Федеральное государственное автономное образовательное учреждение
высшего образования «Московский физико-технический институт
(национальный исследовательский университет)»

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»

Диссертация на соискание учёной
степени кандидата технических наук

Научный руководитель:
Академик РАН, профессор, д.ф.-м.н.
Аветисян Арутюн Ишханович

Москва — 2023

Оглавление

Введение

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 третьей публикации [3] автор описал этапы работы реализованной программы «Futag»: препроцессирование, генерацию и сбор результата.

Во пятой 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 2 приложений. Полный объём диссертации составляет 114 страниц, включая 24 рисунка, 31 листингов и 3 таблиц. Список литературы содержит 74 наименований.

Глава 1. Обзор предметной области

В данной главе проводится краткий обзор работ по теме диссертации, включая технологии фаззинга, вводится понятие фаззинг-обертки. Также проводится анализ особенностей библиотек как объекта тестирования при создании фаззинг-оберток. В конце главы проводится обзор инструментов по генерации фаззинг-оберток для функций библиотек и формулируются основные задачи генерации фаззинг-оберток.

1.1. Методы тестирования библиотек

Тестирование играет важную роль в достижении и оценке качества программного продукта. С одной стороны, качество продуктов улучшается благодаря повторению цикла: “тестирования - обнаружение дефектов - исправление в процессе разработки”. На рисунке 1 иллюстрируется место тестирования в каскадной модели разработки программного обеспечения.

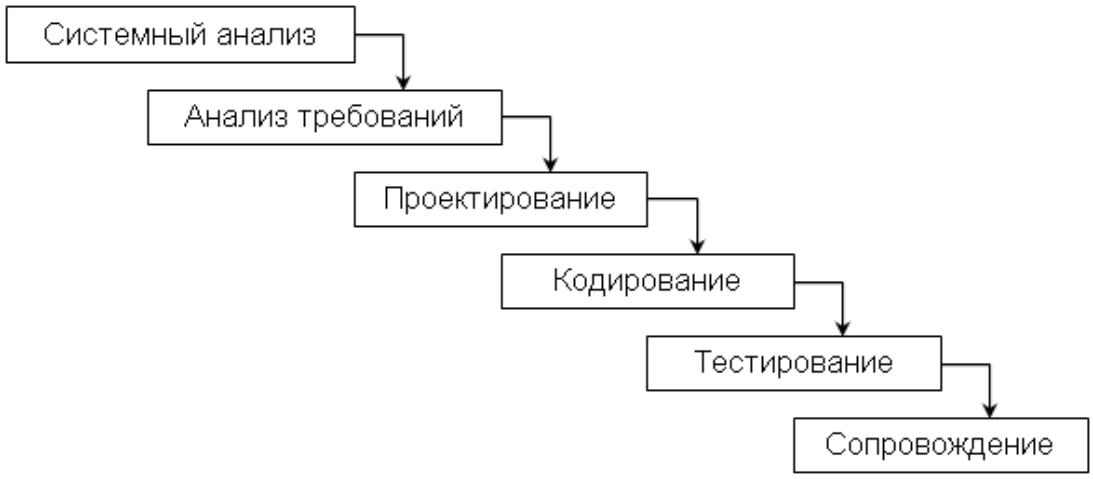

Рисунок 1. Каскадная модель разработки программного обеспечения.

Данная работа посвящена одной из новых технологий тестирования фаззингу, поэтому остановимся на особенностях этой технологии более подробно.

Технология фаззинга:

Технология фаззинга является одним из популярных и эффективных инструментов тестирования [7, 15]. Фаззинг позволяет обнаруживать ошибки, которые могут привести к аварийному завершению программы, утечке памяти, доступу к конфиденциальной информации и другим проблемам безопасности. В последние годы

популярность фаззинга как вида тестирования постоянно возрастает, поскольку современные фаззеры (например, AFL, LibFuzzer) стали более интеллектуальными и способными генерировать более сложные и реалистичные входные данные, что повышает эффективность тестирования.

Основная особенность фаззинга состоит в том, что входные данные, которые передаются на вход тестируемой программы, генерируются автоматически. Первые фаззеры использовали для этого метод случайной генерации, однако на практике полностью случайная генерация входных данных с нуля неэффективна. Современные фаззеры улучшают качество сгенерированных тестовых данных с помощью механизма символьного исполнения и механизма решателей ограничений. Исследователи уделяют большое внимание теме повышения качества генерации входных данных для увеличения покрытия кода. С 2007 в известных изданиях (ACM digital library, Elsevier ScienceDirect, IEEEXplore digital library и т.д.) опубликовано больше 150 публикаций по данной теме [6]. Основными подходами оптимизациями генерации входных данных являются:

- генерация входных данных, базирующая на динамическом анализе инструкций тестируемой программы, выполняемых в оперативной памяти [60];

- генерация входных данных из корректных данных путем последовательного отрицания значений параметров [62];

- генерация сложноструктурированных данных, которая учитывает лексику и грамматику входных данных тестируемой программы и преобразует токены в символы для мутации значений [61];

- генерация входных данных, которые обрабатываются в циклах [68]. По данному методу граф потока управления программы анализируется для обнаружения циклов и ограничений значений переменных до и после выполнения этих циклов. Эти ограничения в дальнейшем используются для создания новых входных данных;

- ускорение генерации входных данных путем удаления бесполезных инструкций при решении уравнений символьного исполнения и выделения зависимостей переменных при решении ограничений [66].

С помощью данной технологии 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 использует фаззинг как один из этапов жизненного цикла разработки программного обеспечения для поиска уязвимостей и повышения стабильности своих продуктов (Microsoft SDL [72]).

Наряду со стремительным развитием информационных технологий повышаются требования к информационной безопасности, разработано много инструментов, платформ для фаззинга, наиболее известные среди них следующие:

- American fuzzy lop [16]: может компилировать исходный код самостоятельно, а затем тестировать его, поэтому этот тип программ называется фаззером серого ящика. Если исходный код недоступен, он эмулируется с помощью QEMU (сокращение от Quick Emulator);
- LibFuzzer [17]: библиотека, являющаяся частью инфраструктуры компилятора LLVM;

  ClusterFuzz [18]: тестовая среда, изначально разработанная Google для тестирования браузера Chrome;
- Peach [19]: высокоавтоматизированное решение для аппаратного и программного обеспечения для фаззинг-тестирования;
- Sulley [20]: набор инструментов на языке сценариев Python; особенно полезно для простых тестовых случаев, таких как генерация случайных данных;
- Powerfuzzer [21]: предоставляет различные сценарии атак, такие как SQL-инъекции; имеет пользовательский веб-интерфейс;
- SAGE [63] SAGAN и JobCenter [64]: – система фаззинга, разработанная и использованная в компании Microsoft. SAGE (Scalable Automated Guided Execution) – инструмент фаззинга белого ящика. Результаты работы SAGE передаются в систему мониторинга SAGAN для анализа. А JobCenter – система управления, которая контролирует развертывание фаззинга на виртуальных машинах;

- ISP-Fuzzer [22, 23, 65]: расширяемая структура фаззинга для нескольких форматов файлов, стандартного ввода, сети, сетевых протоколов;
- Avalanche [69, 70]: инструмент поиска дефектов и уязвимостей с применением параллельного и распределенного динамического анализа программ.

Для обнаружения ошибок в ходе фаззинг-тестирования используются Sanitizers [24] (Санитайзеры - средства “дезинфекции”). Существуют разные санитайзеры для разных целей:

- AddressSanitizer [25, 67]: санитайзер для обнаружения ошибок, связанных с памятью, таких как выход за границы кучи, стека и глобальных переменных, использование после освобождения, использование после возврата и т. д.;
- ThreadSanitizer [26]: санитайзер для обнаружения ошибок типа «race conditions» и «deadlocks»;
- UndefinedBehaviorSanitizer [27]: санитайзер для обнаружения ошибок неопределенного поведения (undefined behavior);
- LeakSanitizer [28]: санитайзер для обнаружения утечки памяти.

В 2008 году ученые из ИСП РАН предложили технологию массового создания тестов работоспособности Азов [58, 59]. Данная технология использует имеющуюся информацию об интерфейсных операциях, о типах параметров и результатов операций. Технология используется для массового создания небольших тестов на работоспособность программных библиотек, реализующих множество интерфейсов стандарта Linux Standard Base (LSB). Предложенная технология достигла большого успеха в случае, когда отсутствует информация о приложениях, использующих интерфейсы, но не может учитывать такую информацию, если она становится доступной. Кроме того, эта технология не работает с фаззерами.

Основная часть перечисленных работ посвящена вопросам генерации входных тестовых данных для фаззинга, однако проблема с генерацией фаззинг-оберток остается актуальной. 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.

Для понимания функций фаззинг-оберток библиотек важно иметь представление об общей схеме фаззинга в данном случае. Фаззинг предполагает выполнение тестируемой программы, которая получает на вход некоторые параметры и при выполнении может обмениваться данными с окружением. В терминах языка Си это означает, что эта программа должна иметь функцию main (главную входную точку программы), с которого начинается выполнение программы. В библиотеках этой функции main нет, поэтому при подготовке к тестированию ее нужно построить.

Из main будут вызываться тестируемые функции, поэтому в ней должны быть предусмотрены все особенности подготовки входных параметров и инициации начального состояния тестирования — это и есть функции-обертки (иногда такие обертки называются тестовыми драйверами). Отличием фаззинг-оберток от обычных тестовых драйверов является способ получения входных тестовых данных, фаззинг-обертка получает их от программы фаззера, точнее, от генератора данных в фаззере.

В целом процессом фаззинг-тестирования управляет выделенный компонент, который получает на вход задание на тестирование, ограничения во времени и описание конфигурации. После этого он провидит инициацию фаззера и других вспомогательных компонентов тестовой системы и в цикле начинает вызывать тестируемую систему, каждый раз обращаясь через входную точку фаззинг-обертки. Обертка в свою очередь запрашивает у фаззера очередной экземпляр тестовых данных, которые вырабатываются генератором данных на основе обратной связи, полученной по данным мониторинга исполнения тестируемых функций (трасса вырабатывается кодом, который предварительно внедряется в тестируемые функции).

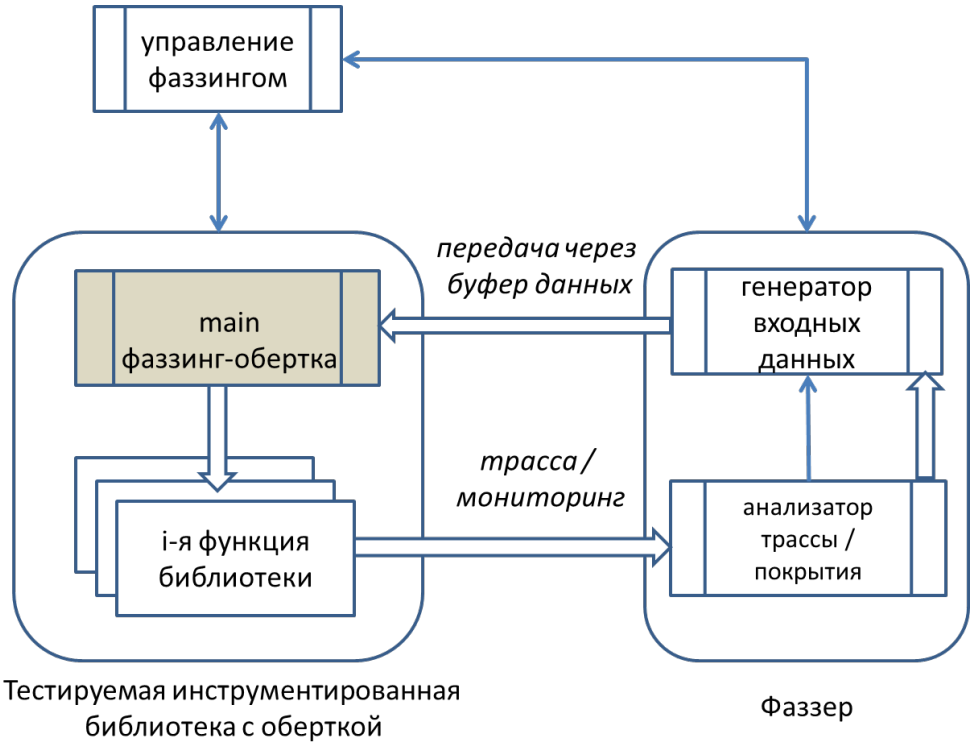

Рисунок 2. Схема организации фаззинга библиотечных функций.

Программа-фаззер является некоторым универсальным инструментом, у нее нет информации о детальном представлении интерфейсных параметров тестируемых функций. Одной из функций фаззинг-обертки является трансформация массива данных, полученных от фаззера, в значения каждого отдельного параметра тестируемой функции. Эта операция получила название «десериализации».

В качестве примера фаззинг-обертки приведем схему обертки, которая строится инструментом LibFuzzer. В листинге 1 приводится фаззинг-обертка LibFuzzer для функции «purple_utf8_salvage» в библиотеке «libpurple». Она включает в себя несколько частей:

- подключение заголовочных файлов: в первой строке подключается заголовочный файл библиотеки «libpurple» «libpurple/util.h», в строках 2–4: подключаются системные заголовочные файлы;
- определение входной функции LibFuzzer: В 6-ой строке определяется входная функция LibFuzzer «LLVMFuzzerTestOneInput», которая принимает на вход буфер «Data» и длину «Size»;

- объявление и инициализация для переменных: в 7-ой строке объявляется переменная «foo» типа «char *» (указатель на символ); в 8-ой строке выделяется память длины «Size+1» для «foo»; в 9-ой строке: копируется область памяти длины «Size» из буфера «Data» для «foo»;
- вызов тестируемой функции: в 11-ой строке вызывается функция «purple_utf8_salvage» с аргументом «foo» и инициализируется переменная «tmp» типа «char *» данным вызовом;
- освобождение выделенной памяти: в строках 12–15 освобождается выделенная память для «tmp».

Листинг 1. Пример фаззинг-обертки.

```
 1 #include "libpurple/util.h"
 2 #include <stdint.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5
 6 extern "C" int LLVMFuzzerTestOneInput(const uint8_t
   *Data, size_t Size){
 7     char *foo;
 8     foo = (char *)malloc(Size + 1);
 9     memcpy(foo, Data, Size);
10     foo[Size] = 0;
11     char *tmp = purple_utf8_salvage(foo);
12     if (tmp == 0) {
13         free(foo);
14         return 0;
15     }
16     return 0;
17 }
```

Целью данной работы является автоматизация тестирования программных библиотек при помощи генерации таких подобных фаззинг-оберток и сбора и анализа результатов фаззинга, поэтому рассмотрим особенности библиотек как объектов тестирования.

1.2. Библиотека

Как уже упоминалось, в отличие от обычной программы библиотека может не содержать точек входа, что требует

дополнительных действий для проведения тестирования. При традиционном тестировании перед проведением тестирования библиотеки разработчик должен: проанализировать исходный код и доступную документацию библиотеки, написать тестовые программы с различными наборами входных данных для подлежащих тестированию функций библиотеки, скомпилировать тестовые программы, собрать результат их выполнения, выполнить анализ полученных данных. Этот процесс называется модульным тестированием (unit-testing) [14], и это важный подход к уменьшению количества дефектов и повышению качества программного обеспечения. Однако библиотека может состоять из сотен функций, десятков определенных структур данных. Программная система может использовать несколько библиотек (в том числе, разделяемых библиотек), поэтому практическая возможность вручную протестировать все задействуемые библиотеки, как правило, отсутствует.

1.2.1. Структура и характеристики

Библиотека — это пакет исходного кода, предназначенный для повторного использования во многих программах [8]. Обычно исходный код библиотеки на языках Си и Си++ состоит из нескольких частей:

- файлы, содержащие информацию о библиотеке, такие как версия, контактная информация, документация, и т.д.;
- скрипты сборки — это скрипты, которые используются для компиляции библиотеки;
- Модули на каком-либо языке программирования, которые реализуют функциональности библиотеки. Каждый модуль может состоять из заголовочных файлов (файлы с расширением h) и исходных файлов («*.c», «*.cpp»). Заголовочные файлы содержат объявления о программных сущностях библиотеки: вновь определенные типы данных, структуры, классы, функции и т.д. Исходные файлы содержат определения функций, методов и т.д.

Существует два типа библиотек: статические и динамические библиотеки [9].

Статическая библиотека (иногда называемая архивом) состоит из подпрограмм, которые скомпилированы и связаны непосредственно с

основной программой. Когда компилируется программа, использующая статическую библиотеку, все функции статической библиотеки, используемые программой, становятся частью исполняемого файла. В Windows статические библиотеки обычно имеют расширение .lib, а в Linux — расширение «.a» (архив). Одним из преимуществ статических библиотек является то, что нужно распространять только исполняемый файл, чтобы пользователи могли запускать программу. Поскольку библиотека становится частью программы, это гарантирует, что с программой всегда будет использоваться правильная версия библиотеки. Кроме того, поскольку статические библиотеки становятся частью программы, их можно использовать точно так же, как функции, которые написаны для основной программы. С другой стороны, поскольку копия библиотеки становится частью каждого исполняемого файла, который ее использует, это может занять много места. Статические библиотеки также не могут быть легко обновлены — обновление библиотеки требует замены всего исполняемого файла.

Динамическая библиотека (также называемая разделяемой библиотекой) состоит из подпрограмм, которые загружаются в приложение во время выполнения. При компиляции программы, использующую динамическую библиотеку, эта библиотека не становится частью исполняемого файла — она остается отдельным объектом. В Windows динамические библиотеки обычно имеют расширение «.dll» (библиотека динамической компоновки, библиотека динамической компоновки), а в Linux — расширение .so (общий объект, общий объект). Одним из преимуществ динамических библиотек является то, что многие программы могут совместно использовать одну копию библиотеки, что экономит место. Возможно, самым большим преимуществом является то, что динамическую библиотеку можно обновить до более новой версии, не заменяя все исполняемые файлы, которые ее используют.

