляции и сборки программы;
- лицензионные ограничения: нужен инструмент с открытым доступом или свободной лицензией;
- поддержка: инструмент развивается и поддерживается в настоящее время;
- наличие средств для формирования запросов по промежуточному представлению кода.

После предварительного отбора оказалось, что для целей диссертационной работы потенциально могут использоваться инструменты статического анализа являются «CodeQL» и «Clang Static Analyzer».

**CodeQL** – это инструмент для анализа кода, разработанный компанией Semmle, которую приобрела компания Microsoft в 2019 году. CodeQL

позволяет производить статический анализ кода, выявлять ошибки, уязвимости и другие проблемы в исходном коде приложений. Основным преимуществом CodeQL является его способность проводить контекстно-чувствительный анализ кода, позволяющий находить дефекты и уязвимости, которые могут оставаться незамеченными при использовании других инструментов. CodeQL также позволяет создавать собственные запросы для анализа кода, что обеспечивает гибкость в работе с инструментом.

Однако при большом количестве запросов к кодовой базе пользователю нужно приобретать коммерческую лицензию CodeQL. То есть CodeQL можно использовать для экспериментов, но нельзя бесплатно использовать в промышленных проектах.

**Clang Static Analyzer** – это инструмент статического анализа кода, который предназначен для поиска ошибок и потенциальных проблем в коде, которые могут привести к сбоям в работе программы, утечкам памяти, неправильному использованию указателей и другим проблемам. С помощью Clang Static Analyzer можно создавать граф потока данных и анализировать состояния программы на различных участках кода. В состав Clang Static Analyzer входят программные средства для формирования запросов по промежуточному представлению кода: «ASTVisitor» (АСД-инспектор) и «ASTMatcher» (АСД-обработчик). Кроме того, Clang Static Analyzer входит в состав проекта LLVM (проект с открытым кодом и свободной лицензией), что позволяет легко его интегрировать в процесс компиляции (рисунок 1).

На основании проведенного сравнения для данной работы был выбран Clang Static Analyzer (схема работы во время компиляции на рис. 1).

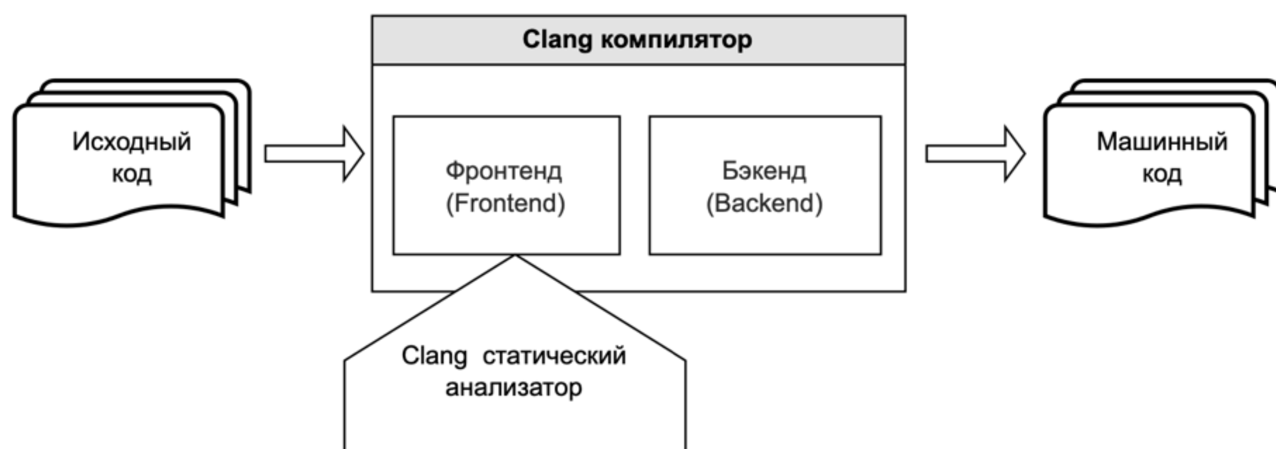


Рисунок 1. Схема использования статического анализатора Clang в Clang-компиляторе.

Раздел 2.2 посвящен конструированию вызовов функций простого и сложного типа в пользовательской программе.

Библиотеки решают типичные задачи со специальными структурными данными: видео, звуковые записи, JSON-формат данных и т.д. Перед обработкой эти специальные данные объявляются и инициализируются. В библиотеках на языке Си используются структуры для определения этих специальных данных и некоторые функции конструирования для инициализации значения этим структурам. А в библиотеках на языке Си++ специальные данные определяются с помощью класса (class), конструкторы которого отвечают за инициализацию объекта данного класса. Поэтому функции в программном интерфейсе библиотеки могут быть разделены на две группы:

- функции, которые отвечают за получение входных данных из пользовательской программы и за создание специальных структур данных. Эти функции имеют только аргументы простого типа;
- функции, которые обрабатывают специальные структуры данных. Перед вызовом этих функций требуется конструирование специальных данных с помощью функций в первой группе. Аргументы этих функций имеют сложный тип.

На рисунке 2 отображается процесс конструирования вызовы функций простого типа в пользовательской программе: для конструирования вызов обращается к определению функции и получает информацию о типе данных аргументов и о возвращаемом типе.