На рисунке 3 иллюстрируются процесс компиляции исходного кода библиотеки.

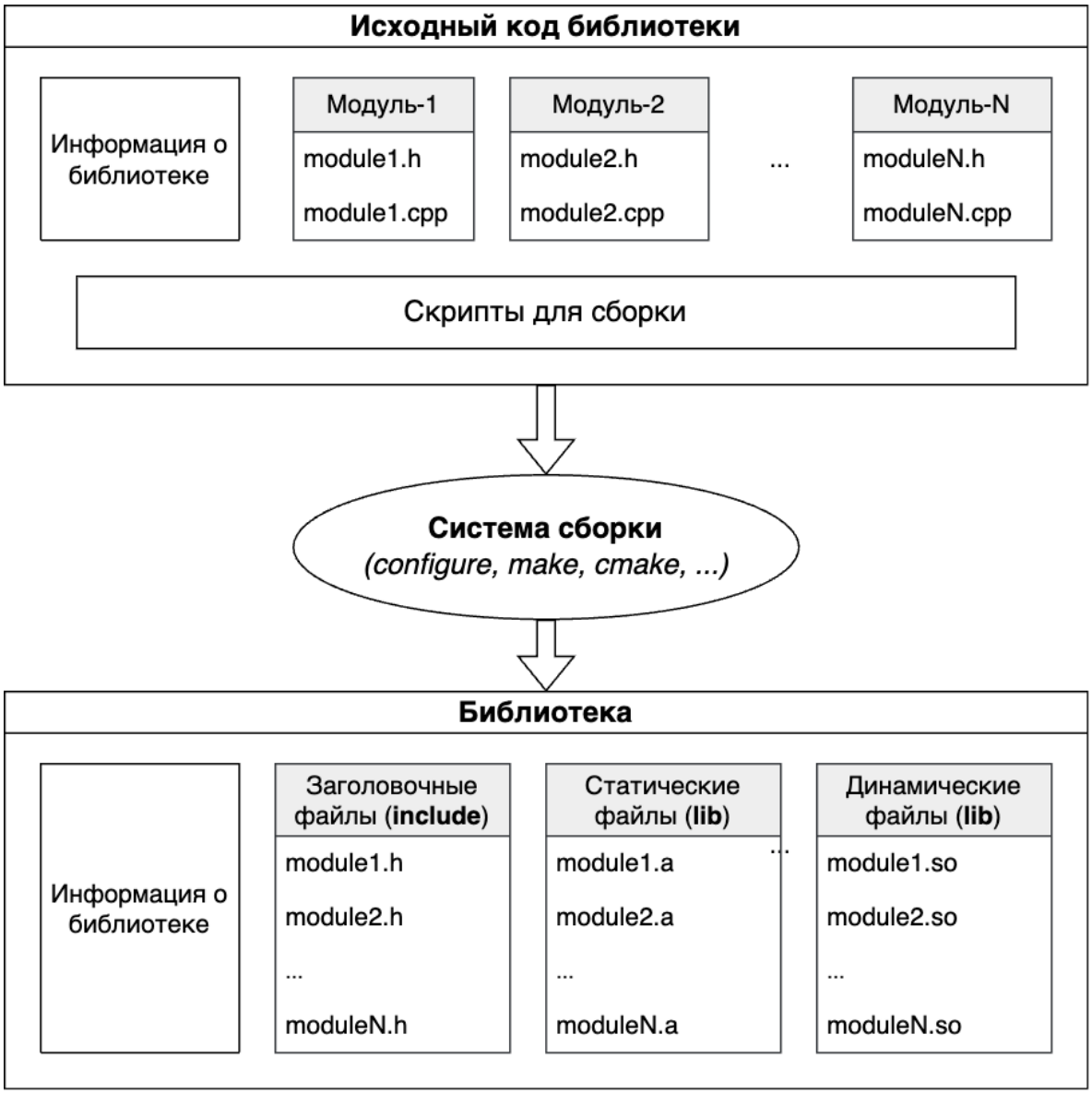

Рисунок 3. Процесс компиляции исходного кода библиотеки.

1.2.2. Использование и компиляция

Далее в работе будет рассматриваться метод генерации фаззинг-оберток в случае, когда можно учитывать контекст программ-приложений, пользующихся тестируемой библиотекой. Это позволяет построить более эффективные в рамках конкретной программной системы тесты. Анализ контекста сводится к выявлению программных сущностей, которые используются при генерации фаззинг-оберток. Для этого предлагается общая схема структуры программы-приложения.

Пользовательская программа взаимодействует с библиотекой через функции, находящиеся в доступной области видимости, и типы данных, известные как программный интерфейс (API). Функции программного

интерфейса могут вызывать другие функции библиотеки для обработки полученных данных. На рисунке 4 отображаются взаимодействия между библиотекой и пользовательской программой.

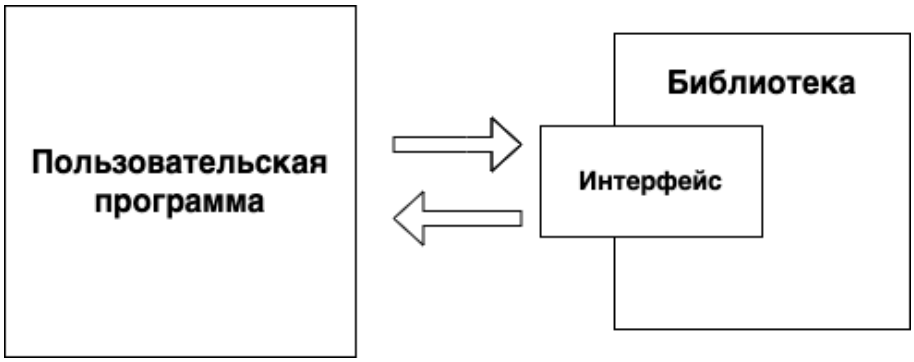

Рисунок 4. Взаимодействия между библиотекой и программой.

Для использования библиотеки в своей программе разработчики должны изучить документацию, структуру, программный интерфейс библиотеки для подключения и формирования необходимых вызовов. Основные этапы для подключения (использования) библиотеки в пользовательских программах в языках Си и Си++:

- подключить заголовочные файлы, которые содержат определения необходимых программных сущностей;
- объявить переменные, структуры, классы;
- вызвать необходимые функции, методы;
- скомпилировать код со статическим или динамическим файлом библиотеки.

В листинге 2 демонстрируется пример использования библиотеки «pugixml» в файле «loadfile.cpp»:

- строки 1: подключаются заголовочный файл системной библиотеки iostream и заголовочный файл «pugixml.hpp» библиотеки pugixml, содержащий определения функций, типов данных и т.д.;
- 5-я строка: объявляется объект «doc» класса «pugi::xml_document»;
- 6-я строка: объявляется объект «result» класса «pugi::xml_parse_result» и присваивается результат чтения файла «tree.xml» методом «doc.load_file()». Данный метод является API библиотеки «pugixml»;

– 7-я строка: выводит в стандартный выход значения обработанного объекта XML.

Листинг 2. Простой пример использования библиотеки pugixml

```
 1 #include "pugixml.hpp"
 2 #include <iostream >
 3
 4 int main(){
 5     pugi::xml_document doc;
 6     pugi::xml_parse_result result =
   doc.load_file("tree.xml");
 7     std::cout << "Load result: " <<
   result.description() << ", mesh name: " <<
   doc.child("mesh").attribute("name").value() <<
   std::endl;
 8     return 0;
 9 }
```

Файл «loadfile.cpp» на следующем этапе компилируется следующей командой:

```
g++ loadfile.cpp –Iinclude –o loadfile lib/pugixml.a
```

– «g++»: компилятор;
– «loadfile.cpp»: исходный файл;
– «-Iinclude»: путь для поиска заголовочных файлов;
– «-o loadfile»: указать название выходного файла;
– «lib/pugixml.a»: статическая библиотека pugixml для компоновки.

В связи с ростом объема программного обеспечения растет и количество сущностей, которые необходимо брать в рассмотрение при генерации фаззинг-обертки. Тем самым выполнение анализа приложений усложняется, и продуктивность разработчиков снижается [10] [11]. В таблице 1 приводится количество программных сущностей некоторых библиотек, которые необходимо принимать во внимание при генерации фаззинг-оберток:

– LoC: количество строк кода;
– Кол. функций: количество определенных функций;
– Кол. записей: количество определенных структур, объединений или классов;

– Кол. новых типов: количество новых типов, определенных ключевым словом «typedef».

Таблица 1. Оценка количества программных сущностей в популярных библиотеках.

Библиотека	LoC	Кол. функций	Кол. записей	Кол. новых типов
libjson-c	55.114	264	21	16
libpostgres	114.868	589	40	137
curl	509.256	1414	196	167
openssl	1.407.092	8744	577	1177
pugixml	144.706	637	79	58
libopus	163.602	538	22	71
libmpeg2	41.556	289	18	40

Как видно из таблицы 1, число программных сущностей, которые приходится анализировать и описывать при разработке фаззинг-оберток, и, соответственно, трудоемкость разработки фаззинг-оберток достаточно велики, поэтому актуальность и практическая значимость заявленной темы исследования высокая.

1.3 Обзор инструментов генерации фаззинг-оберток

В данном разделе описываются альтернативные инструменты по генерации фаззинг-оберток для функций в библиотеке.

1.3.1. Инструмент FUDGE

В 2019 году компания Google представила инструмент FUDGE [29], который помогает генерировать фаззинг-обертки для библиотек. FUDGE предоставляет графический пользовательский интерфейс, который упрощает процесс создания, управления и анализа результатов работы фаззинг-оберток. FUDGE способен генерировать тесты на основе результата предыдущих тестов. Работа инструмента иллюстрируется на рисунке 5.

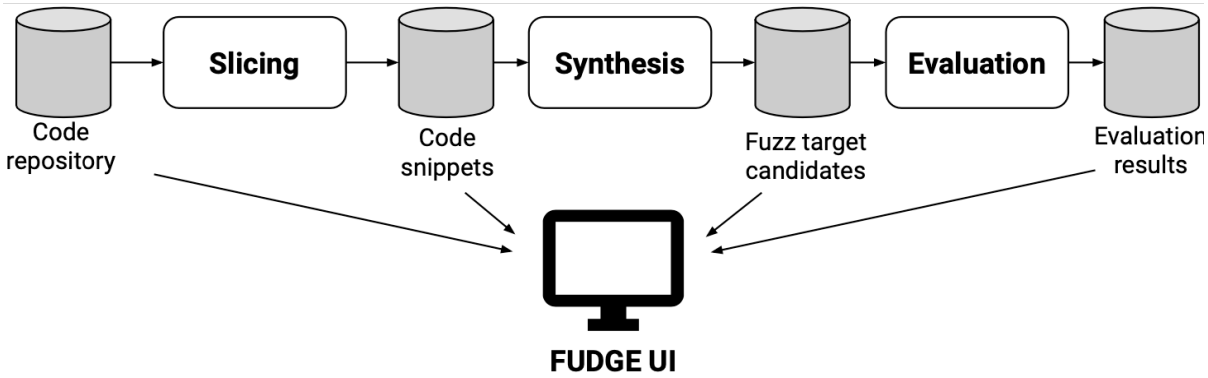

Рисунок 5. Схема работы инструмента FUDGE.

Процесс генерации фаззинг-обертки в FUDGE состоит из нескольких шагов:

- слайсинг (агл. slicing) - FUDGE сканирует код библиотеки, ищет и извлекает из функций примеры использования функций целевой библиотеки. Анализируя абстрактное синтаксическое дерево (АСД) программы, FUDGE извлекает только те части дерева, которые относятся к целевой библиотеке. Функция выбирается для анализа, если она содержит хотя бы один вызов библиотечной функции, подходящей для фаззинга: например, один из аргументов которой является буфером. Вызовы функций целевой библиотеки выбираются в качестве начальных операторов в этих функциях.
- анализ зависимостей по данным и по управлению: выделяются зависимые операторы. Процесс синтаксического анализа помечает выражения, содержащие неизвестные символы, определенные вне функции, такие как параметр функции, глобальная переменная, функция, определенная в другой единице трансляции, или непримитивные типы, определенные вне целевой библиотеки. Следующим шагом является синтез кода, который заменяет помеченные операторы в извлеченных частях кода новыми выражениями с использованием сопоставления с образцом, которые могут содержать данные из буфера, сгенерированного фаззером.
- генерация фаззинг-обертки и её запуск с ограничением по времени выполнения. Это позволяет отсеять бесполезные обертки, вызовы из которых немедленно прекращают работу, потому что входные данные содержали тривиальные ошибки. В некоторых

случаях FUDGE работает хорошо, но в больших проектах генерирует много некорректных оберток, которые нужно редактировать вручную.

К сожалению, доступ к инструменту FUDGE закрыт. Кроме того, в условиях отсутствия кода пользовательских программ FUDGE не может генерировать фаззинг-обертки для функций целевой библиотеки.

1.3.2. Инструмент FuzzGen

Во второй половине 2021 года на конференции «30th USENIX Security Symposium» представлен автоматический генератор фаззинг-оберток FuzzGen [30]. Работа FuzzGen иллюстрируется на рисунке 6 и разбивается на 3 этапа:

- на первом этапе FuzzGen сканирует базу кода и извлекает взаимодействия с API целевой библиотеки;
- на втором этапе строится специальный граф Abstract API Dependence Graph (A2DG), который охватывает все взаимодействия с API. A2DG - направленный граф, схожий с графом потока управления, который отображает последовательности правильных вызовов API. Вершины графа – вызовы функций целевой библиотеки, ребра – передача управления. В ребрах дополнительно содержится информация о возможных диапазонах значений параметров, что улучшает итоговую производительность. A2DG позволяет выразить сложные зависимости между компонентами API. Данная структура выделяет функции, которые вызываются первыми, последними, и как функции зависят друг от друга;
- на третьем этапе работы FuzzGen создается обёртка на основе имеющегося графа A2DG. Вместо генерации большого числа оберток для каждого графа FuzzGen создает одну более общую обертку, которая воспроизводит некоторую реалистичную последовательность вызовов.

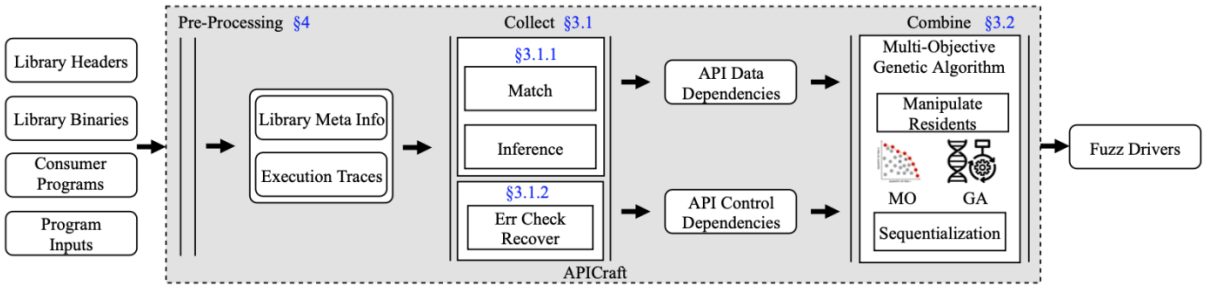

Рисунок 6. Работа инструмента FuzzGen.

По сравнению с FUDGE, FuzzGen создает более объёмные, общие фаззеры, которые способны выразить сложные взаимодействия с API по сравнению с более короткими последовательностями, получаемыми в FUDGE. Также FuzzGen использует для анализа несколько библиотек, использующих целевую, а FUDGE только одну.

FuzzGen полагается на полноту примеров использования и входных данных целевой библиотеки, без них он не способен создавать эффективные обёртки для фаззинга. Имеются также негативные отзывы об использовании FuzzGen, так в 2020 году эксперты проекта тестирования библиотеки libtorrent [31], который поддерживается фондом «Mozilla´s Secure Open Source Fund», не смогли сгенерировать фаззинг-обертки с помощью данного инструмента.

1.3.3. Инструмент IntelliGen

Инструмент «IntelliGen» [32] был представлен в конце 2021 года. Данный инструмент анализирует все функции библиотеки и присваивает каждой из них “приоритет уязвимости”, который зависит от использования в функции потенциально опасных операций, таких как разыменование указателя и вызов функции работы с памятью (например, memcpy). Далее функции сортируются по приоритету и выбираются наиболее уязвимые. Данные функции считаются входными, и для них выполняется анализ. На следующем этапе производится синтез значений аргументов данных функций в процессе выполнения:

- для аргумента скалярных типов значение берется из подпоследовательности байтов из буфера фаззера;
- для указателей выделяется память без инициализации, и указателю присваивается адрес этой памяти;
- для массивов и структур выполняются предыдущие действия рекурсивно для каждого элемента.

Код выбранных функций затем инструментируется, чтобы реализовать принцип ленивой инициализации памяти, когда выделенная память будет инициализирована, только если она позже используется. Intelligen анализирует низкоуровневое промежуточное представление кода (IR) целевой функции, находит из сравнений подходящие значения для параметров функции и использует эту информацию при генерации параметров. Также, как и в FUDGE, полученные обёртки запускаются на ограниченное время. Таким образом, в случае ошибки отфильтровываются обертки, содержащие тривиальные ошибки. На рисунке 7 показан процесс определения входных функций инструмента IntelliGen.

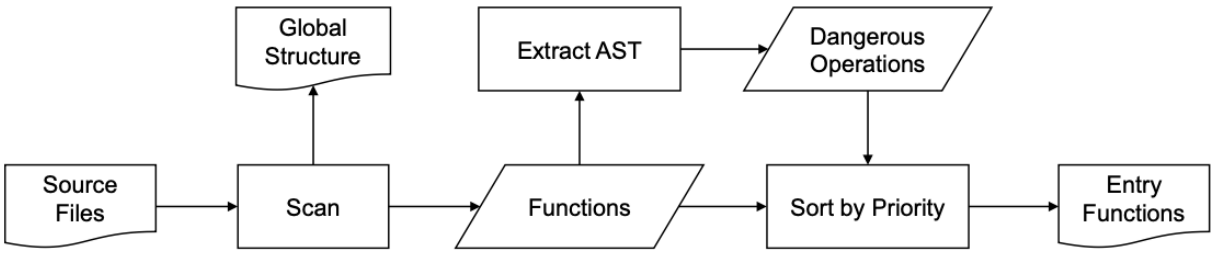

Рисунок 7. Процесс определения входных функций инструмента IntelliGen.

Принцип работы IntelliGen более универсальный по сравнению с сопоставлением закономерностей параметров в FUDGE и анализом функций API в существующих тестовых случаях в FuzzGen, IntelliGen может определить уязвимые функции без необходимости в дополнительной информации. Из описанных ранее методов IntelliGen наиболее эффективный: в среднем показывает лучшие результаты по количеству покрытых блоков и числу найденных путей. Однако для аргументов неполных типов (см. п. 3.1.3 ниже) инструмент IntelliGen не представляет способ синтеза значений. Аргументы этих типов часто используются в функциях, обрабатывающих данные, полученные из пользовательской программы. До сих пор доступ к данному инструменту тоже закрыт.

1.4. Выводы по главе 1

Обзор работ по теме диссертации показал, что основные исследования были посвящены методам генерации тестовых данных для вызова тестируемых функций. Автоматизированная генерация фаззинг-оберток сводится к тому, что нужно автоматизировать:

– анализ интерфейсов и интерфейсных данных в тестируемых библиотеках (определения функций, методов, определения структур, перечислений и т. д.), выделение необходимых программных сущностей и их взаимосвязи;
- организацию передачи мутационных данных из буфера фаззера аргументам тестируемой библиотеки;
- генерацию фаззинг-оберток для функции в условиях отсутствия информации о контексте использования библиотеки;
- генерацию фаззинг-оберток для функций с учетом контекста использования.

Решение этих задач и является предметом данной диссертации.

Глава 2. Анализ исходного кода

В данной главе описывается процесс анализа исходного кода библиотек для выделения программных сущностей и их взаимосвязей в программном коде, необходимых для конструирования вызовов функций, подлежащих фаззинг-тестированию и для подготовки данных для генераторов фаззинг-оберток при разных условиях:

- в случае анализа исходного кода тестируемой библиотеки — это определения сущностей, взаимосвязи между ними и параметры компиляции;
- в случае анализа пользовательской программы это контекст использования тестируемой библиотеки, который включает переменные и вызовы функций данной библиотеки.

В следующих разделах приведено краткое описание методов и инструментов статического анализа (раздел 2.1); описание особенностей процесса конструирования вызовов функций библиотеки в пользовательской программе (раздел 2.2) и описание схемы интеграции статического анализа для целей генерации фаззинг-оберток в процесс компиляции и сборки программного продукта (разделе 2.3).

2.1 Статический анализ и инструменты статического анализа

2.1.2. Статический анализ

Для анализа исходного кода библиотеки используются средства статического анализа [33]. Статический анализ – это процесс анализа исходного кода без его выполнения, при котором проверятся наличие синтаксических ошибок, возможных дефектов и других опасных конструкций [34].

Для анализа код программы часто преобразуется в АСД. АСД генерируется при компиляции или интерпретации программного кода и представляет собой дерево, в котором каждый узел представляет собой конструкцию языка программирования, а каждый лист представляет собой литеральное значение, константу или переменную. АСД позволяет анализатору программы обойти всю структуру программы и произвести анализ каждого элемента кода, используя информацию о его типе, значениях и связях с другими элементами программы.

На рисунке 8 иллюстрируется пример АСД для функции, реализующей алгоритм Евклида.

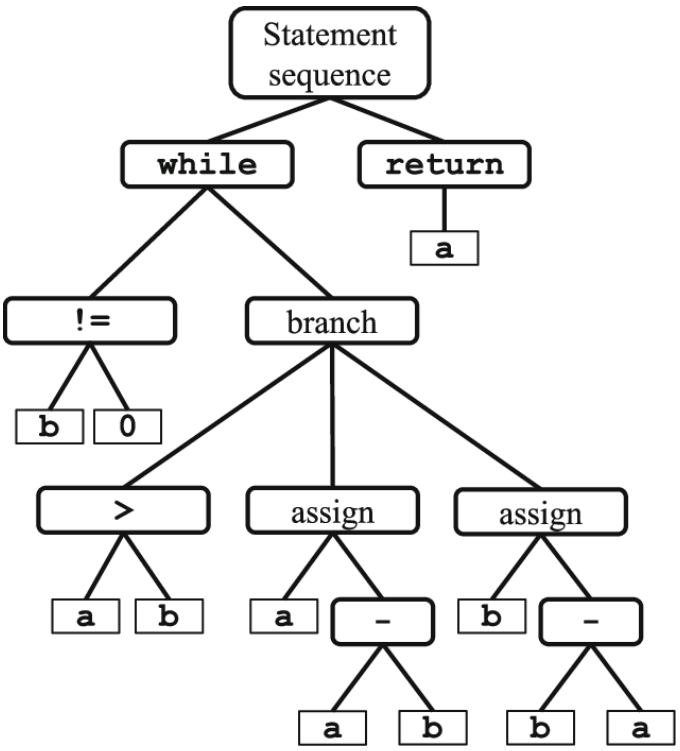

Рисунок 8. Пример АСД для алгоритма Евклида.

2.1.3 Инструменты статического анализа

Существуют разные инструменты статического анализа, 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**» [36] – это инструмент для анализа кода, разработанный компанией Semmle, которую приобрела Microsoft в 2019 году. CodeQL позволяет производить статический анализ кода, выявлять ошибки, уязвимости и другие проблемы в исходном коде приложений. Основной функцией CodeQL является контекстно-чувствительный анализ кода, позволяющий находить дефекты и уязвимости, которые могут оставаться незамеченными при использовании других инструментов. В контекте данного исследования CodeQL может рассматриваться как один из инструментов статического анализа, поскольку он позволяет создавать собственные запросы для анализа кода. Важно, что CodeQL может быть использован для анализа кода на разных языках программирования, включая Си++, C#, Java, JavaScript, Python и другие. Он интегрируется со средами разработки, такими как Visual Studio Code и GitHub Codespaces, что позволяет проводить анализ кода непосредственно в процессе разработки.

Анализ, который выполняет CodeQL, состоит из трех этапов:

- подготовка исходного кода путем создания базы данных CodeQL (кодовой базы);
- выполнение запросов CodeQL к этой базе данных;
- интерпретация результатов этих запросов.

Создание базы знаний

Собственно анализу исходного кода при использовании CodeQL предшествует преобразование текстового представления исходного кода в так называемую кодовую базу, внутреннее представление, для которого удобно формировать запросы похожие на SQL-запросы. В кодовой базе содержится полное иерархическое представление этого кода, включая абстрактное синтаксическое дерево, граф потока данных и граф потока управления. Функции (методы) CodeQL позволяют писать запросы к кодовой базе, используя принципы объектно-ориентированного программирования. Кодовая база CodeQL также включает в себя хранилище с исходным кодом и его зависимостями. Этот же исходный код потом используется для визуализации результатов выполнения запросов.

Выполнение запроса

После создания базы данных CodeQL для нее выполняется один или несколько запросов. Запросы CodeQL пишутся на специально разработанном объектно-ориентированном языке запросов, именуемом QL.

Тестировщики могут запускать запросы, извлеченные из репозитория CodeQL (или пользовательские запросы, которые тестировщики написали самостоятельно), используя CodeQL для расширения VS Code или CodeQL CLI.

Инструмент CodeQL можно использовать по бесплатной лицензии, но при большом количестве запросов к кодовой базе нужно приобретать коммерческую лицензию.

Clang static analyzer [37] (Clang статический анализатор – Clang SA) - 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 этого статического анализатора можно создавать граф потока данных и анализировать его состояния на различных участках кода. Он поддерживает анализ как Си/Си++, так и Objective-C кода и способен обрабатывать как отдельные файлы, так и целые проекты. Благодаря тому, что Clang SA входит в состав проекта LLVM, он может запускаться во время компиляции для анализа, и устанавливать другое ПО не требуется.

Clang SA предоставляет детализированные отчеты об ошибках и предупреждениях, обнаруженных в коде, а также может быть интегрирован в редакторы кода или системы сборки, чтобы обеспечить автоматический анализ кода в процессе разработки.

Clang SA является мощным инструментом для повышения качества и надежности кода и широко используется в индустрии программного обеспечения. Он также доступен как открытое программное обеспечение и может быть адаптирован и настроен для конкретных нужд проекта. На рисунке 9 приведена схема работы Clang SA во время компиляции.

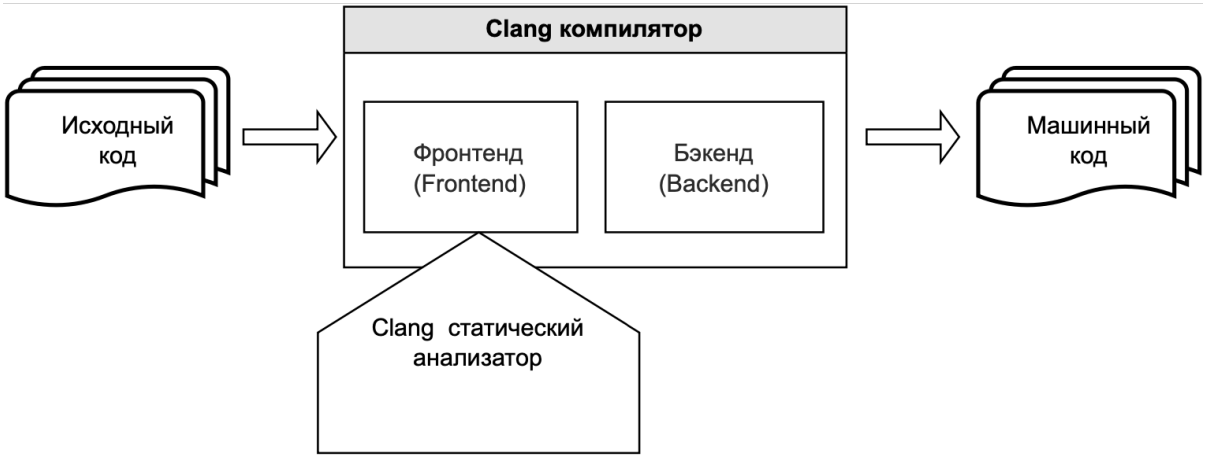

Рисунок 9. Схема работы Clang SA.

В листинге 3 иллюстрируется сериализованное представление АСД для следующей простой функции f(x):

```
int f(int x) {
    int result = (x / 42);
    return result;
}
```

Где:

- FunctionDecl - означает определение функции, в этом случае это функция f с возвращаемым типом «int» и принимает 1 аргумент типа «int»;
- ParmVarDecl - означает определение параметров, в этом случае это аргумент «x»;
- CompoundStmt - означает составной оператор, который включает в себя другие операторы;
- DeclStmt - означает оператор объявления;
- VarDecl - означает определение переменной;
- ParentExpr - означает возвращаемое выражение;
- BinaryOperator - означает бинарный оператор, в данном случае это оператор «/»;
- ImplicitCastExpr - означает неявное преобразование для выражения;
- DeclRefExpr - означает переданное выражение для определения;
- IntegerLiteral - означает целую константу со значением 42;

- ReturnStmt - означает возвращаемый оператор.

Листинг 3. Промежуточное представление кода в Clang SA.

```
‘-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f ’int (int)’
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x ’int’
  ‘-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | ‘-VarDecl 0x5aeac10 <col:3, col:23> result ’int’
    |   ‘-ParentExpr 0x5aeacf0 <col:16, col:23> ’int’
    |     ‘-BinaryOperator 0x5aeacc8 <col:17, col:21> ’int’
’/’
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> ’int’
<LValueToRValue>
    |       | ‘-DeclRefExpr 0x5aeac68 <col:17> ’int’ lvalue
ParmVar 0x5aeaa90 ’x’ ’int’
    |       ‘-IntegerLiteral 0x5aeac90 <col:21> ’int’ 42
    ‘-ReturnStmt 0x5aead68 <line:3:3, col:10>
      ‘-ImplicitCastExpr 0x5aead50 <col:10> ’int’
<LValueToRValue>
        ‘-DeclRefExpr 0x5aead28 <col:10> ’int’ lvalue Var 0
x5aeac10 ’result’ ’int’
```

Clang SA предоставляет собой набор из двух инструментов «ASTVisitor» (АСД-инспектор) и ASTMatcher (АСД-обработчик), эти инструменты могут запускаться с помощью модуля «Clang checker» (Clang-детектор).

АСД-инспектор — это инструмент, который позволяет обходить АСД и выполнять определенные действия при посещении узлов АСД. Когда Clang-детектор обнаруживает узел, который соответствует определенному типу, он вызывает соответствующий АСД-инспектор, который выполняет необходимые действия.

АСД-обработчик — это инструмент, который позволяет осуществлять поиск в АСД с помощью заданных пользователем шаблонов. АСД-обработчик предоставляет гибкий интерфейс для создания шаблонов с использованием DSL (Domain Specific Language - язык предметной области), который позволяет выразить сложные критерии поиска в компактной и понятной форме. После запуска поиска по шаблонам, если находятся нужные узлы, для обработки может запускаться функция обратного вызова MatchCallback. Пример АСД-

обработчика, который выполняет поиск некоторых программных сущностей, приведен в листинге 4:

- binaryOperator: поиск бинарного оператора;
- isAssignmentOperator: ограничение, что найденые бинарные операторы являются оператором присваивания;
- hasLHS: ограничение для левой части найденых бинарных операторов;
- hasRHS: ограничение для правой части найденых бинарных операторов;
- hasDescendant: ограничение для дочерних элементов;
- hasName: ограничение по имени сущности;
- bind(): отметка для найденных сущностей.

Листинг 4. Пример шаблона запроса АСД-обработчика.

```
binaryOperator(
    isAssignmentOperator(),
        hasLHS(
           declRefExpr(to(
                varDecl(
           hasName(arg->getDecl()->getNameAsString())))),
        hasRHS(
           hasDescendant(
               declRefExpr())))
.bind("FutagBinOpArg");
```

Разработчики могут создать свои Clang-детекторы, АСД-инспекторы или АСД-обработчики для анализа кода по своим требованиям.

В результате сравнительного анализа в данной работе был выбран Clang SA как инструмент анализа исходного кода библиотеки.

2.2. Схема процесса конструирования вызовов функций библиотеки

Библиотеки, которые часто становятся объектом фаззинг-тестирования, решают типичные задачи обработки таких “специальных” структур данных, как: видео, звуковые записи, JSON-формат данных и т.д. (именно через такие структуры данных часто осуществляются кибератаки). 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 этих структур. В библиотеках на языке Си++ специальные данные определяются с помощью класса (class), конструкторы которых отвечают за инициализацию значения объекта данного класса. Соответственно, 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 конструирования специальных данных с помощью функций в первой группе. Аргументы этих функций имеют сложный тип.

На рисунке 10 отображается процесс конструирования вызова функция простого типа в пользовательской программе: генератор для конструирования вызова обращается к определению функции и получает информацию о типе данных аргументов и о возвращаемом типе.

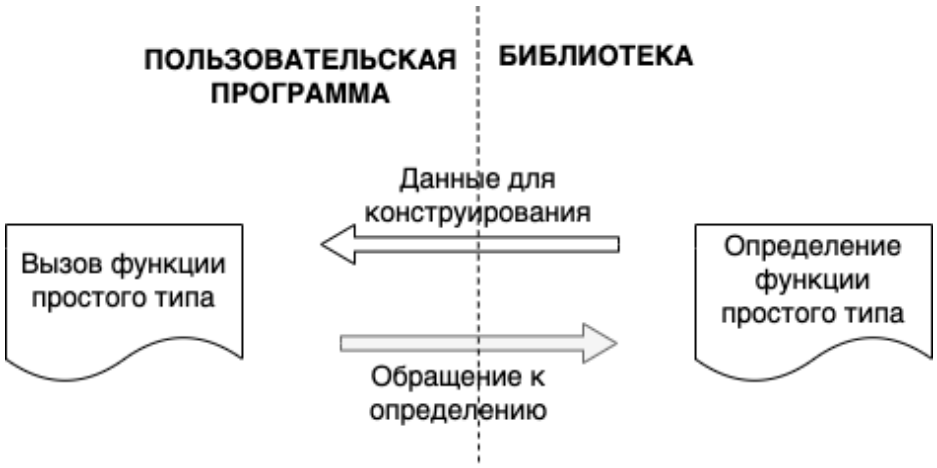

Рисунок 10. Процесс конструирования вызова функции простого типа.

На рисунке 11 отображается процесс конструирования вызова функции 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 на третий этап.
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 возвращеат значение нужного сложного типа.
6. В последнем шаге генерируется вызов функции сложного типа.