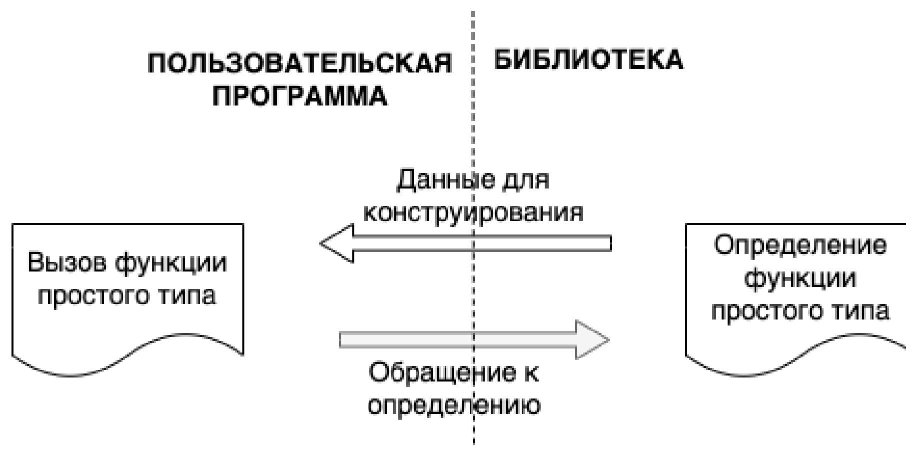


Рисунок 2. Процесс конструирования вызова функции простого типа.

На рисунке 3 отображается процесс конструирования вызовов функций сложного типа в пользовательской программе:

1. Генератор вызова функции сложного типа обращается к определению данной функции за информацией о типе данных аргументов и о возвращаемом типе.
2. Генератор анализирует определение функции сложного типа и его список аргументов. При обнаружении аргумента сложного типа генератор ищет определение функции простого типа, которое возвращает сложный тип аргумента для получения информации о конструировании. В случае, когда найдено несколько подходящих

определений, можно передавать каждое определение генератору и переходить к третьему этапу.

3. Генератор получает данные для конструирования найденного вызова функции простого типа.
4. На данном шаге генератор получает необходимую информацию для конструирования вызова функции сложного типа.
5. Для конструирования вызова функции сложного типа сначала генератор должен создать значение ее аргументов, для чего создает необходимый вызов функции простого (или более простого) типа, и уже эта функция возвращает значение нужного сложного типа.
6. На последнем шаге генерируется вызов функции сложного типа.

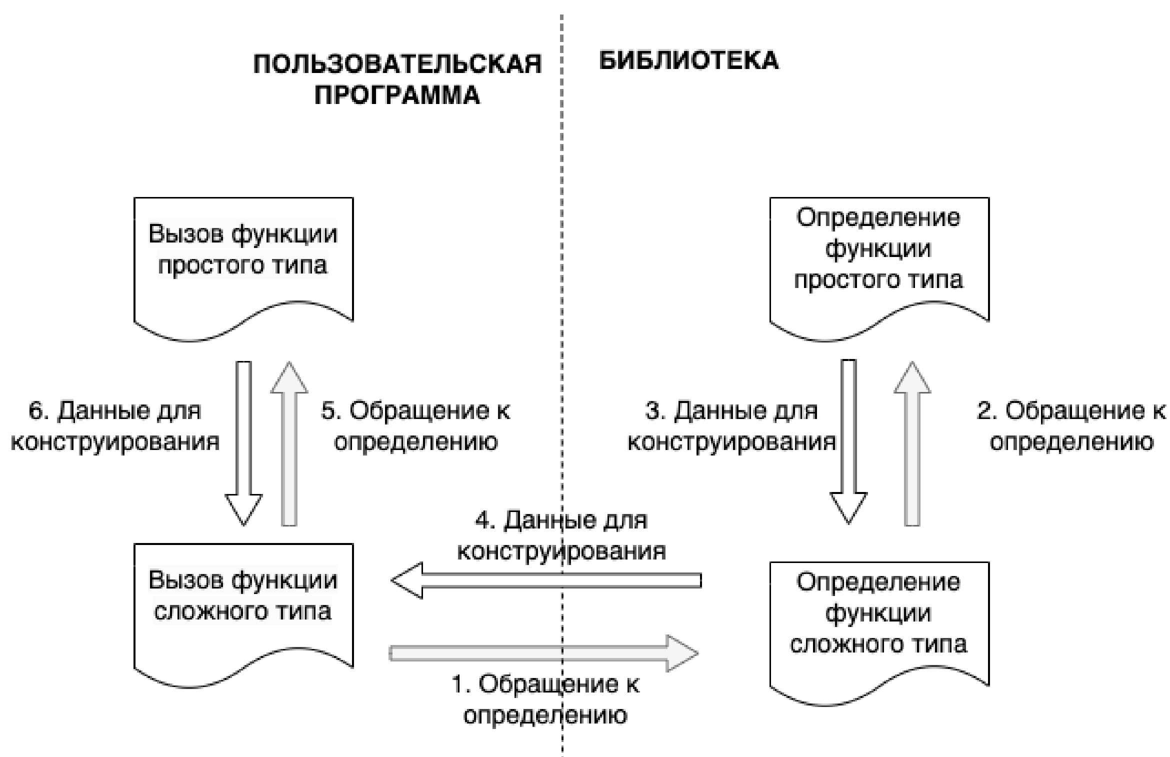


Рисунок 3. Процесс конструирования вызова функции сложного типа.

В разделе 2.3. описывается схема интеграции статического анализа в процесс компиляции и сборки библиотеки. Схема интеграции статического анализа в процесс компиляции и сборки библиотеки иллюстрируется на рисунке 4 и состоит из 4 этапов:

1. Обнаружение и запуск конфигурационного скрипта: на вход данного этапа может поступать исходный код библиотеки или исходный код пользовательской программы. Был разработан скрипт для обнаружения и запуска конфигурационного скрипта с конкретными параметрами: переменные окружающей среды, количество потоков при компиляции, место установки и т.д.
2. Запуск статический анализа: для разных задач запускаются разные инструменты статического анализа. Например, для задачи генерации фаззинг-оберток в условиях отсутствия контекста использования

тестируемой библиотеки запускаются инструменты для сбора характеристик и взаимосвязей между сущностями кода; для задачи генерации фаззинг-оберток с учетом контекста использования тестируемой библиотеки запускаются инструменты для обнаружения контекст использования. Инструменты Clang статического анализа могут запускаться с помощью утилиты «scan-build».