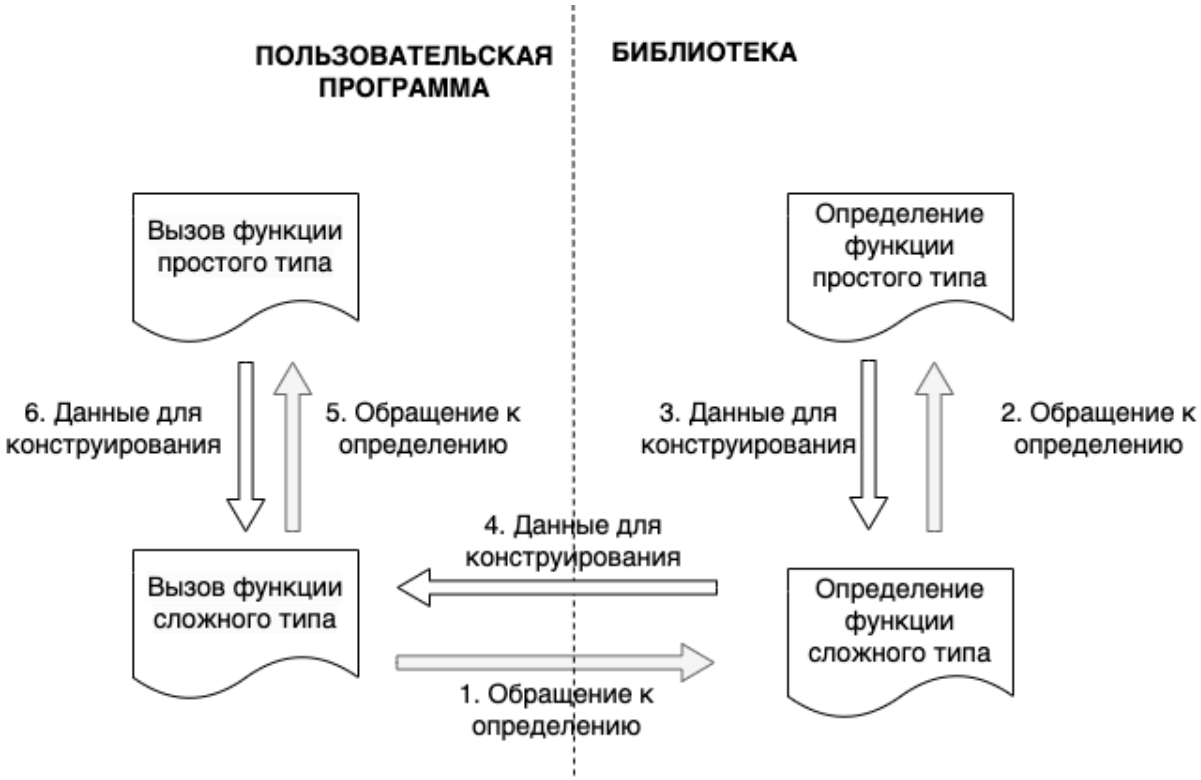

Рисунок 11. Процесс конструирования вызова функции сложного типа.

Можно сделать вывод о том, что для конструирования вызова функции простого и сложного типа необходимо проанализировать исходный код библиотеки для обнаружения определений функций,

сущностей и взаимосвязи между ними. Данный анализ описывается в следующем разделе «Интеграция статического анализа в процесс компиляции ПО» и выполняется с помощью АСД-инспекторов.

2.3. Интеграция статического анализа в процесс компиляции ПО

Как показано на рисунке 3, библиотеки обычно компилируются с помощью системы сборки. В данном разделе описывается процесс автоматизации статического анализа во время компиляции для получения характеристик и свойств кода. Также описываются разработанные инструменты для статического анализа исходного кода.

Система сборки – это вспомогательная система, которая отвечает за:

- определение структуры библиотеки и среды компиляции;
- очистку временных файлов;
- конфигурацию проекта;
- проверку кода до компиляции;
- сборку проекта;
- тестирование – запуск тестов определенным образом;
- упаковку – создание архивов и инсталляторов.

Основным средством системы сборки является утилита make, которая поддерживается в большинстве широко используемых инструментов: Automake, CMake, imake, qmake, nmake, wmake, Apache Ant, Apache Maven, OpenMake Meister, Gradle.

В системе Linux распространено используются следующие популярные системы сборки [38]: make, cmake, ninja, bazel, и др. В рамках данной работы рассматривается работа систем make и cmake. Схема интеграции статического анализа в процесс компиляции показывается на рисунке 12 и состоит из 4 этапов:

1. Первый этап: обнаружение и запуск конфигурационного скрипта: на вход данного этапа поступает исходный код библиотеки. 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. Второй этап: запуск статического 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 программными сущностями кода; для задачи генерации фаззинг-оберток с учетом контекста использования тестируемой библиотеки запускаются инструменты для обнаружения контекста использования. Инструмент Clang SA может запускаться с помощью утилиты «scan-build»;
3. Третий этап: компиляция с санитайзерами: при необходимости исходный код компилируется с санитайзерами для обнаружения дефектов в процессе фаззинга;
4. Четвертый этап: сбор результатов анализа – на данном этапе собираются результаты статического анализа (этап 2) и параметры компиляции (этап 3).

Рисунок 12. Процесс интеграции статического анализа в процесс компиляции программного продукта.

Для анализа программных сущностей в узлах АСД были разработаны разные типы АСД-инспекторов для разных типов сущностей:

- для функции: АСД-инспектор собирает следующую информацию: название, возвращаемый тип, хэш-значение, хэш-значение родительской сущности, список аргументов, каждый из которых определяется своим названием и типом данных. Также собирается информация об области видимости функции. Область видимости одной функции – это область, в которой функция может вызываться. В языке Си++ функция может определяться в классах (публичная, приватная или защищенная), в пространстве имен, как глобальная или статическая функция. В языке Си, для того

чтобы ограничить доступ к конкретной функции в единице компиляции, в ее объявлении используется ключевое слово «static». Определение области видимости помогает выбрать правильные функции для фаззинга;

- для записи: под термином «запись» в АСД понимаются структура, класс или объединение. Для каждой записи собирается следующая информация: название, хэш-значение и список полей, каждое из которых определяется своим названием и типом данных;
- для перечисления (определяется ключевым словом «enum»): собирается информация о названии, хэш-значении и всех его константах;
- для новых определяемых типов данных: в языках Си и Си++ новые типы могут определяться с помощью ключевого слова «typedef» путем композиции фундаментальных типов. Для этих типов АСД-инспектор собирает информацию об их названии, хэш-значении и базовом типе, для которого определяется новое название.

Также был разработан собственный АСД-инспектор для анализа каждой единицы трансляции. Данный АСД-инспектор собирает следующую информацию:

- использованный компилятор (для Си или Си++);
- список включенных заголовочных файлов;
- заданные пути для поиска заголовочных файлов (с помощью опции «-I»);
- пользовательские параметры компиляции.

Вся собранная информация о коде библиотеки АСД-инспекторами сохраняется в JSON-файлах, которые образуют базу знаний о тестируемой библиотеке.

Глава 3. Метод генерации фаззинг-оберток для функций библиотеки в условиях отсутствия информации о контексте использования

Данный метод дает возможность генерации фаззинг-обертки для функций библиотеки даже в условиях отсутствия контекста использования тестируемой библиотеки. Перед тем как приступить к описанию собственно метода генерации фаззинг-оберток, сначала опишем предварительные действия, необходимые для работы метода, а именно, исследование способов инициализации переменных разных типов и способов десериализации буфера фаззера для передачи мутационных данных в тестируемую функцию. Затем описывается собственно предложенный метод и способы повышения корректности генерации фаззинг-оберток.

Как показано в листинге 1, фаззинг-обертка играет роль пользовательской программы, использующей тестируемую библиотеку. Фаззинг-обертка вызывает тестируемую функцию и передает ее аргументам мутационные данные буфера фаззера. В случае, когда аргумент функции имеет простой тип данных, выполняется десериализация буфера данных для их передачи аргументам функции. В случае, когда аргумент функции имеет тип данных, который не известен пользовательской программе, необходимо искать способ сконструировать значение данных такого типа при помощи обращений к функциям программного интерфейса библиотеки.

При конструировании значений каждого типа данных необходимо знать размер области памяти, куда они будут помещаться. Далее задача определения размеров памяти рассматриваются сначала для простых (фундаментальных) типов данных, затем для более сложных (структур и массивов) и вопросы, которые возникают при десериализации буфера фаззера.

3.1 Типы данных и способы инициализации переменных

3.1.1 Фундаментальные типы

Ниже перечисляются некоторые фундаментальные типы данных языков Си и Си++.

- **char** – символ, целочисленный, самый маленький из возможных адресуемых типов, размер 1 байт;
- **bool** – логическое значение;
- **int** – целое число, обычно отражающее естественный размер целых чисел на компьютере;
- **float** - вещественное число одинарной точности;
- **double** - вещественное число двойной точности.

К этим основным типам можно применить ряд квалификаторов: signed, unsigned, short, long. Квалификатор signed или unsigned может быть применен к типу char или любому целому числу. Квалификатор unsigned обозначает, что значение типа всегда положительно или равно нулю. Квалификаторы short и long определяют короткий или длинный размер целого числа, например:

```
char c;
unsigned int positive_number;
short int small_integer;
long int large_integer;
```

Квалификатор long может также быть применен к типу double.

Размер типа данных в языках Си и Си++ зависит от среды компиляции. Размер «int» в 16-битных системах составляет 2 байта, а в 32-битных или 64-битных системах – 4 байта. Для определения размера типа используется оператор «sizeof».

```
size_t size_of_int = sizeof(int);
```

3.1.2 Производные типы

Производные типы определяются из фундаментальных типов, дальше перечисляются популярные типы, размер которых можно явно определить.

Перечисление

Перечисление — это тип данных, значение которого ограничено диапазоном значений, заданных явным количеством констант целого числа и размером перечисления является размер того типа целого числа. Пример определения перечисления «color» со значениями «red», «yellow», «green», «blue» иллюстрируется в следующем: «color» является новым типом данных и другие переменные могут определяться этим типом. Нужно отметить, что в большинстве случаев значение константы перечисления определяется ее порядком. Например «red» = 0, «yellow» = 1, «green» = 2, «blue» = 3 – это свойство позволяет передавать мутационное значение аргументу типа перечисления.

```
enum color {red, yellow, green, blue};
enum color c = yellow;
int num = green;
```

Строки

В языке Си строка представляет собой массив символов (char *), который заканчивается нулевым символом ‘\0’. Размер строки равен количеству символов в массиве.

В языке Си++ строка определяется классом «string», и значение строки «string» в языке Си++ может инициализироваться из строки в стиле языка Си. Пример определения строки иллюстрируется в следующем:

```
char * s1 = "this is string";
std::string s2(s1);
```

Массив

Пример определения массива целых чисел и массива символов приводится ниже: массив «a» состоит из 4 целых чисел, поэтому его размер равен 16 байтам (4 элемента x 4 байта - размер типа int в 32-битных или 64-битных системах), а размер массива «symbols» равен 8.

```
int a[4];
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
char symbols [8];
```

Структура

Пример определения структуры приводится ниже. В данном примере структура «myStruct» состоит из 2 полей: целое число «num» размера 4 байтов и символ «c» размера 1 байт.

```
// создание структуры
struct myStruct {
    int num;
    char c;
};
int main() {
    //объявление переменной "s1" типа структуры "myStruct"
    struct myStruct s1;
    // присваивание значения для полей s1
    s1.num = 5;
    s1.c = 'X';
    return 0;
}
```

3.1.3 Неполный тип данных

В языках Си и Си++ существуют неполные типы данных [40]. Неполный тип — это тип, который описывает идентификатор, но не содержит информации, необходимой для определения его размера. Неполным типом может быть:

- структура, поля которой еще не указаны или не известны;
- объединение, поля которого еще не указаны или не известны;
- массив, размер которого еще не указан.

Например, в листинге 5 приводится код функции «send_exchange_report» библиотеки «libstorj», которая использует библиотеку «json-c» для сохранения и оформления данных, функция «json_object_object_add» принимает 3 аргумента:

- struct json_object *jso;
- const char *key;
- struct json_object *val.

То есть значения аргументов функций из библиотеки «libstorj» нельзя инициализировать явным значением типа «struct json_object», так как структура данного типа скрыта от пользователя. Поэтому разработчики

должны инициализировать эти аргументы с помощью функций «json_object_new_object()» (4–я строка), «json_object_new_string()» (5–я строка) или «json_object_new_int64()» (7–я строка) – эти функции входят в программный интерфейс библиотеки «json-c».

Листинг 5. Пример использования библиотеки json-c в пользовательской программе

```
1  static void send_exchange_report(uv_work_t *work) {
2      shard_send_report_t *req = work->data;
3      storj_download_state_t *state = req->state;
4      struct json_object *body = json_object_new_object();
5      json_object_object_add(body,"dataHash",
   json_object_new_string(req->report->data_hash));
6      json_object_object_add(body,"token",
   json_object_new_string (req->report->token));
7      json_object_object_add(body,"exchangeStart",
   json_object_new_int64(req->report->start));
8      json_object_object_add(body,"exchangeEnd",
   json_object_new_int64(req->report->end));
9      json_object_object_add(body,"exchangeResultCode",
   json_object_new_int(req->report->code));
10     json_object_object_add(body,"exchangeResultMessage",
   json_object_new_string(req->report->message));
11     int status_code = 0;
12     // there should be an empty object in response
13     struct json_object *response = NULL;
14     int request_status = fetch_json(req->http_options,
   req-> options, "POST", "/reports/exchanges", body, true,
   &response
   , &status_code);
15     if (request_status) {
16         state->log->warn(state->env->log_options,state-
   >handle, "Send exchange report error: %i",
   request_status);
17     }
18     req->status_code = status_code;
19     // free all memory for body and response
20     if (response) {
21         json_object_put(response);
22     }
23     json_object_put(body);
24 }
```

3.2. Десериализация буфера фаззера для передачи аргументам тестируемой функции

В данном разделе описываются способы десериализации буфера фаззера для передачи мутационных данных в тестируемую функцию.

За основу способа десериализации в данном случае взят подход, предложенный в статье «A Case Study on Automated Fuzz Target Generation for Large Codebases» [41]. Обычно фаззер передает мутационные данные на вход тестируемой функции через буфер, а в случае, когда тестируемая функция имеет несколько параметров разных типов, нужна десериализация для передачи. На приведенном примере (рисунок 13) рассматривается случай, когда в сгенерированной фаззинг-обертке имеются 3 аргумента разных типов: логическое значение $bool$ размера 1, целое число int размера 4 и строка $char$ размера 5 из 3 элементов – тогда минимальная длина буфера равна 10.

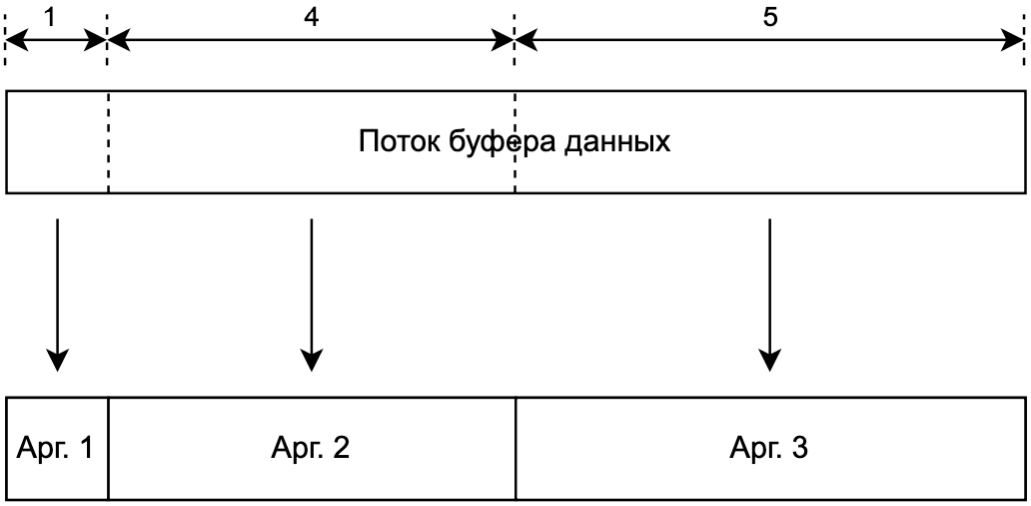

Рисунок 13. Пример разделения буфера фаззера.

3.2.1. Десериализация буфера для переменной фундаментального типа

В листинге 6 показан код инициализации аргумента фундаментального типа. Переменная фундаментального типа объявляется, затем копируется значение части буфера в ее область памяти. Размер части буфера равен размеру фундаментального типа аргумента.

Листинг 6. Десериализация буфера для переменной фундаментального типа

```
//GEN_BUILTIN
int num;
memcpy(&num, buffer_part1, sizeof(int));
char c;
memcpy(&c, buffer_part2, sizeof(char));
```

3.2.2. Десериализация буфера для переменной типа перечисления (enum)

Перечисление состоит из набора констант, в данной работе рассматриваются классические перечисления, где значения идут подряд. При фаззинге переменной типа перечисления передача значений фаззером за пределами данного типа нецелесообразна, поэтому мутацией значения для перечисления в данной работе является подбор значения только в наборе заданных значений. В листинге 7 показан код инициализации аргумента «e_encoding» типа перечисления «pugi::xml_encoding»:

- 2-я строка: объявляется переменная «e_encoding_enum_index» целого типа, которая используется для индексации констант данного типа; – 3-я строка: копируются мутационные данные размера целого числа из буфера переменной «e_encoding_enum_index»;
- 4-я строка: рассчитает остаток деления «e_encoding_enum_index» на количество констант (10), это значение является индексом значения перечисления. Переменная «e_encoding» получается путем явного приведения (static_cast) от индекса «e_encoding_enum_index».

Листинг 7. Десериализация буфера для переменной типа перечисления.

```
1 // GEN_ENUM
2 unsigned int e_encoding_enum_index;
3 memcpy(&e_encoding_enum_index ,Futag_pos,sizeof(unsigned
  int));
4 enum pugi::xml_encoding e_encoding = static_cast <enum
  pugi::xml_encoding>(e_encoding_enum_index % 10);
```

3.2.3. Десериализация буфера для переменной строкового типа

Строка – это последовательность ненулевых символов, которая заканчивается нулевым символом (\0). В листинге 8 показан код инициализации аргумента «str_1_str0» строкового типа «char *» длины «dyn_cstring_size».

- 2-я строка: объявляется указатель на символ «rstr_1_str0» и выделяется область памяти размера «dyn_cstring_size + 1» для него;
- 3-я строка: инициализируется выделенная область нулевыми значениями с помощью функции «memset»;
- 4-я строка: копируются мутационные данные длины dyn_cstring_size из буфера в данную область памяти;
- 5-я строка: создается переменная «str_1_str0» строкового типа «const char *» и указывается на адрес «rstr_1_str0». В итоге «str_1_str0» имеет длину dyn_cstring_size и заканчивается нулевым байтом.

Листинг 8. Десериализация буфера для переменной строкового типа.

```
1 //GEN_CSTRING
2 char * rstr_1_str0 = (char *) malloc((dyn_cstring_size +
  1)* sizeof(char));
3 memset(rstr_1_str0,   0,   dyn_cstring_size   +   1);
4 memcpy(rstr_1_str0, Futag_pos, dyn_cstring_size);
5 const char * str_1_str0 = rstr_1_str0;
```

3.2.5 Десериализация буфера для указателя и константы

Для указаталей фаззер генерирует значения не адресов, а указываемых переменных, поэтому для инициализации аргумента типа указателя сначала создается переменная со значением, полученным из буфера, затем объявляется аргумент, значение которого указывает на адрес созданной переменной. Инициализация для константы проходит сходным образом. В листинге 9 показан код инициализации для указателя и константы.

Листинг 9. Десериализация буфера для указателя и константы.

```
//GEN_POINTER
int ref_a;
```

```
memcpy(&ref_a, buffer_part1, sizeof(int));
int * a = &ref_a;

//GEN_CONSTANT
int ref_b;
memcpy(&ref_b, buffer_part2, sizeof(int));
const int b = ref_b;
```

3.3 Метод генерации фаззинг-оберток для функций библиотеки в условиях отсутствия информации о контексте использования

Данный метод позволяет генерировать фаззинг-обертки для всех доступных функций библиотеки в условиях отсутствия контекстов ее использования. Схема данного метода представлена на рисунке 14.

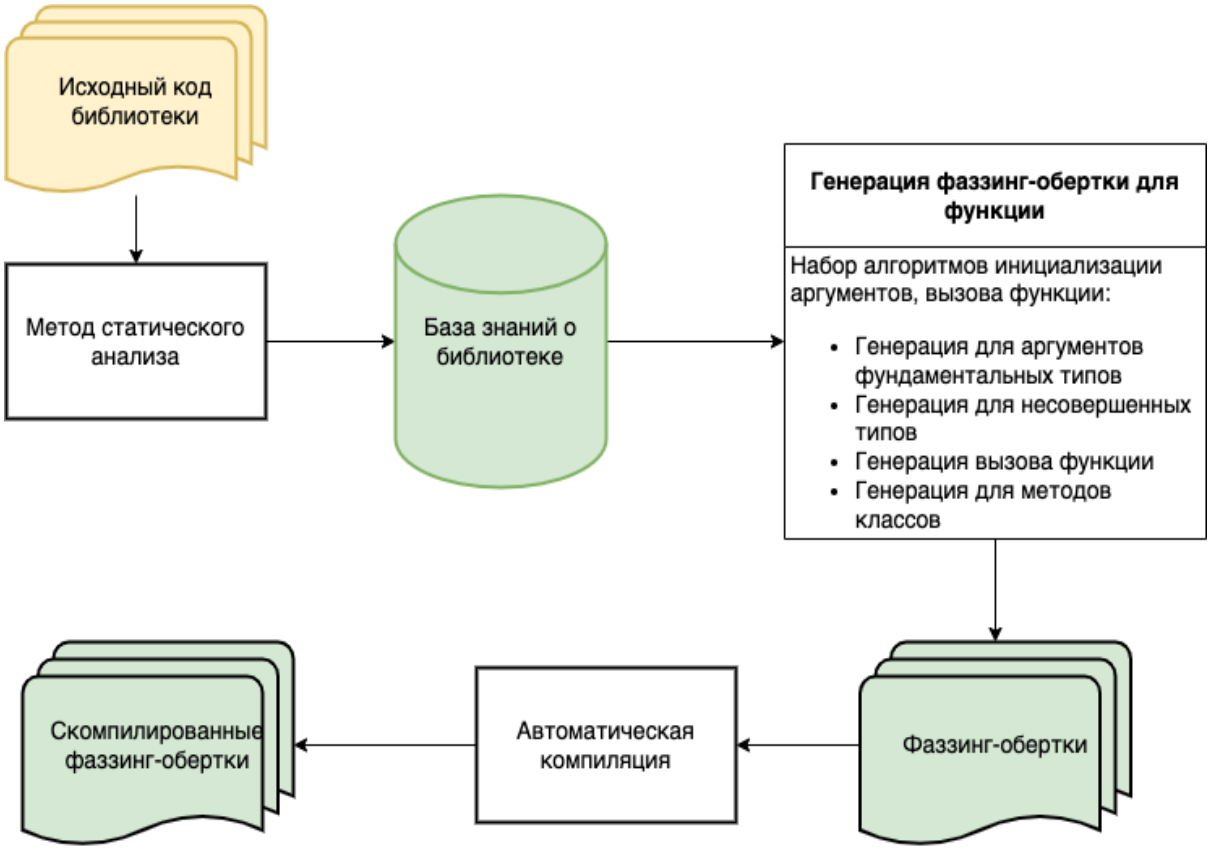

Рисунок 14. Схема метода генерации фаззинг-оберток для функций библиотеки в условиях отсутствия контекста использования.

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 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 иллюстрируется на рисунке 16:

1. Проходит итерация всех функций в базе знаний о библиотеке. Для каждой функции 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 артифактов и т.д.

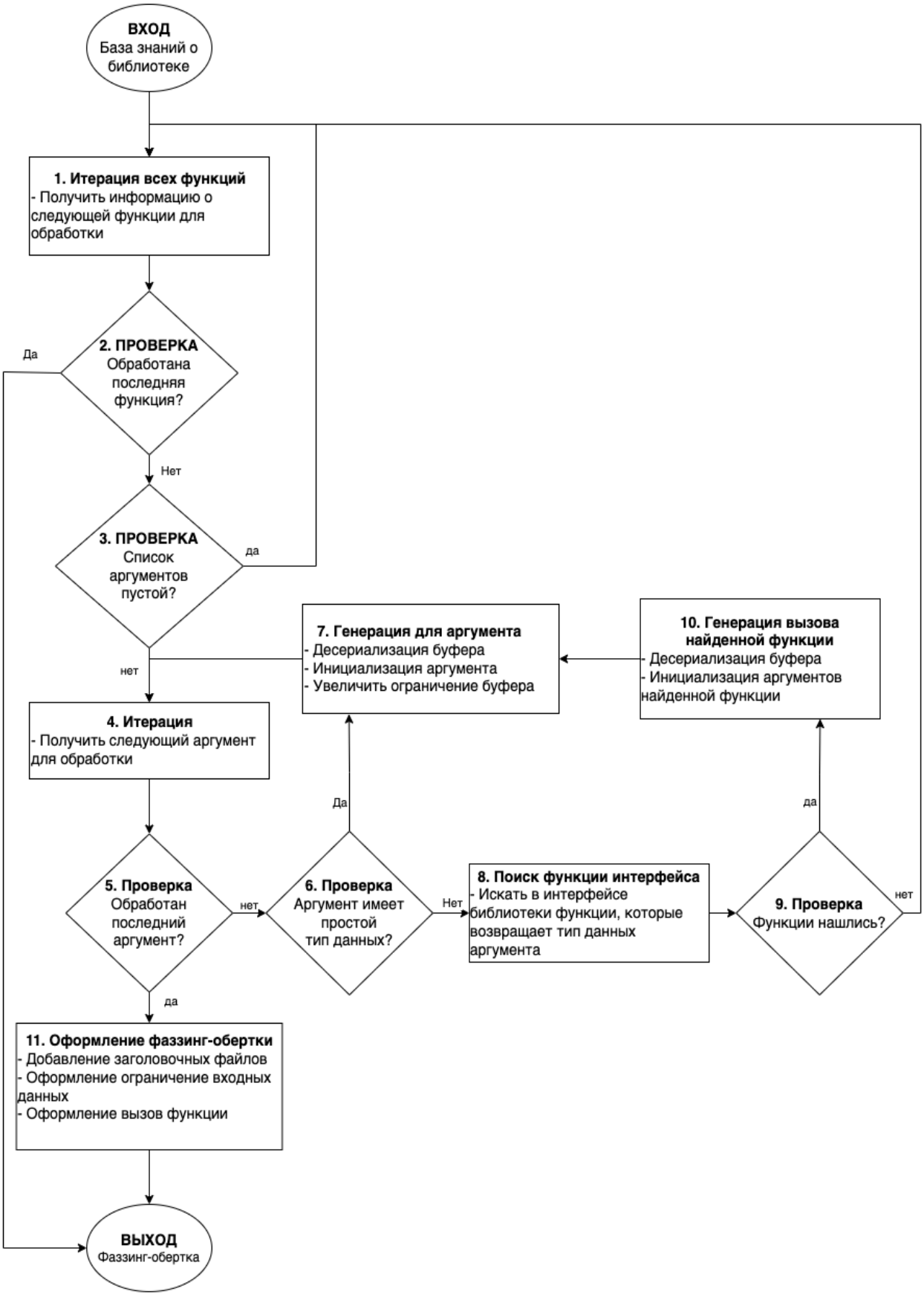

Рисунок 15. Алгоритм построения фаззинг-оберток для функций библиотеки в условиях отсутствия информации о контекстах использования.

В листинге 10 приведен пример результата генерации фаззинг-обертки для функции «json_tokener_parse»:

- В строках 1–3: приводятся заголовочные файлы, которые Futag нашел при статическом анализе.
- В строке 5: объявляется функция «LLVMFuzzerTestOneInput» для платформы LibFuzzer.
- В строке 6: задаётся ограничение буфера. Функция «json_tokener_parse» принимает только один аргумент строкового типа, размер которой динамически зависит от размера входного буфера фаззера.
- В строке 7: индикатор «Futag_pos» создается и указывает на начало буфера.
- В строках 8–12: показывается генерация строки с динамической длиной «rstr_str». Эта строка принимает мутационные данные из фаззера функцией «memcpy» в строке 12.
- В строках 13: объявляется указатель строки «str_str» типа «const char *», которая указывает на адрес строки «rstr_str».
- В строке 14: индикатор «Futag_pos» указывает на следующую часть буфера.
- В строке 17: формируется вызов тестируемой функции «json_tokener_parse».
- В строках 19–22: освобождение выделенной памяти. Это важная часть при работе с указателями и памятью [40].

Листинг 10. Результата генерации фаззинг-обертки для функции «json_tokener_parse».

```
1 #include "json_object.h"
2 #include "json_tokener.h"
3 #include "json_util.h"
4
5 extern "C" int LLVMFuzzerTestOneInput(uint8_t * Fuzz_Data , size_t Fuzz_Size){
6     if (Fuzz_Size < 0) return 0;
7     uint8_t * Futag_pos = Fuzz_Data;
8     size_t dyn_cstring_size = Fuzz_Size;
9     //GEN_CSTRING1
```

```
10      char * rstr_str = (char *) malloc((dyn_cstring_size
    + 1)* sizeof(char));
11      memset(rstr_str, 0, dyn_cstring_size + 1);
12      memcpy(rstr_str, Futag_pos, dyn_cstring_size);
13      const char * str_str = rstr_str;
14      Futag_pos += dyn_cstring_size;
15
16      //FUNCTION_CALL
17      json_tokener_parse(str_str );
18      // FREE
19      if (rstr_str) {
20          free(rstr_str);
21          rstr_str = NULL;
22      }
23      return 0;
24  }
```

3.4 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 (Листинг 11). 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.

Листинг 11. Пример ситуации, когда знаний о типе данных недостаточно.

```
struct json_object *json_object_from_file(const char *fname)
{
    struct json_object *obj;
    int fd;
    if ((fd = open(fname, O_RDONLY)) < 0){
        return NULL;
    }
    obj = json_object_from_fd(fd);
    close(fd);

}
```

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 АСД-обработчики и внешние АСД-обработчики.

Внутренний АСД-обработчик – это АСД-обработчик, который анализирует код в теле тестируемой функции и обнаруживает, какие системные вызовы используют аргументы данной функции. Если такие вызовы найдутся, АСД-обработчик анализирует место аргумента в списке параметров вызова и выводит способ использования аргумента. Код реализации внутреннего АСД-обработчика показан в листинге 12:

- Строки 2-5: объявляется запрос «MatchFuncCall», который ищет вызов (CallExpr) в теле определения фукнции с условиями: вызов имеет любой аргумент (hasAnyArgument), название которого есть название анализируемого аргумента тестируемой функции.
- 6-я строка: объявляется АСД-обработчик «Finder».

- 7-я строка: объявляется узел «curr_node» - это тело определения функции (func->getBody ()).
- 8-я строка: объявляется обратный вызов, который вызывается при нахождении подходящего вызова АСД-обработчиком «Finder». Если найденный вызов входит в список специальных системных вызовов то, обратный вызов определяет места нахождения анализируемого аргумента в списке параметров этого вызова и способ его инициализации.
- 9-я строка: добавляется запрос «MatchFuncCall» в АСД-обработчик «Finder».
- 10-я строка: запускается поиска вызовов.

Листинг 12. Реализации внутреннего АСД-обработчика.

```
 1 // Match all callExprs, where one of the arguments has
   the same name
 2 auto MatchFuncCall = callExpr(hasAnyArgument(
 3     hasDescendant(declRefExpr(to(
 4         varDecl(hasName(param->getName()))))).bind("
 5             FutagCalledFuncArgument"))));
 6 MatchFinder Finder;
 7 Stmt *curr_node = func->getBody();
 8 Futag::FutagArgUsageDeterminer
   func_call_callback{param_info , Mgr , func_body , func };
 9 Finder.addMatcher(MatchFuncCall, &func_call_callback);
10 Finder.FutagMatchAST(Mgr.getASTContext(), func_body);
```

Внешний АСД-обработчик — это АСД-обработчик, который запускается в каждом определении функций библиотеки и ищет вызов других функций. При нахождении вызова запускается обратный вызов, чтобы исследовать, как аргументы найденного вызова инициализировались. Код реализации внешнего АСД-обработчика показан в листинге 13:

- строки 3-6: объявляется запрос «MatchCallExpr», который ищет вызов (CallExpr) других функций в теле данного определения.
- 8-я строка: объявляется АСД-обработчик «Finder».
- 9-я строка: объявляется узел «curr_node» — это тело данного определения функции (func->getBody ()).

- 10-я строка: объявляется обратный вызов, который вызывается при нахождении вызова АСД-обработчиком «Finder». Этот обратный вызов отвечает за определение способа инициализации параметров (с помощью бинарного оператора).
- 11-я строка: добавляется запрос «MatchCallExpr» в АСД-обработчик «Finder».
- 12-я строка: запускается поиска вызовов методом «FutagMatchAST».

Листинг 13. Реализация внешнего АСД-обработчика.

```
 1 // Match all CallExpression of target function
 2
 3 auto MatchCallExpr =
 4 callExpr(callee(functionDecl(unless(
 5     isExpansionInSystemHeader()))))
 6     .bind("FutagCalledFunc");
 7
 8 MatchFinder Finder;
 9 Stmt *curr_node = func->getBody();
10 Futag::FutagMatchCallExprCallBack
   target_func_call_callback{call_context_json, Mgr,
   curr_node, func};
11 Finder.addMatcher(MatchCallExpr,&target_func_call_callb
   ack);
12 Finder.FutagMatchAST(Mgr.getASTContext(), curr_node);
```

В итоге с помощью представленного метода в условиях отсутствия пользовательских программ удается генерировать фаззинг-обертки для функций библиотеки. Пример результата метода приводится в листинге 14 – фаззинг-обертка для функции «json_tokener_parse_ex». Данная функция принимает 3 аргумента:

- аргумент «s_tok» типа «struct json_tokener *» (строка 13); 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 19);

– аргумент «sz_len» типа «int», которому присваивается длина строки «str_str» (строка 22).

Листинг 14. Фаззинг-обертка для функции «json_ tokener_parse_ex».

```
1  // Match all CallExpression of target function
2
3  extern "C" int LLVMFuzzerTestOneInput(uint8_t *
   Fuzz_Data, size_t Fuzz_Size){
4      if (Fuzz_Size < 0) return 0;
5      size_t dyn_cstring_buffer = Fuzz_Size;
6      //generate random array of dynamic string sizes
7      size_t dyn_cstring_size[1];
8      dyn_cstring_size[0] = dyn_cstring_buffer;
9      //end of generation random array of dynamic string
   sizes
10     uint8_t * Futag_pos = Fuzz_Data;
11
12     //GEN_VAR_FUNCTION
13     struct json_tokener * s__tok = json_tokener_new();
14     //GEN_CSTRING1
15     char * rstr_str = (char *)
   malloc((dyn_cstring_size[0] + 1)* sizeof(char));
16     memset(rstr_str, 0, dyn_cstring_size[0] + 1);
17     memcpy(rstr_str, Futag_pos, dyn_cstring_size[0]);
18     Futag_pos += dyn_cstring_size[0];
19     const char * str_str = rstr_str;
20
21     //GEN_SIZE
22     int sz_len = (int) dyn_cstring_size[0];
23
24     //FUNCTION_CALL
25     json_tokener_parse_ex(s__tok, str_str, sz_len );
26     //FREE
27     if (rstr_str) {
28         free(rstr_str);
29         rstr_str = NULL;
30     }
31     return 0;
   }
```

Глава 4. Метод генерации фаззинг оберток для функций библиотеки с учетом контекста использования в пользовательской программе

В исходном коде тестируемой библиотеки часто отсутствует контекст вызовов функций этих библиотек. Для того, чтобы сконструировать корректный вызов (для чего, возможно, необходимо выполнить некоторые предварительные действия, в частности вызов других функций), тестировщикам необходимо прочитать документацию и примеры использования. В данной главе дается более строгое определение контекста, рассматриваются задачи анализа потока данных, решение которых необходимо для построения контекста, сам способ автоматизированного определения контекста вызовов функций тестируемой библиотеки при помощи анализа программ, где есть вызовы тестируемых функций, и метод генерации фаззинг-оберток для тестирования функций с учетом построенных контекстов.