3. Компиляция с санитайзерами: при необходимости исходный код компилируется с санитайзерами для обнаружения дефектов в процессе фаззинга.
4. Сбор результата анализа: на данном этапе собираются результат статического анализа (этап 2) и параметры компиляции (этап 3).

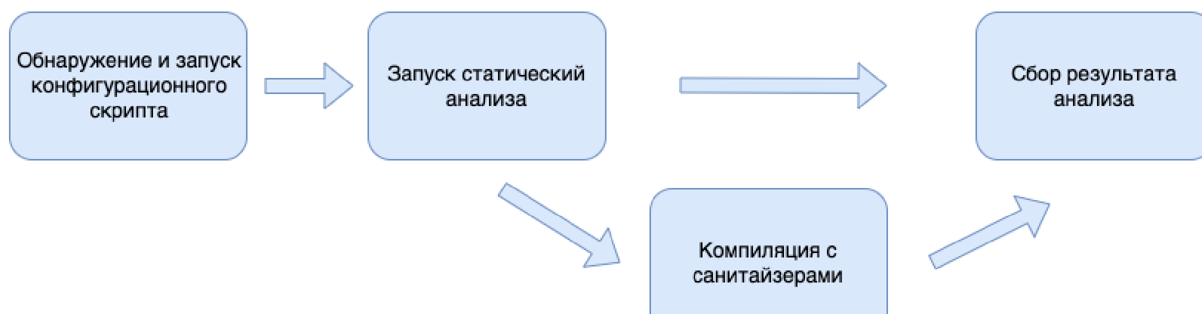


Рисунок 4. Интеграция статического анализа в процесс компиляции библиотеки.

Результатом статического анализа является база знаний, в которой содержится информация о сущностях и их взаимосвязях, необходимых для построения вызовов тестируемых функций. База знаний представлена в форме JSON-файлов.

Вывод: в главе предложена и описана схема реализации и интеграции статического анализа в процесс компиляции, которая позволяет исследовать совокупность сущностей и их взаимосвязей в программном коде, необходимых для конструирования вызовов функций, подлежащих фаззинг-тестированию. Для решения задач анализа исходного кода были разработаны необходимые АСД-инспекторы, АСД-детекторы и АСД-обработчики.

В **третьей главе** описывается **«метод генерации фаззинг-оберток для функций библиотеки в условиях отсутствия контекста использования»,** который выносится на защиту.

Данный метод дает возможность генерации фаззинг-обертки для функций библиотеки даже в условиях отсутствия контекста использования тестируемой библиотеки. Для представления данного метода сначала проводятся анализ способов инициализации переменных разных типов и способов десериализации буфера фаззера для передачи мутационных данных в тестируемую функцию, затем описывается предложенный метод. В конце главы описывается способ повышения корректности генерации фаззинг-оберток.

В разделе 3.1. проводится анализ способов инициализации переменных разных типов данных в языках Си и Си++.

В разделе 3.2. описываются способы десериализации буфера фаззера для передачи мутационных данных в тестируемую функцию, которая имеет аргументы разных типов.

В разделе 3.3. описывается **метод генерации фаззинг-оберток для функций библиотеки в условиях отсутствия информации о контексте использования**.

Ключевыми отличиями представленного метода являются:

- генерация оберток для всех функций библиотеки, находящихся в доступной области видимости;
- способ повышения корректности генерации путем анализа функций и их вызовов во всем АСД для дополнения знаний о типах данных.

Данный метод позволяет генерировать фаззинг-обертки для всех доступных функций библиотеки в условиях отсутствия контекстов ее использования. Схема данного метода представлена на рисунке 5.

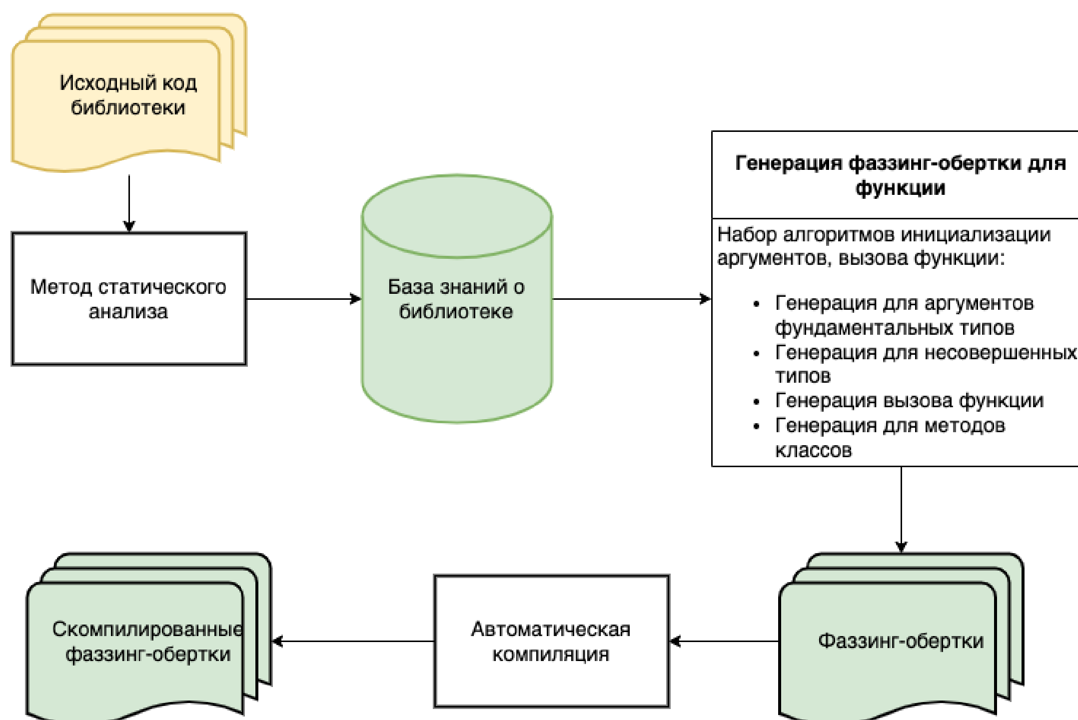


Рисунок 5. Схема метода генерации фаззинг-оберток для функций библиотеки в условиях отсутствия контекста использования.

Метод состоит из трех последовательных этапов. Первым этапом является статический анализ, на вход которого поступает исходный код тестируемой библиотеки. Работа данного этапа описывается в разделе 2.3. На выходе первого этапа получается база знаний о тестируемой библиотеке, которая включает в себя:

– определение функции: возвращаемый тип, список аргументов и их тип данных;
– определение перечисляемого типа: название, список констант;
– определение структуры, ее полей;
– определение класса и его атрибуты, методы;
– определение нового типа данных с помощью ключевого слова typedef;
– перечень заголовочных файлов, включенных в анализируемую единицу трансляции;
– параметры компиляции.

На втором этапе генерируются фаззинг-обертки для функций, информация о которых поступает из базы знаний о библиотеке. Данный этап включает набор способов генерации: генерация аргументов разных типов, генерация вызова функции или генерация для метода классов. Алгоритм обработки и запуска генерации состоит из следующих шагов (блок-схема алгоритма приведена в диссертационной работе):

1. Для каждой функции из базы знаний о библиотеке собирается дополнительная информация о файле, содержащем тестируемую функцию: список включенных заголовочных файлов, пути включения и параметры при компиляции.
2. Проверка: обработана ли последняя функция в базе знаний, если да, то алгоритм завершается, иначе переходим на шаг 3.
3. Проверка: список аргументов функции пустой? Если да, то берем следующую функцию для обработки (шаг 1), иначе переходим на 4 для итерации по всем аргументам.
4. Запускается итерация по всем аргументам.
5. Проверка: если обработан последний аргумент, то переходим на шаг 11, иначе переходим к шагу 6.
6. Проверка: если тип данного аргумента - простой, то переходим к шагу 7, иначе переходим к шагу 8.
7. Происходит десериализация буфера; аргумент функции инициализируется и уточняется размер буфера. После этого переходим на шаг 4 для продолжения итерации аргументов.
8. Происходит поиск функций интерфейса, которые возвращают тип данных анализируемого аргумента.
9. Проверка: если нашлись функции в шаге 8, то переходим к шагу 10, иначе мы не знаем, как можно инициализировать аргумент данного типа и обрабатывает следующую функцию.
10. Происходит генерация вызова каждой из найденных функций, и для каждого вызова создается фаззинг-обертка. Все ее аргументы инициализируются, и переходим к шагу 7 для генерации анализируемого аргумента через найденную функцию.

11. Формируется фаззинг-обертка для тестируемой функции: к фаззинг-обертке добавляются заголовочные файлы, размер буфера данных и вызов функции с инициализируемыми аргументами. В случае, когда функция является конструктором класса, формируется объявление объекта класса, затем генерируется вызов конструктора. В случае, когда функция является методом класса, формируется объявление объекта класса с помощью конструктора, затем генерируется вызов метода. В данной работе рассматриваются 2 типа конструкторов: конструкторы по умолчанию и конструкторы, все аргументы которых имеют простой тип. Конструкторы с параметрами сложного типа пока не рассматриваются.

В результате второго этапа для одной функции может сгенерироваться несколько фаззинг-оберток.

На третьем этапе фаззинг-обертки компилируются с параметрами, предоставленными из базы знаний о тестируемой библиотеке: пути к заголовочным файлам, параметры компиляции, место для сохранения артефактов и т.д.

В разделе 3.4. описываются способы повышения корректности генерации фаззинг-оберток.

Имеется ряд причин, которые негативно сказываются на качестве (корректности) фаззинг-оберток, например:

1. Недостаточно информации о типе данных для генерации. Например: аргумент функции имеет тип целого типа, который обозначает файловый дескриптор. Значение файлового дескриптора задается после создания нового потока ввода-вывода, поэтому, если передавать мутационное значение, то генерация является неверной; или аргумент функции имеет строковый тип, который обозначает путь к файлу. В этом случае необходимо передавать строку в формате системного пути, иначе результат генерации будет некорректным.

2. Значения входных параметров могут оказаться несогласованными между собой. Например: в языке Си в качестве входных аргументов функции часто передаётся пара строка и её размер, то есть при попытке передать случайные значения результат генерации будет некорректным.

3. Значение входных данных может быть не инициализировано до вызова. Например: перед вызовом тестируемой функции необходимо вызвать другие функции по конкретному контексту.

Для устранения первой и второй проблемы необходимо:

- проанализировать определение функции для определения того, как параметры используются внутри функции;

- проанализировать, как инициализируются параметры для создания вызова данной функции в других функциях.