4.1 Контекст вызовов функций библиотеки в пользовательских программах

Контекст использования (вызова) функции — это упорядоченная последовательность операторов (в том числе вызовов других функций) в программе, которые взаимодействуют между собой посредством передачи данных через значения переменных с общей целью получения значений входных параметров в данном вызове функции. Понятие контекста, как оно вводится здесь, близко к понятию program slicing [73]. Mark Weiser предложил термин «program slice» - это подпрограмма, которая получается из исходной программы путем удаления в ней лишних инструкций, гарантируя поведение по заданному критерию. По предложному методу Mark Weiser критерий для слайсинга - это набор <i, V>, где:

- i – место проведения слайсинга;
- V – это список переменных в исходной программе, над которыми анализируются взаимосвязи по графу потока управления и по потоку данных, данный список нужно задавать до проведения слайсинга.

Отличие контекстов от слайсинга будет выяснено в дальнейшем.

Один контекст использования библиотеки «json-c» в другой программе показан в листинге 5, который включает в себя:

- 4-я строка: создается json-объект «body» вызовом json_object_new_object();
- 5-я строка: добавляется в объект «body» элемент с названием "dataHash" и значением, полученным из вызова «json_object_new_string(req->report->data_hash)»;
- 6-я строка: добавляется в объект «body» элемент с названием "token" и значением, полученным из вызова «json_object_new_string(req->report->token)»;
- 7-я строка: добавляется в объект «body» элемент с названием "exchangeStart" и значением, полученным из вызова «json_object_new_int64(req->report->start)»;
- 8-я строка: добавляется в объект «body» элемент с названием "exchangeEnd" и значением, полученным из вызова «json_object_new_int64(req->report->end)»;
- 9-я строка: добавляется в объект «body» элемент с названием "exchangeResultCode" и значением, полученным из вызова «json_object_new_int(req->report->code)»;
- 10-я строка: добавляется в объект «body» элемент с названием "exchangeResultMessage" и значением, полученным из вызова «json_object_new_string(req->report->message)»;
- 23-я строка: уменьшается счетчик ссылок для объекта «obj» функций «json_object_put».

Контекст позволяет узнать, какие входные данные, поступившие из пользовательской программы, обрабатываются тестируемой библиотекой. Контекст использования тестируемой библиотеки 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 в нужном формате.

- слайсинговые инструкции — это инструкции, которые модифицируют инициализируемые переменные.

4.2 Анализ потока управления и потока данных

Методической основой реализации анализа потока управления и потока данных являются методы исследования графа потока управления (Control Flow Graph - CFG) [44] и методы анализа графа потока данных (Data Flow Analysis) [45].

4.2.1 Анализ потока управления

Анализ потока управления необходим для сбора информации о последовательности выполнения операторов в программе. Он позволяет определить, какие участки кода будут выполнены во время выполнения программы, и какие переменные могут получать новые значения в различных точках программы. Граф потока управления – это ориентированный граф, который моделирует поток выполнения программы. В графе потока управления каждый базовый блок представлен узлом, а переходы между блоками - ребрами [46]. Граф потока управления функции обозначается как Г(Ф): Г(Ф) = (Б, Р, $б_н$, $б_к$). Где:

- Б – набор базовых блоков (Базовый блок — это последовательность инструкций, в которой выполнение начинается только с начала блока и завершается только в конце блока, без переходов внутри блока);
- Р – набор направленных ребер;
- $б_н$ – начальный или входной блок;
- $б_к$ – конечный или выходной блок.

На рисунке 16 показан граф потока управления для функции «send_exchange_report» в листинге 5. Кроме входного (B1) и выходного блока (B7) код функции отображается в 5 блоках: B2, B3, B4, B5, B6. Функция «send_exchange_report» может выполняться через разные пути в графе потока управления:

- B1 → B2 → B3 → B4 → B5 → B6 → B7
- B1 → B2 → B4 → B5 → B6 → B7
- B1 → B2 → B4 → B6 → B7
- B1 → B2 → B3 → B4 → B6 → B7

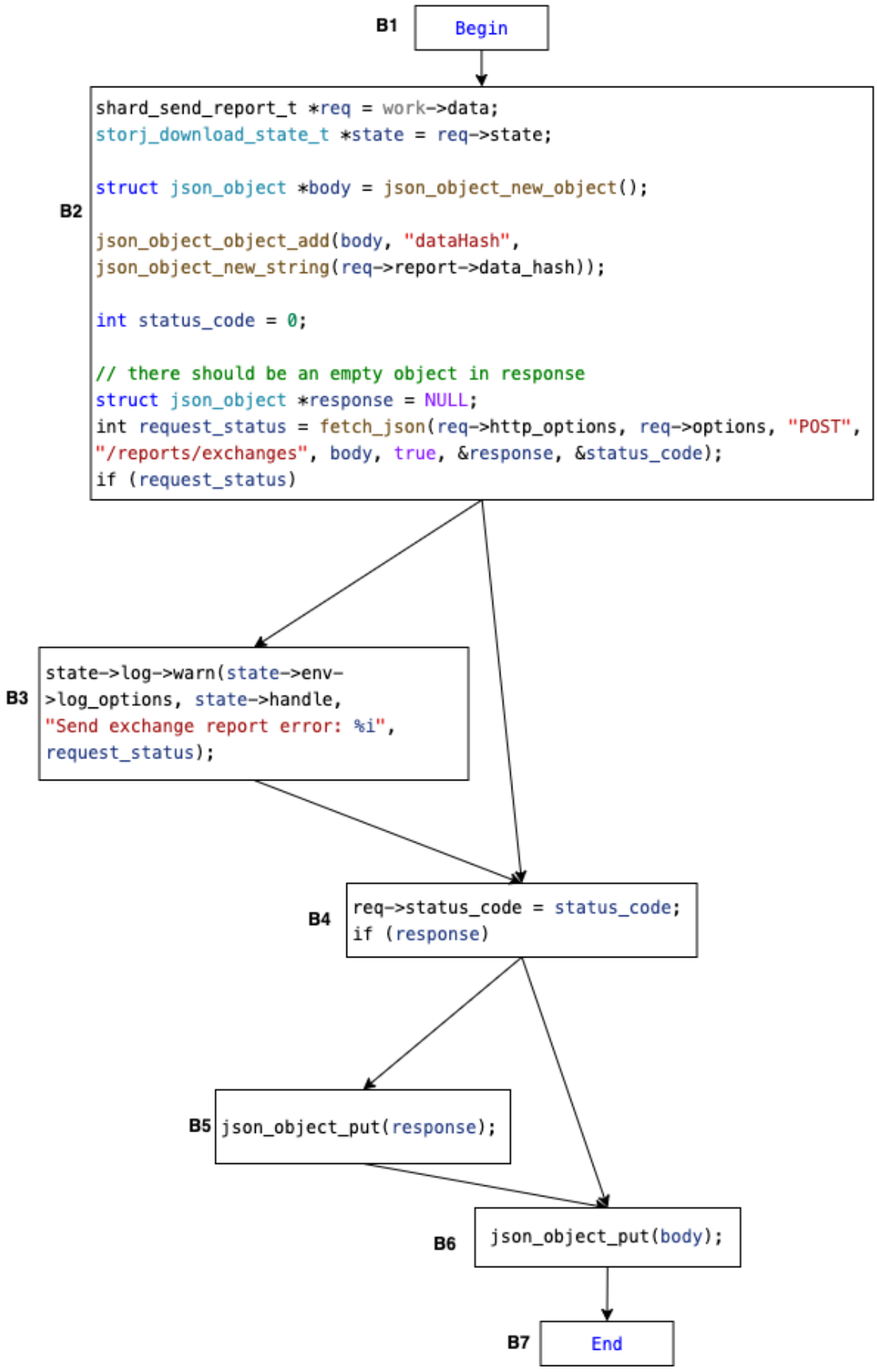

Рисунок 16. Граф потока управления для функции «send_exchange_report».

4.2.2 Анализ потока данных

Для анализа зависимостей между вызовами используется метод анализа потока данных (Data Flow Analysis) – это метод, который позволяет определить потоки передачи значений от одних переменных или констант к другим переменным (параметрам функций) и анализировать их свойства, такие как зависимости, переопределение, использование и т.д. В практике анализа и верификации программного кода анализ потока данных помогает выявить ошибки и уязвимости в программе, а также повысить ее производительность [50].

Анализ потока данных может быть выполнен с помощью различных инструментов и методов, таких как анализ графа потока данных (Data Flow Graph), системы типов, аннотации и т.д. Во время анализа потока данных инструменты и методы используются для поиска зависимостей между переменными, функциями и операторами в программе [51]. Например, анализ потока данных может помочь выявить неиспользуемые переменные, неиспользуемый код, некорректные пути выполнения программы и т.д.

В данной работе целью анализа потока данных является определение (построение описания) контекста использования тестируемой библиотеки.

4.3 Метод генерации фаззинг-оберток для функций библиотеки с учетом контекста использования

Данный метод позволяет определить контекст использования библиотеки в пользовательских программах и генерировать фаззинг-обертки для функций с учетом выявленных контекстов использования. Метод состоит из трех последовательных этапов (рисунок 17):

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

2. Найденные на первом этапе контексты поступают на вход второго этапа для генерации фаззинг-обертки.
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 дальше.

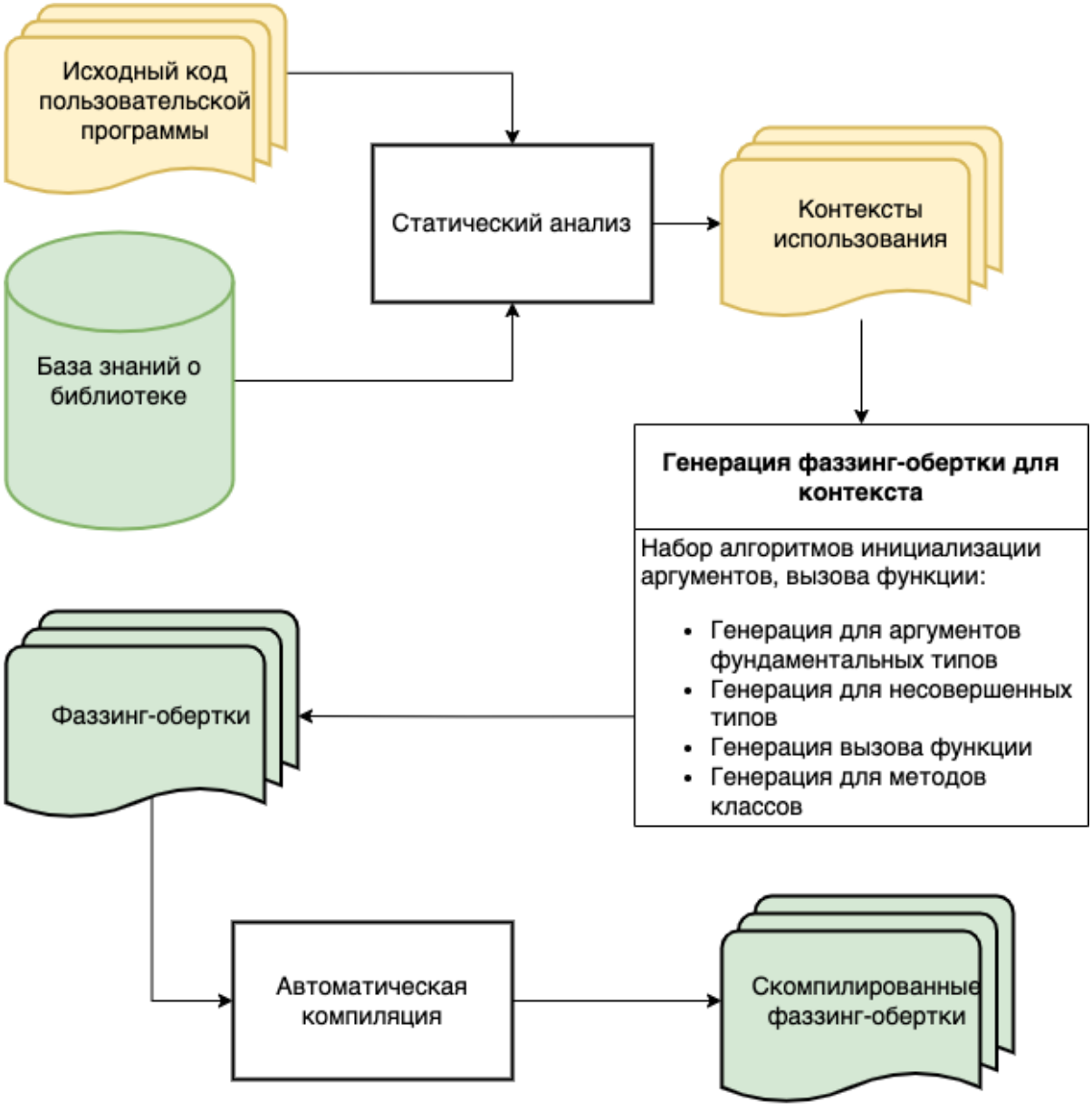

Рисунок 17. Схема метода генерации фаззинг-оберток для функций библиотеки с учетом контекста использования.

Алгоритм определения контекстов принимает на вход исходный код пользовательской программы и базу знаний о тестируемой библиотеке и включает в себя шаги (рисунок 18):

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 в 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.

   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 наборов И и К по путям выполнения: по базовым блокам и по инструкциям внутри каждого базового блока.

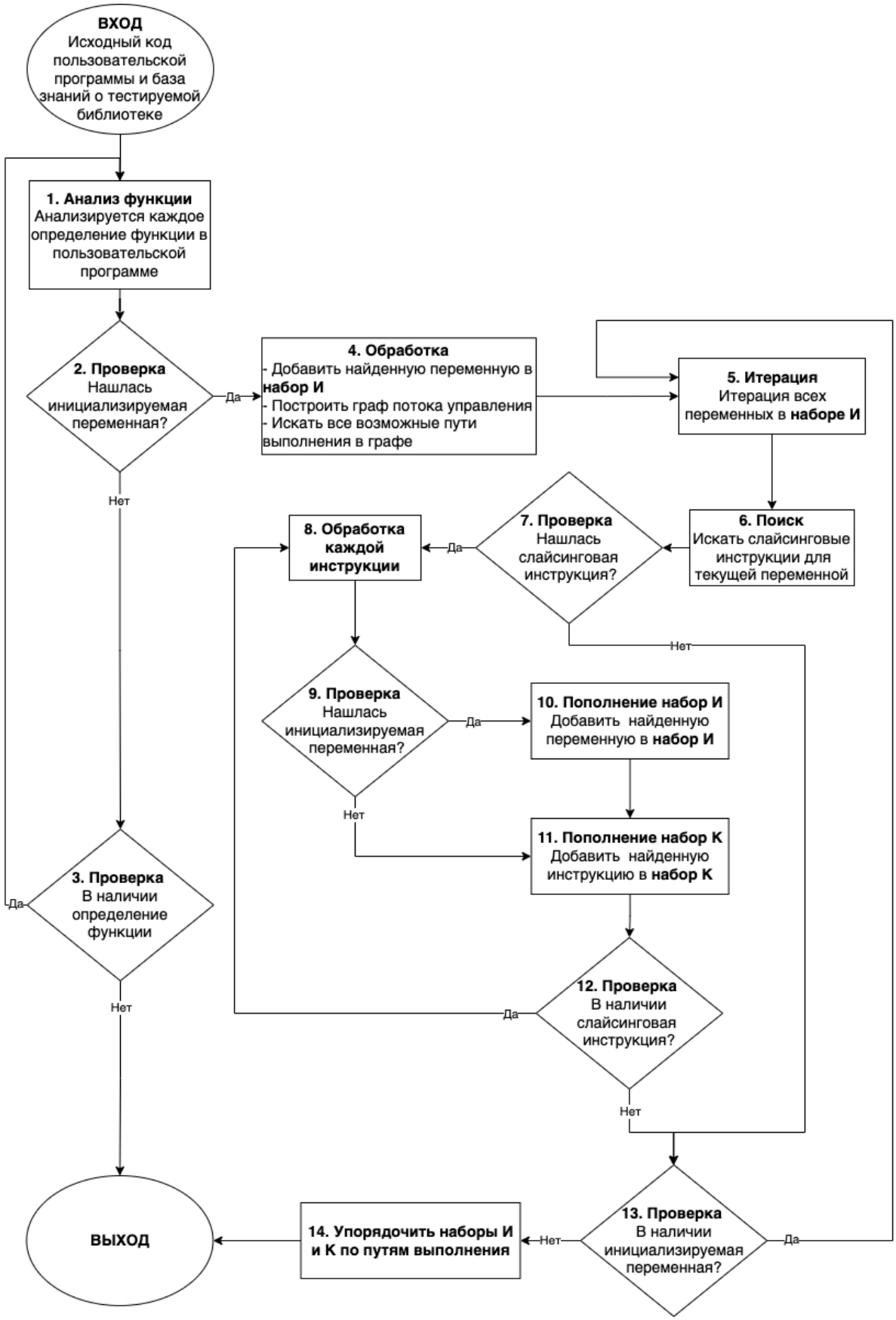

Рисунок 18. Алгоритм реализации метода определения контекста вызовов.

На выходе получаются контексты использования тестируемой библиотеки в пользовательской программе. Эти контексты могут включать переменные, которые не инициализированы в анализируемом узле. В этом случае переменные могут быть инициализированы с помощью:

- десериализации буфера фаззера, если переменная имеет простой тип данных;
- программного интерфейса библиотеки. То есть выполняется поиск функции в программном интерфейсе библиотеки, которая возвращает тип данных данной переменной. В случае, когда находится несколько подходящих функций, создается контекст для каждого способа инициализации переменной.

Из описания приведенного алгоритма видно, что контекст отличается от классического слайсинга тем, что в нашем случае:

- количество переменных до выполнения поиска заранее не задано;
- поиск контекста выполняется только локально, в определении тела функции пользовательской программы.

Результат генерации фаззинг-обертки для контекста использования библиотеки json-c в функции «send_exchange_report» показан в листинге 15:

- Строка 21: инициализируется переменная «body».
- Строка 81: выполняется вызов функции из слайса «json_object_object_add», который принимает «body» как первый аргумент, «key7» («key7» принимает значение константы на строке 79) как второй аргумент и «FutagRefVarpta» (инициализирован на строке 29 вызовом функции «json_object_new_string») как третий аргумент.
- Строка 101: вызывается «json_object_put» для «body».

Контекст успешно сформировался.

Листинг 15. Результат метода генерации фаззинг-обертки с контекстом использования.

```
1 #include "json.h"
2 // Другие заголовочные файлы
3
```

```
 4 extern "C" int LLVMFuzzerTestOneInput(uint8_t *
   Fuzz_Data, size_t Fuzz_Size){
 5     if (Fuzz_Size < (sizeof(int64_t) + sizeof(int64_t)
   + sizeof(int32_t)) return 0;
 6     size_t dyn_cstring_buffer = (size_t) ((Fuzz_Size -
   (sizeof(int64_t) + sizeof(int64_t) + sizeof(int32_t)
   )));
 7     //generate random array of dynamic string sizes
 8     size_t dyn_cstring_size[3];
 9     srand(time(NULL));
10     if(dyn_cstring_buffer == 0) dyn_cstring_size[0] =
   dyn_cstring_buffer;
11     else dyn_cstring_size[0] = rand() %
   dyn_cstring_buffer;
12     size_t remain = dyn_cstring_size[0];
13     for(size_t i = 1; i< 3 - 1; i++){
14         if(dyn_cstring_buffer - remain == 0)
   dyn_cstring_size[i] = dyn_cstring_buffer - remain;
15         else dyn_cstring_size[i] = rand() %
   (dyn_cstring_buffer - remain);
16         remain += dyn_cstring_size[i];
17     }
18     dyn_cstring_size[3 - 1] = dyn_cstring_buffer -
   remain;
19     //end of generation random array of dynamic string
   sizes
20     uint8_t * Futag_pos = Fuzz_Data;
21     struct json_object * body =
   json_object_new_object();
22     //GEN_CSTRING
23     char * rstr_s0 = (char *)
   malloc(dyn_cstring_size[0] + 1);
24     memset(rstr_s0, 0, dyn_cstring_size[0] + 1);
25     memcpy(rstr_s0, Futag_pos, dyn_cstring_size[0] );
26     Futag_pos += dyn_cstring_size[0];
27     const char * str_s0 = rstr_s0;
28
29     struct json_object * FutagRefVarpta =
   json_object_new_string(str_s0 );
30     //FREE
31     if (rstr_s0) {
32         free(rstr_s0);
33         rstr_s0 = NULL;
```

```
34      }
35      //GEN_CSTRING
36      char * rstr_s1 = (char *)
    malloc(dyn_cstring_size[1] + 1);
37      memset(rstr_s1, 0, dyn_cstring_size[1] + 1);
38      memcpy(rstr_s1, Futag_pos, dyn_cstring_size[1] );
39      Futag_pos += dyn_cstring_size[1];
40      const char * str_s1 = rstr_s1;
41
42      struct json_object * FutagRefVarizK =
    json_object_new_string(str_s1 );
43      //FREE
44      if (rstr_s1) {
45          free(rstr_s1);
46          rstr_s1 = NULL;
47      }
48      //GEN_BUILTIN
49      int64_t b_i2;
50      memcpy(&b_i2, Futag_pos, sizeof(int64_t));
51      Futag_pos += sizeof(int64_t);
52
53      struct json_object * FutagRefVarlcM =
    json_object_new_int64(b_i2 );
54      //GEN_BUILTIN
55      int64_t b_i3;
56      memcpy(&b_i3, Futag_pos, sizeof(int64_t));
57      Futag_pos += sizeof(int64_t);
58
59      struct json_object * FutagRefVarRLT =
    json_object_new_int64(b_i3 );
60      //GEN_BUILTIN
61      int32_t b_i4;
62      memcpy(&b_i4, Futag_pos, sizeof(int32_t));
63      Futag_pos += sizeof(int32_t);
64
65      struct json_object * FutagRefVarirx =
    json_object_new_int(b_i4 );
66      //GEN_CSTRING
67      char * rstr_s5 = (char *)
    malloc(dyn_cstring_size[2] + 1);
68      memset(rstr_s5, 0, dyn_cstring_size[2] + 1);
69      memcpy(rstr_s5, Futag_pos, dyn_cstring_size[2] );
70      Futag_pos += dyn_cstring_size[2];
```

```
 71     const char * str_s5 = rstr_s5;
 72
 73     struct json_object * FutagRefVarG8H =
    json_object_new_string(str_s5 );
 74     //FREE
 75     if (rstr_s5) {
 76         free(rstr_s5);
 77         rstr_s5 = NULL;
 78     }
 79     const char * key7 = "dataHash";
 80     //FUNCTION_CALL
 81     json_object_object_add(body ,key7 ,FutagRefVarpta
    );
 82     const char * key10 = "token";
 83     //FUNCTION_CALL
 84     json_object_object_add(body ,key10 ,FutagRefVarizK
    );
 85     const char * key13 = "exchangeStart";
 86     //FUNCTION_CALL
 87     json_object_object_add(body ,key13 ,FutagRefVarlcM
    );
 88     const char * key16 = "exchangeEnd";
 89     //FUNCTION_CALL
 90     json_object_object_add(body ,key16 ,FutagRefVarRLT
    );
 91     const char * key19 = "exchangeResultCode";
 92     //FUNCTION_CALL
 93     json_object_object_add(body ,key19 ,FutagRefVarirx
 94 );
 95     const char * key22 = "exchangeResultMessage";
 96     //FUNCTION_CALL
 97     json_object_object_add(body ,key22 ,FutagRefVarG8H
    );
 98     //FUNCTION_CALL
 99     json_object_to_json_string(body );
100     //FUNCTION_CALL
101     json_object_put(body );
102     return 0;
103 }
```

Глава 5. Реализация предложенных методов

Для реализации предложенных методов было разработано программное средство Futag. Futag предоставляет возможность генерации фаззинг-оберток как в случае отсутствия информации о контекстах использования тестируемой библиотеки, так и при наличии информации о контекстах. Futag использует в качестве внешнего инструмента для анализа исходного кода библиотек инструменты на основе Clang. Доступ к исходному коду программы свободный по ссылке: https://github.com/ispras/Futag и содержит:

- измененный проект LLVM с набором инструментов: компилятор clang, clang++, датчики обнаружения ошибок AddressSanitizer и т.д.;
- АСД-детекторы, АСД-инспекторы и АСД-обработчики;
- Python-пакет, который позволяет управлять процессом компиляции тестируемой библиотеки, автоматически генерировать фаззинг-обертки и собирать результаты фаззинга; данный пакет состоит из более 10 тысяч строк кода. Работа и структура данного пакета и его модулей описываются в следующих разделах;
- дополнительные скрипты для сборки, установки программы Futag и шаблоны для экспорта результата в разных форматах.

Сгенерированные фаззинг-обертки компилируются и запускаются модулем фаззинга программы Futag. Результаты фаззинга анализируются и записываются в формате системы Svace [53, 54].

В данной главе описывается работа трех основных модулей программного средства Futag:

1. модуль автоматизированного анализа;
2. модуль автоматизированной генерации;
3. модуль автоматизированного фаззинга и анализа результатов.

Практический результат диссертационной работы, а именно программное средство Futag, используется в Научно-техническом центре «ФОБОС-НТ» при выполнении фаззинг-тестирования исследуемого программного обеспечения, исходные коды которого

написаны на языках программирования Си/Си++. Акт внедрения приведен в приложении А.

5.1 Модуль автоматизированного анализа

Модуль автоматизированного анализа предназначен для:

- анализа исходного кода библиотеки или пользовательской программы;
- компиляции библиотеки с санитайзерами при необходимости;
- сбора результатов анализа в базу знаний (о библиотеке или о контексте использования тестируемой библиотеки).

Работа данного модуля в условиях отсутствия информации о контексте использования библиотеки реализована с помощью класса «Builder». Схема работы модуля иллюстрируется на рисунке 20:

- На вход принимаются исходный код библиотеки и подготовленный тестировщиком скрипт запуска модуля (Листинг 16). В скрипте возможно указывать: путь к программе Futag, путь к каталогу исходного кода библиотеки, количество потоков, флаги и дополнительные параметры для компиляции. Для некоторых библиотек требуется указывать опцию с/без шифрования (например, –with-ssl или –without-ssl).
- На первом шаге выполняется поиск системы сборки, использованной в исходном коде библиотеки. Также создаются папки для сохранения результата анализа, компиляции и инсталляции.
- На втором шаге класс «Builder» запускает анализатор «scan-build» с разработанными АСД-детекторами; АСД-детекторы запускают АСД-инспекторы (рисунок 19) и АСД-обработчики для сбора информации о сущностях и взаимосвязи между ними. После анализа результат сохраняется в JSON-файлах, которые содержат базу знаний о библиотеке. Пример собранной информации о функции, полученной статическим анализом, показан на листинге 17.
- На последнем шаге анализа библиотека компилируется с заданными параметрами и с санитайзерами для обнаружения

ошибки в процессе фаззинга. Скомпилированная библиотека будет передаваться на вход модуля генератора для сборки сгенерированных фаззинг-оберток.

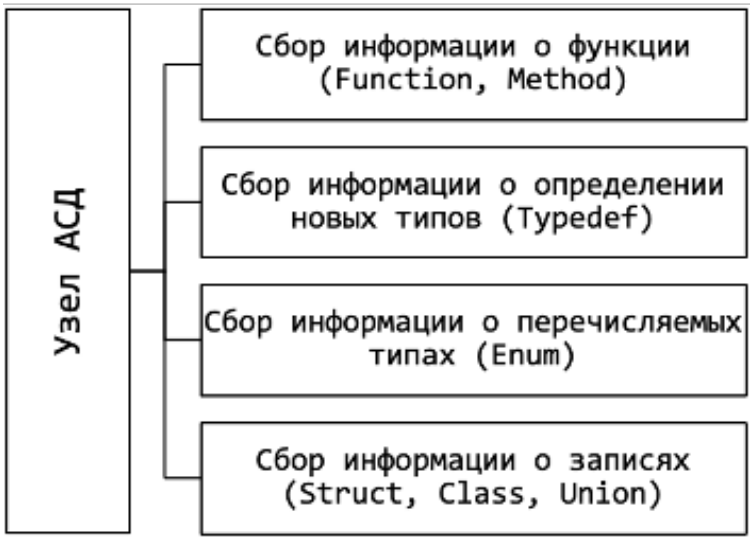

Рисунок 19. Реализованные АСД-инспекторы.

Листинг 16. Скрипт запуска модуля автоматизированного анализа с тестируемой библиотекой.

```
 1 from Futag.preprocessor import *
 2
 3 Futag_PATH = "/home/Futag/Futag-tests/Futag-llvm/"
 4 library_root = "json-c-json-c-0.16-20220414"
 5
 6 build_test = Builder(
 7         Futag_PATH,
 8         library_root,
 9         clean=True,
10         processes=16,
11         build_ex_params="--without-ssl"
12     )
13
14 build_test.auto_build()
15 build_test.analyze()
```

В данном скрипте:

- На строке 3 объявляется путь к программе Futag – «Futag_PATH».
- На строке 4 объявляется путь к каталогу исходного кода библиотеки «library_root».

- На строке 6 объявляется объект класса «Builder», который принимает в свой список аргументов: Futag_PATH, library_root, опцию clean=True для очистки созданных каталогов при анализе, количество потоков при компиляции «processes» и дополнительные параметры «build_ex_params» при подготовке библиотеки к сборке.
- На строке 14 вызывается функция автоматизированной сборки.
- На строке 15 вызывается функция «analyze» для сборки результата анализа.

Листинг 17. Пример собранной информации о функции.

```
{
    "name": "json_object_new_int",
    "qname": "json_object_new_int",
    "hash": "4134616113",
    "is_simple": true,
    "func_type": 4,
    "access_type": 3,
    "storage_class": 0,
    "parent_hash": "",
    "return_type": "struct json_object *",
    "params": [
        {
            "param_name": "i",
            "param_type": "int32_t",
            "param_usage": "UNKNOWN"
        }
    ],
    "fuzz_it": true,
    "location": {
        "file": "json_object.c",
        "line": "688",
        "directory": json-c-json-c-0.16-20220414",
        "fullpath":                    json-c-json-c-0.16-
20220414/json_object.c"
    }
}
```

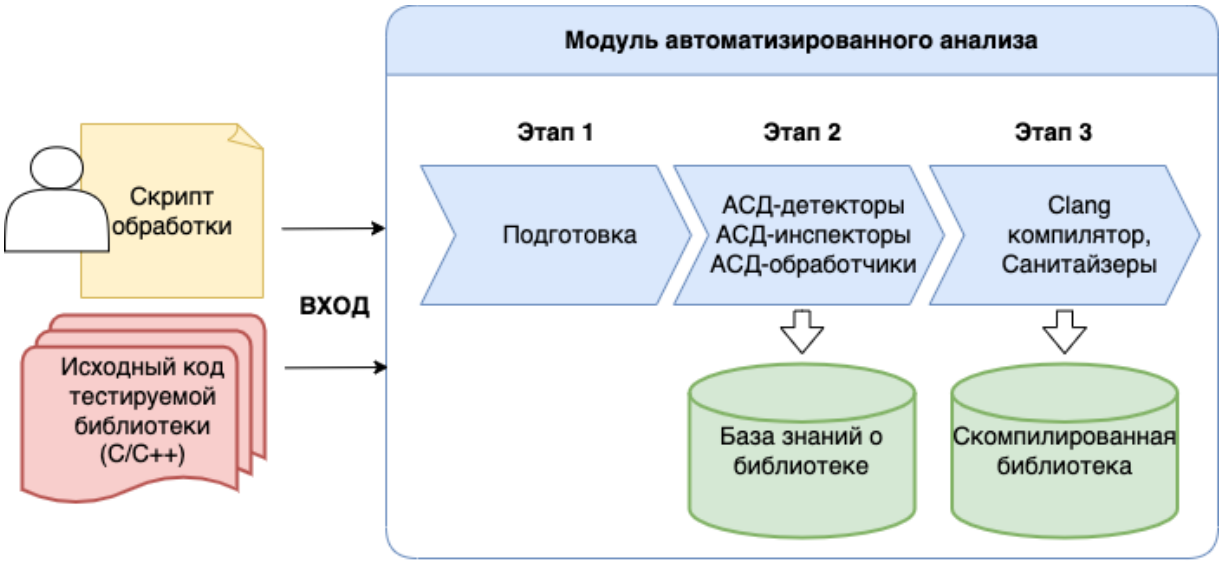

Рисунок 20. Работа модуля автоматизированного анализа в условии отсутствия пользовательских программ для выделения контекста.

В наличии исходного кода пользовательской программы работа автоматизированного анализа реализована классом «ConsumerBuilder», схема работы модуля иллюстрируется на рисунке 21:

- На входе принимаются база знаний тестируемой библиотеки, подготовленный тестировщиком скрипт запуска (Листинг 18) и исходный код пользовательской программы.
- На этапе подготовки также проходит поиск системы сборки, использованной в исходном коде пользовательской программе. Класс «ConsumerBuilder» запускает систему сборки с разработанным АСД-детектором и с неоходимыми параметрами с помощью инструмента «scan-build». АСД-детектор запускает АСД-инспекторы и АСД-обработчики для извлечения контекстов использования. АСД-инспектор генерирует граф потока управления для данного узла, ищет все пути выполнения. АСД-обработчики обнаруживают инициализируемые переменные и слайсинговые инструкции, которые образуют контекст использования тестируемой библиотеки. Эти контексты записываются в JSON-файлах, которые образуют базу знаний о контекстах использования тестируемой библиотеки.

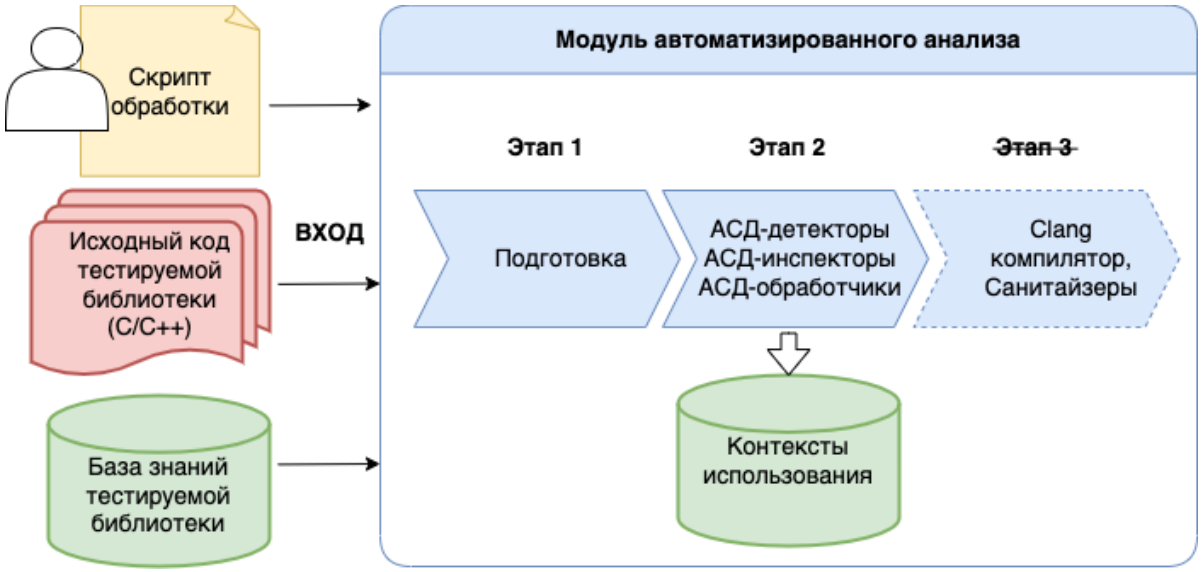

Рисунок 21. Работа модуля автоматизированного анализа при наличии исходного кода пользовательских программ.