А для устранения проблемы третьего вида необходимо анализировать контексты использования тестируемой библиотеки в пользовательских программах (глава 4).

Были разработаны АСД-обработчики для анализа функций и их вызовов во всем АСД: внутренние и внешние АСД-обработчики.

Внутренний АСД-обработчики – это АСД-обработчик, который анализирует код в теле тестируемой функции и определяет, какие системные вызовы используют аргументы данной функции. Если такие вызовы найдутся, то АСД-обработчик ищет места использования аргументов и дополняет знания о способе их использования.

Внешний АСД-обработчик – это АСД-обработчик, который запускается в каждом определении функций библиотеки и ищет вызов анализируемой функции. При нахождении вызова АСД-обработчик анализирует способ инициализации аргументов и дополняет знания о способах их использования.

В итоге с помощью представленного метода в условиях отсутствия пользовательских программ генерируются фаззинг-обертки для функций библиотеки. Пример результата метода показывается на листинге 1 для функции *«json_tokener_parse_ex»*. Данная функция принимает 3 аргумента:

- аргумент «s_tok» типа «struct json_tokener *» (строка 8); в момент анализа данный тип является неполным, и в результате поиска функции, возвращающей данный тип, нашлась функция «json_tokener_new». Поэтому «s_tok» инициализируется с помощью функции «json_tokener_new»;
- аргумент «str_str» типа «const char *», который принимает разделенное значение буфера (строка 14);
- аргумент «sz_len» типа «int», которому присваивается длина строки «str_str» (строка 15).

Листинг 1. Результат метода генерации фаззинг-обертки в условиях отсутствия контекста использования

```
1  #include "json.h"
2  extern "C" int LLVMFuzzerTestOneInput(uint8_t * Fuzz_Data, size_t Fuzz_Size){
3      if (Fuzz_Size < sizeof(char)) return 0;
4      size_t dyn_cstring_buffer = (size_t) ((Fuzz_Size - sizeof(char))));
5      size_t dyn_cstring_size [1];
6      dyn_cstring_size[0] = dyn_cstring_buffer;
7      uint8_t * futag_pos = Fuzz_Data;
8      struct json_tokener * s__tok = json_tokener_new();
9      char * rstr_str = (char *) malloc((dyn_cstring_size[0] + 1)* sizeof(char));
10 //GEN_CSTRING
11     memset(rstr_str, 0, dyn_cstring_size[0] + 1);
12     memcpy(rstr_str, futag_pos, dyn_cstring_size[0]);
```

| 13 | futag_pos += dyn_cstring_size [0]; |
|---|---|
| 14 | const char * str_str = rstr_str; |
| 15 | int sz_len = (int) dyn_cstring_size[0]; //GEN_SIZE |
| 16 | json_tokener_parse_ex(s__tok, str_str, sz_len ); // FUNCTION_CALL |
| 17 | if (rstr_str){ |
|  | free(rstr_str); |
| 18 | rstr_str = NULL; |
| 19 | } |
| 20 | return 0; |
| 21 | } |

В **четвёртой главе** описывается **«метод генерации фаззинг-оберток для функций библиотеки с учетом контекста использования»**, который выносится на защиту.

При использовании сложной библиотеки для выполнения некоторых задач требуются объявление специальных переменных и создание последовательности инструкций, которые являются контекстом использования библиотеки. Для фаззинг-тестирования таких библиотек сначала необходимо определить контексты использования в пользовательской программе, затем сгенерировать фаззинг-обертку для найденных контекстов. Предлагаемый метод позволяет повысить нацеленность фаззинга за счет использования информации о контексте.

Для представления метода сначала описывается способ определения контекстов использования (раздел 4.1.), затем проводится обзор методов анализа для поиска контекстов в исходном коде программы: анализ потока управления и потока данных (раздел 4.2.). После этого описывается предложенный метод (раздел 4.3.).

В разделе 4.1. описывается способ определения контекста вызовов функций библиотеки в пользовательских программах. Контекст использования функций — это последовательность упорядоченных инструкций, операторов в программе, которые взаимодействуют между собой с общей целью. Контекст использования тестируемой библиотеки базируется на процессе обработки данных между библиотекой и пользовательской программой и включает в себя:

- инициализируемые переменные – это переменные, которые инициализируются функциями программного интерфейса библиотеки. Такие переменные обычно используются для передачи данных из пользовательской программы в библиотеку и оформление полученных данных в формат, обрабатываемый тестируемой библиотекой.
- слайсинговые инструкции – это инструкции, которые модифицируют инициализируемые переменные.

В разделе 4.2. проводится обзор методов анализа потока управления и потока данных, которые используются для обнаружения контекста использования библиотеки.

В разделе 4.3. описывается **метод генерации фаззинг-оберток для функций библиотеки с учетом контекста использования в пользовательской программе**.

Метод состоит из трех последовательных этапов:

Этап 1: на входе принимаются исходный код пользовательской программы, использующий тестируемые функции, и база знаний о тестируемой библиотеке. Для каждого определения функции запускается статический анализ для нахождения инициализируемых переменных и слайсинговых инструкций. На выходе получается набор контекстов использования тестируемой библиотеки.

Этап 2: найденные контексты на первом этапе поступают на вход второго этапа для генерации фаззинг-обертки.

Этап 3: третий этап аналогичен третьему этапу метода генерации фаззинг-оберток для функций в условиях отсутствия контекста использования. Однако при инициализации переменных и генерации вызовов есть особенности, которые будут описаны на рисунке 6.

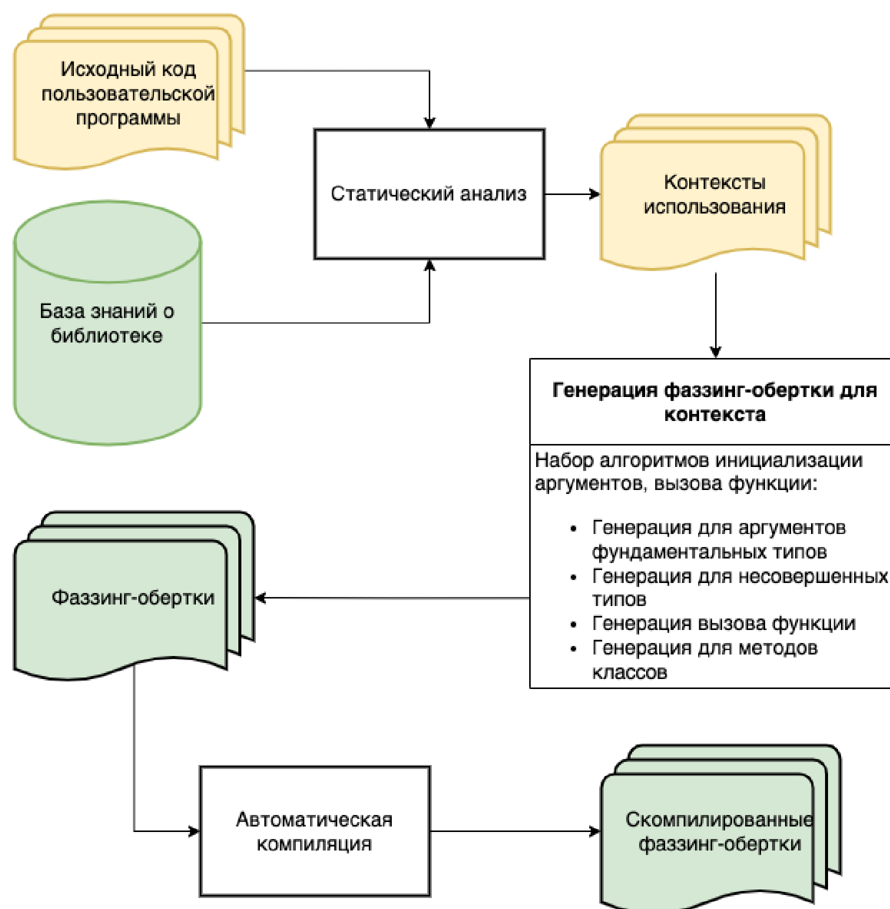


Рисунок 6. Схема метода генерации фаззинг-оберток для функций библиотеки с учетом контекста использования.

Алгоритм нахождения контекстов на этапе 1 принимает на входе исходный код пользовательской программы и базу знаний о тестируемой

библиотеке и включает в себя следующие шаги (блок-схема алгоритма приведена в диссертационной работе):

1. Анализируются все функции в пользовательской программе.
2. Проверка: если в анализируемой функции есть инициализируемая переменная, то переходим на шаг 4, иначе переходим на шаг 3. Для нахождения инициализируемой переменной выполняется поиск инструкций, которые являются либо объявлением переменной, которая имеет тип данных, входящий в базу знаний о тестируемой библиотеке; либо являются бинарным оператором, правая часть которого является вызовом функции программного интерфейса тестируемой библиотеки.
3. Проверка: если в наличии есть функция, которая анализировалась, то берем следующую функцию и идем на шаг 2, иначе алгоритм завершается.
4. На данном шаге найденная переменная добавляется в набор инициализируемых переменных И, строится граф потока управления и выполняется поиск всех путей выполнения данного графа; с помощью этих путей все найденные инструкции упорядочиваются на шаге 14. После данного шага запускается поиск слайсинговых инструкций.
5. Выполняется итерация всех инициализируемых переменных в наборе И.
6. Выполняется поиск слайсинговых инструкций для обрабатываемой переменной. Инструкция считается слайсинговой, если:
   – она является бинарным оператором «=», в правой части которого присутствует обрабатываемая инициализируемая переменная.
   – она является вызовом функции, в списке аргументов которой присутствует обрабатываемая инициализируемая переменная.
5. Нужно отметить, что в данной работе обрабатываются эти 2 условия, но возможно добавить еще другие условия для поиска слайсинговых инструкций при необходимости. Например, слайсинговая инструкция являются частью сложного выражения или слайсинговая инструкция является условием циклов «for» и т.д.
7. Проверка: если не нашлась слайсинговая инструкция, переходим к шагу 13 для проверки наличия инициализируемых переменных, иначе переходим на шаг 8.
8. Каждая найденная инструкция обрабатывается. Если:
   – она является бинарным оператором «=», тогда переменная в правой части передается на проверку инициализируемой переменной.
   – она является вызовом функции, все аргументы которой обрабатываются: если аргумент является константой – эта константа сохраняется при генерации фаззинг-обертки; если

аргумент является вызовом функции, то данный вызов заменяется новой переменной для упрощения инструкции; если аргумент является переменной, то она передается на проверку инициализируемой переменной.

9. Проверка: если нашлась новая инициализируемая переменная, то переходим на 10, иначе переходим на 11.
10. На этом шаге переменная добавляется в набор инициализируемых переменных И для дальнейшей обработки, после этого переходим на шаг 11.
11. Слайсинговая инструкция добавляется в набор слайсинговых инструкций К.
12. Проверка: если была обработана последняя слайсинговая инструкция, то переходим на шаг 13, иначе переходим на 8 для обработки следующей инструкции.
13. Проверка: если была обработана последняя инициализируемая переменная, то переходим на шаг 14, иначе переходим на 5 для обработки следующей переменной.
14. Для подготовки контекста проходит упорядочивание наборы И и К по путям выполнения: по базовым блокам и по инструкциям внутри каждого базового блока.

На выходе получаются контексты использования тестируемой библиотеки в пользовательской программе.

В итоге с помощью представленного метода в условиях наличия пользовательской программы сгенерировались фаззинг-обертки для контекстов использования библиотеки. Пример результата метода показывается в листинге 2. На строке 17 определяется инициализируемая переменная «body», которая обрабатывается вызовами функции «json_object_object_add» на строках 24, 32 и вызовом функции «json_object_put» на строке 33.

Листинг 2. Результат обработки метода генерации фаззинг-обертон с учетом контекста использования

```
1  #include "json.h"
2  int LLVMFuzzerTestOneInput(uint8_t * Fuzz_Data, size_t Fuzz_Size){
3      if (Fuzz_Size < 2) return 0;
4      size_t dyn_buffer = (size_t) ((Fuzz_Size - 2));
5      //generate random array of dynamic string
6      sizes size_t dyn_size [2];
7      srand(time(NULL));
8      dyn_size[0] = rand() % dyn_buffer;
9      size_t remain = dyn_size [0];
10     for(size_t i = 1; i< 2 - 1; i++){
11         dyn_size[i] = rand() % (dyn_buffer – remain);
12         remain += dyn_size[i];
13     }
14     dyn_size[1] = dyn_buffer - remain
15     //end of generation random sizes
```

```
16	    uint8_t * pos = Fuzz_Data;
17	    struct json_object *body = json_object_new_object();
18	    char * rstr_str0 = (char *) malloc(dyn_size[0] + 1);
19	    memset(rstr_str0, 0, dyn_size[0] + 1);
20	    memcpy(rstr_str0, pos, dyn_size[0] );
21	    pos += dyn_size [0];
22	    const char * str_str0 = rstr_str0;
23	    struct json_object *FutagRefVarueK = json_object_new_string(str_str0);
24	    json_object_object_add(body, "dataHash", FutagRefVarueK);
25	    //GEN_CSTRING
26	    char * rstr_str1 = (char *) malloc(dyn_size[1] + 1);
27	    memset(rstr_str1, 0, dyn_size[1] + 1);
28	    memcpy(rstr_str1, pos, dyn_size[1] );
29	    pos += dyn_size [0];
30	    const char * str_str1 = rstr_str1;
31	    struct json_object *FutagRefVar73q = json_object_new_string(str_str1);
32	    json_object_object_add(body,"token",FutagRefVar73q);
33	    json_object_put(body);
34	    if (rstr_str){
35	        free(rstr_str);
36	        rstr_str = NULL;
37	    }
38	    return 0;
39	}
```

В **пятой главе** описана реализация предложенных методов программным средством Futag, которое состоит из модулей:

- модуль автоматизированного анализа (раздел 5.1.), который предназначен для анализа тестируемой библиотеки во время ее компиляции;
- модуль автоматизированной генерации фаззинг-оберток (раздел 5.2.), который предназначен для генерации и компиляции фаззинг-оберток в двух условиях: с и без контекста использования тестируемой библиотеки;
- модуль автоматизированного фаззинга и анализа результатов (раздел 5.3.), который предназначен для сбора и анализа результатов фаззинга.

В разделе 5.4 проводится количественная и качественная оценка работы программного средства Futag сравнивая с альтернативными инструментами. Также перечислены найденные ошибки программным средством Futag в популярных библиотеках: libpng и pugixml.

Результат работы, а именно программу Futag применяется в Научно-техническом центре «ФОБОС-НТ» при выполнении фаззинг-тестирования исследуемого программного обеспечения, исходные коды которого написаны на языках программирования Си/Си++. Архитектура программы Futag показана на рисунке 7.

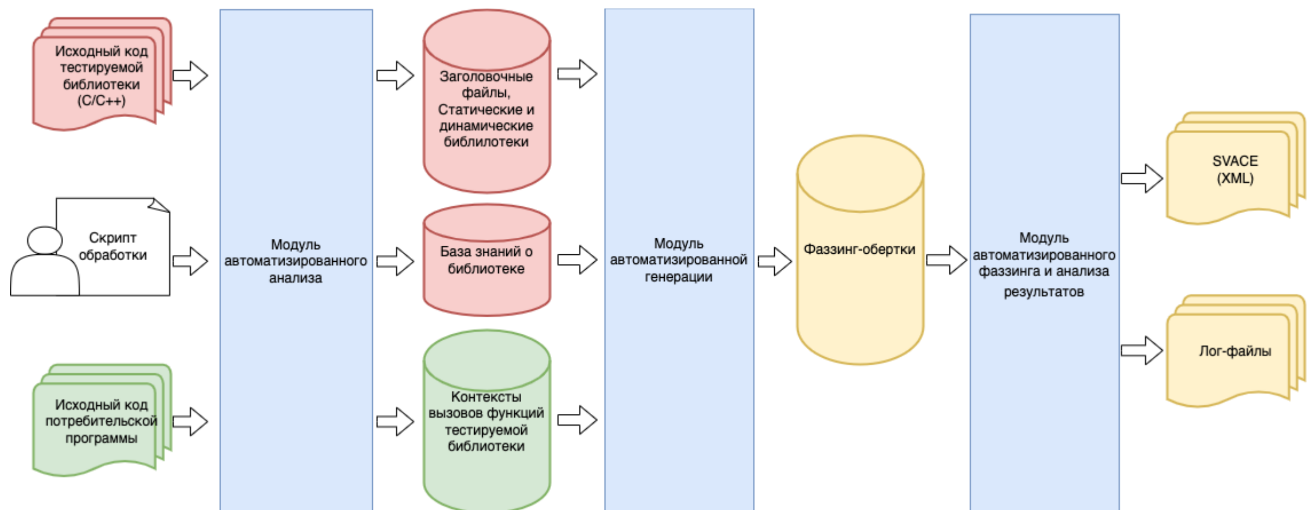


Рисунок 7. Архитектура программы Futag.

При использовании программы для тестирования библиотеки тестировщик создает скрипт запуска, пример которого показывается в листинге 3.

Листинг 3. Скрипт для запуска инструмент Futag

```
1   from futag.preprocessor import *
2   from futag.generator import *
3   from futag.fuzzer import *
4
5   testing_lib = Builder(
6       "futag -llvm/", # path to Futag package
7       "path/to/library/source/code"
8   )
9   testing_lib.auto_build() # build automatically
10  testing_lib.analyze() # analyze the information
11  g = Generator(
12      "futag -llvm/", # path to Futag package
13      "path/to/library/source/code"
14  )
15  g.gen_targets() # Generate fuzz drivers
16  g.compile_targets() # Compile fuzz drivers
17  fuzzer = Fuzzer(
18      "futag -llvm/", # path to Futag package
19      "futag-fuzz-drivers", #contains fuzz-targets
20      fork=4,
21      totaltime=60,
22      gdb=True,
23  )
24  fuzzer.fuzz()
```

После запуска скрипта Futag выполняет следующие действия:

- компилирует исследуемую библиотеку с датчиками обнаружения ошибок (AddressSanitizer) и собирает информации о параметрах команд компиляции;
- генерирует и компилирует фаззинг-обертки;

– запускает скомпилированные фаззинг-обертки и собирает результат фаззинга;
– результаты фаззинга автоматически обрабатывается отладчиком GDB для сбора информации о переменных в месте ошибки.

Для оценки работы программы Futag оцениваются ее результаты для разных библиотек. Результаты анализа показаны в таблице 1. Как правило, в небольших проектах (в которых участвует не более 10 разработчиков) один разработчик пишет от 20 до 125 строк кода в день. Если экстраполировать эту статистику на процесс написания фаззинг-обёрток, то получим результат в таблице 2.

Futag автоматизирует процесс создания обёрток для функций библиотеки, что заметно сокращает временные затраты разработчиков на их создание.

| **Библиотека** | **Время генерации (секунды)** | **Кол-во фаззинг-оберток** | **Время компиляции (секунды)** | **Кол-во строк кода** |
|---|---|---|---|---|
| lib json-c | 180 | 612 | 3111 | 280.019 |
| libpostgres | 105 | 29 | 749 | 84.387 |
| curl | 4.210 | 21 | 152 | 9.617 |
| openssl | 2.172 | 255 | 269 | 19.458 |
| pugixml | 55 | 58 | 61 | 2.815 |
| libopus | 75 | 7 | 422 | 12.606 |

Таблица 1. Результат генерации фаззинг-оберток для библиотек

| **Библиотека** | **Кол-во фаззинг-оберток** | **Кол-во строк кода** | **Мин.время написания (человеко-дни)** | **Макс.время написания (человеко-дни)** |
|---|---|---|---|---|
| libjson-c | 612 | 280.019 | 2240 | 11.200 |
| libpostgres | 29 | 84.387 | 675 | 3375 |
| curl | 21 | 9.617 | 77 | 385 |
| openssl | 255 | 19.458 | 155 | 778 |
| pugixml | 58 | 2.815 | 22 | 113 |
| libopus | 7 | 12.606 | 100 | 504 |

Таблица 2. Время написания фаззинг-оберток вручную

В **заключении** приведены основные результаты работы, которые состоят в следующем:

1. Предложен метод генерации фаззинг-оберток для функций библиотеки, который позволяет генерировать нацеленные тесты в

условиях отсутствия информации о контексте использования библиотеки.
2. Предложен метод генерации фаззинг-оберток для функций библиотеки с учетом контекста использования библиотеки в пользовательской программе, который позволяет нацелить генератор только на используемые интерфейсы библиотеки. Качество работы данного метода зависит от качества кода пользовательской программы. Для улучшения метода при обнаружении контекстов необходимо анализировать сложные конструкции, такие как циклы, сложные условия и т. п.

Также в работе был получен практический результат: разработана программа, в которой реализованы предложенные методы автоматизированной генерации фаззинг-оберток. Программа используется в компании «ФОБОС-НТ» для тестирования программного обеспечения.

Разработанная программа состоит из:
- набора инструментов статического анализа;
- модулей для препроцессирования, генерации фаззинг-оберток и сбора и анализа результатов фаззинга.

Научные результаты исследования были опубликованы в сборниках трудов конференций «2021 Ivannikov Memorial Workshop (IVMEM)» и «2022 Ivannikov ISPRAS Open Conference (ISPRAS)». Результаты выносились на обсуждение на конференциях:

1. Открытая конференция ИСП РАН. Москва. 2020.
2. Международная конференция «Иванниковские чтения». Нижний Новгород 2021.
3. Международная техническая конференция по открытой СУБД PostgreSQL «PGConf.Russia». Москва. 2021.
4. Ломоносовские чтения. Научная конференция. Москва. 2022.
5. IX International Conference «Engineering & Telecommunication - En&T-2022». Москва. 2022.
6. Открытая конференция ИСП РАН. Москва. 2022.

Все научные и практические результаты получены лично автором.

Дальнейшее развитие исследований может идти в следующих направлениях:
- поддержка других языков программирования;
- расширение анализа для обнаружения контекста использования библиотеки в циклах и в сложных инструкциях.