Пример скрипта обработки для анализа контекстов использования библиотеки «json-c» в библиотеке «libstorj» показывается в листинге 18:

- В данном скрипте на строке 6 вызывается объект класса «ConsumerBuilder» для анализа.
- Кроме Futag_PATH и library_root, необходимо задавать путь к исходному коду пользовательской программы libstorj на строке 5.
- Остальные параметры аналогичны параметрам класс «Builder» в листинге 16.

Листинг 18. Скрипт запуска модуля автоматизированного анализа с пользовательской программой.

```
 1 from Futag.preprocessor import *
 2
 3 Futag_PATH = "/home/Futag/Futag-tests/Futag-llvm/"
 4 library_root = "json-c-json-c-0.16-20220414"
 5 consumer_root = "libstorj-1.0.3"
 6 build_test = ConsumerBuilder(
 7    Futag_PATH,
 8    library_root,
 9    consumer_root,
10    clean=True,
11    processes=16,
12 )
13 build_test.auto_build()
14 build_test.analyze()
```

5.2 Модуль автоматизированной генерации фаззинг-оберток

Модуль автоматизированной генерации фаззинг-оберток включает в себя 2 подмодуля: модуль генерации фаззинг-оберток в условиях отсутствия информации о контексте использования тестируемой библиотеки (класс «Generator») и модуль генерации фаззинг-оберток при наличии информации о контекстах использования тестируемой библиотеки (класс «ContextGenerator»). Данный модуль включает в себя методы для объявления и инициализации для переменных разных типов данных, ниже перечислены некоторые методы:

- __gen_builtin: метод для генерации переменной встроенного типа;
- __gen_cstring: метод для генерации переменной строкового типа на языке Си;
- __gen_cxxstring: метод для генерации переменной строкового типа на языке Си++;
- __gen_enum: метод для генерации переменной типа перечисления;
- __gen_array: метод для генерации массива;
- __gen_pointer: метод для генерации указателей;
- __gen_struct: метод для генерации переменной типа структуры;
- __gen_union: метод для генерации переменной типа обядинения;
- __gen_file_descriptor: метод для генерации файл-дескрипторов;
- __gen_class: метод для генерации объекта для классов.

Работа модуля иллюстрируется на рисунке 22:

- На входе принимаются база знаний тестируемой библиотеки, скомпилированная библиотека, подготовленный тестировщиком скрипт обработки и, возможно, контексты использования (при наличии). В скрипте обработки тестировщики могут задавать опцию для генерации фаззинг-обертки для LibFuzzer или AFLplusplus.
- На базе полученной информации Python-модуль генерации создает фаззинг-обертки для функций в библиотеке или для контекстов использования.

– Сгенерированные фаззинг-обертки затем передаются в Python-модуль компиляции для сборки. Для оформления команды компиляции на данном этапе анализируются все заголовочные файлы и все параметры единицы трансляции, в которой находятся функции тестируемой библиотеки сгенерированной фаззинг-обертки.

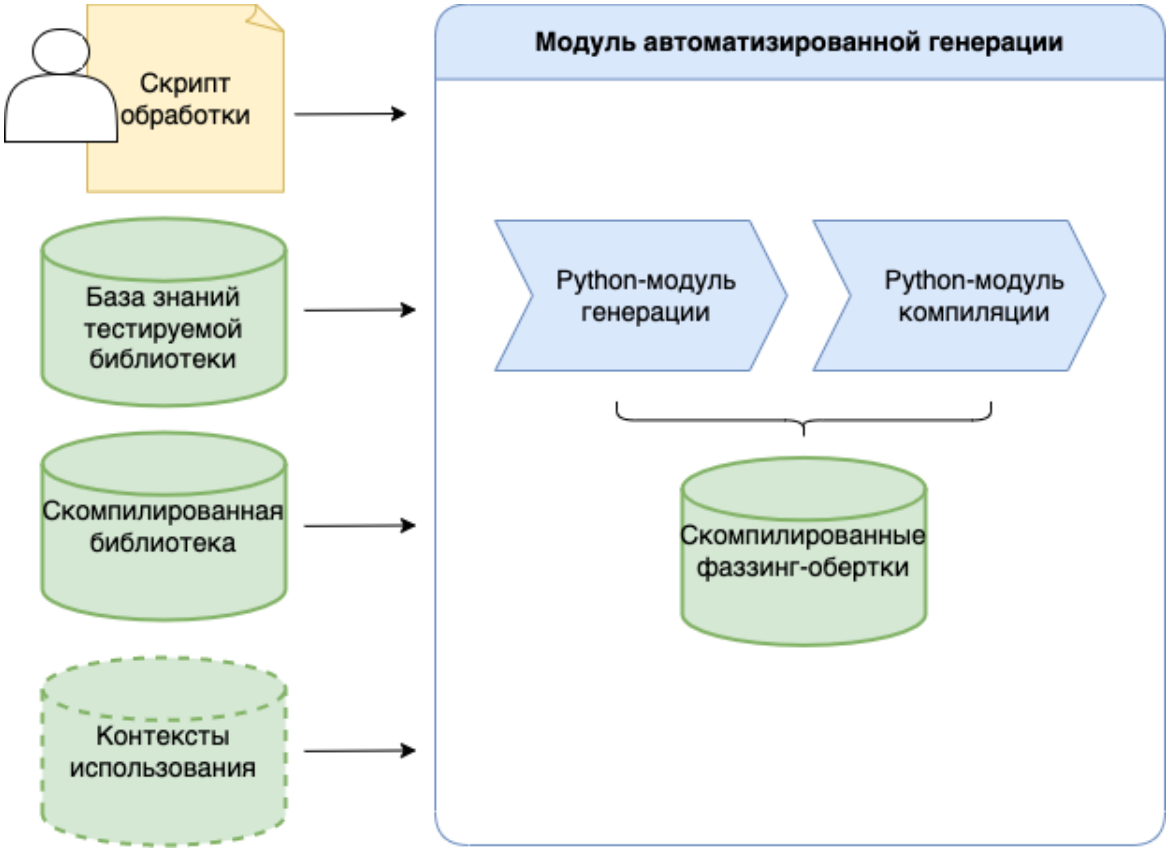

Рисунок 22. Работа модуля автоматизированной генерации фаззинг-оберток.

Пример скрипта запуска модуля автоматизированной генерации фаззинг-оберток в условиях отсутствия информации о контексте использования тестируемой библиотеки показывается в листинге 19:

– класс «Generator» также принимает «Futag_PATH» и «library_root» на входе;

– на строке 6 задается опция target_type=LIBFUZZER для генерации фаззинг-оберток для LibFuzzer;

– на строке 9 вызывается функция «gen_target» для запуска генерации. Эта функция принимает параметр логического типа «anonymous». Данный параметр используется для глубокого анализа исходного кода библиотеки. Если значение anonymous

равно «True», то генерируются фаззинг-обертки для функций, находящихся в недоступной области видимости (например, в анонимном пространстве имен);

- на строке 10 вызывается функция «compile_targets» для компиляции сгенерированных фаззинг-оберток. Данная функция принимает: аргумент «keep_failed=True» для сохранения нескомпилированных фаззинг-оберток; аргумент «workers» для задания количества потоков компиляции; аргумент «extra_dynamiclink» для задания динамической компоновки.

Листинг 19. Скрипт для запуска модуля генерации фаззинг-оберток для функций.

```
 1 from Futag.generator import *
 2
 3 generator = Generator(
 4     Futag_PATH,
 5     library_root,
 6     target_type=LIBFUZZER
 7 )
 8
 9 generator.gen_targets(anonymous=True)
10 generator.compile_targets(
11     keep_failed=True,
12     workers=4,
13     extra_dynamiclink=" -lm -lz "
14 )
```

5.3 Модуль фаззинга и анализа результатов

Данный модуль позволяет запускать фаззинг-тесты и анализировать результаты фаззинга. Работа модуля иллюстрируется на рисунке 23:

- на входе принимаются скомпилированные фаззинг-обертки и подготовленный тестировщиком скрипт обработки;
- данный модуль запускает 2 подмодуля: модуль фаззинга и модуль анализа результатов. Модуль фаззинга запускает скомпилированные фаззинг-обертки с заданными параметрами. Модуль анализа результатов обрабатывает лог-файлы для обнаружения трасс до места аварийного завершения; запускает и

собирает результаты отладки для создания файлов результатов в формате Svace.

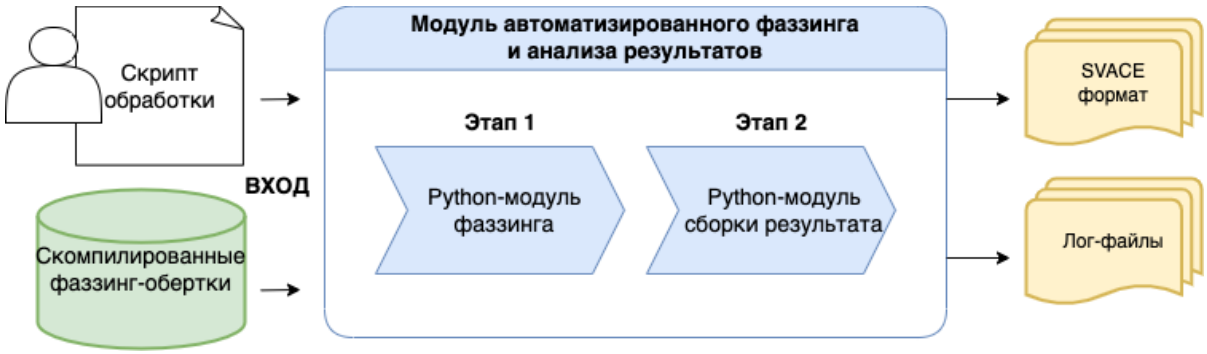

Рисунок 23. Работа модуля автоматизированного фаззинга.

Пример скрипта обработки показывается в листинге 20:

- на строке 4 объявляется объект класса «Fuzzer», который принимает на вход: путь к программе Futag, путь к каталогу скомпилированных фаззинг-обертки «fuzz_driver_path»;
- «totaltime» - время фаззинга для одной обертки, «svres» опция для оформления результата в Svace-формате;
- «gdb» - опция для загрузки ошибок в отладчик GDB для печати значений переменных в их месте завершения, «fork» - опция для задания количества потоков фаззинга;
- на строке 12 вызывается метод fuzz(), который последовательно выполняет фаззинг.

Листинг 20. Пример скрипта запуска модуля фаззинга.

```
 1 from Futag.fuzzer import *
 2 Futag_PATH = "/home/Futag/Futag-tests/Futag-llvm/"
 3 fuzz_driver_path = "json-c/Futag-fuzz-drivers"
 4 fuzzer = Fuzzer(
 5     Futag_PATH,
 6     fuzz_driver_path,
 7     totaltime=10,
 8     svres=True,
 9     gdb=True,
10     fork=1
11 )
12 fuzzer.fuzz()
```

В процессе фаззинга результат фаззинга отображается на экране, в случае, когда программа аварийно завершается, значение буфера

записывается в бинарный файл. Для анализа результатов фаззинга программа Futag выполняет следующие задачи:

- извлечь трассу выполнения до места аварии из выданной информации фаззера; пример трассы фаззинга показывается в листинге 21;
- найти места перехода для создания списка точек останова;
- загрузить фаззинг-обертку, значение буфера фаззера при аварии и список точек останова в отладчик GDB [55];
- вывести значения переменных в каждом фрейме при остановке и сохранить в XML-файл.

В результате получается XML-файл в формате системы Svace, который показан на рисунке 24.

Листинг 21. Пример трассы фаззинга.

```
==32440==ERROR: AddressSanitizer: heap-buffer-overflow on
address 0x60c000000773 at pc 0x000000493624 bp
0x7ffdb1dc6110 sp 0x7ffdb1dc58d0
READ of size 116 at 0x60c000000773 thread T0
    #0 0x493623 in __interceptor_strlen.part.52
/Futag/compiler-
rt/lib/asan/../sanitizer_common/sanitizer_common_intercept
ors.inc:389
    #1 0x55cfc1 in dopr /Github/Futag-tests/libpq-
standalone/libpq-standalone/libpq/snprintf.c:444:20
    #2 0x5603ea in pg_vfprintf /Github/Futag-tests/libpq-
standalone/libpq-standalone/libpq/snprintf.c:257:2
    #3 0x560c56 in pg_printf /Github/Futag-tests/libpq-
standalone/libpq-standalone/libpq/snprintf.c:288:8
    #4 0x55c238 in LLVMFuzzerTestOneInput /Github/Futag-
tests/libpq-standalone/libpq-standalone/Futag-fuzz-
drivers/pg_printf/pg_printf1/pg_printf1.c:20:5
    #5 0x446ef6 in
fuzzer::Fuzzer::ExecuteCallback(unsigned char const*,
unsigned long) /Futag/compiler-
rt/lib/fuzzer/FuzzerLoop.cpp:611
    #6 0x428c6a in fuzzer::RunOneTest(fuzzer::Fuzzer*,
char const*, unsigned long) /Futag/compiler-
rt/lib/fuzzer/FuzzerDriver.cpp:324
    #7 0x436621 in fuzzer::FuzzerDriver(int*, char***, int
(*)(unsigned char const*, unsigned long)) /Futag/compiler-
```

```
rt/lib/fuzzer/FuzzerDriver.cpp:860
    #8 0x41f362 in main /Futag/compiler-
rt/lib/fuzzer/FuzzerMain.cpp:20
    #9 0x7f5b153c8c86 in __libc_start_main /build/glibc-
uZu3wS/glibc-2.27/csu/../csu/libc-start.c:310
```

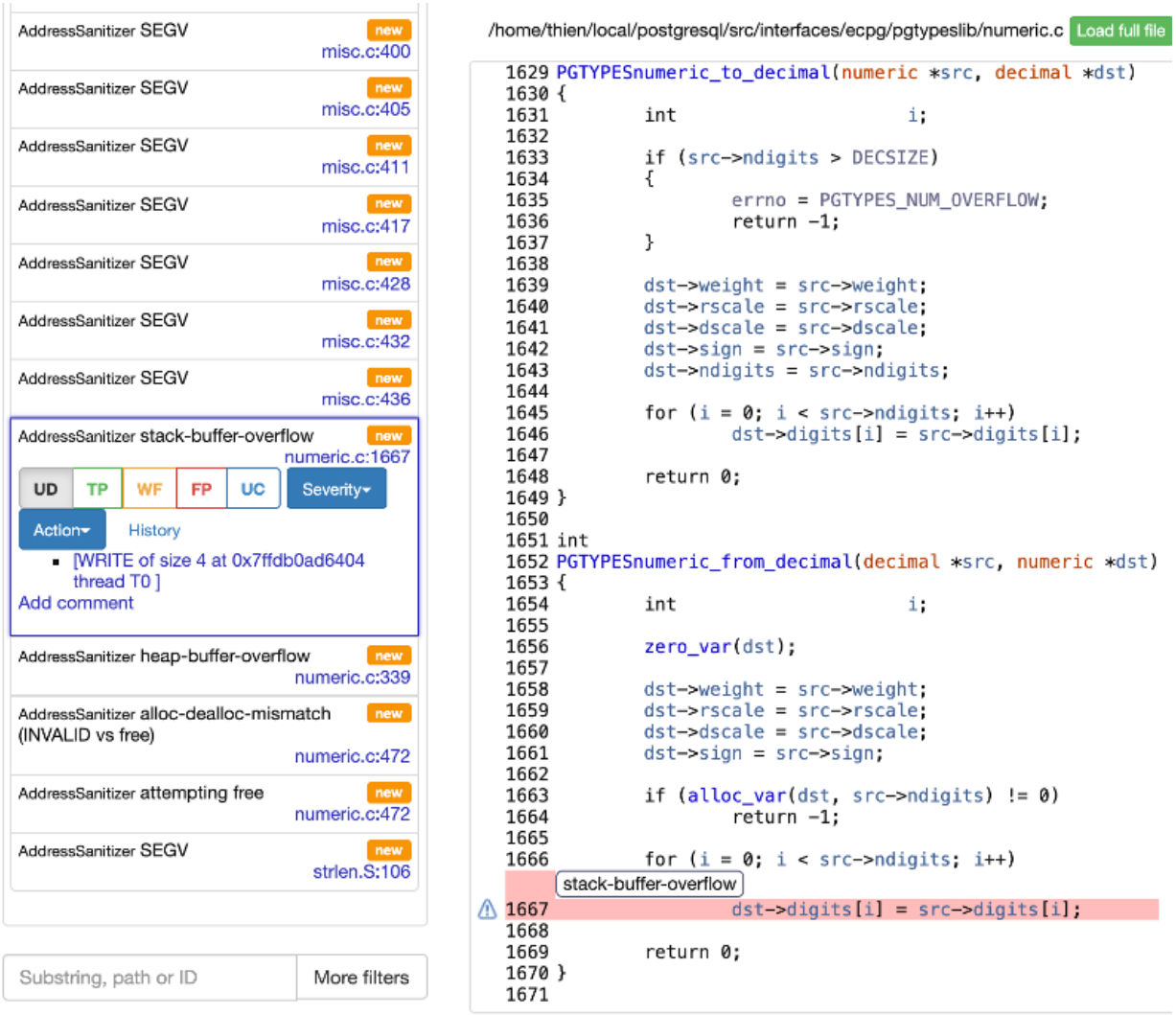

Рисунок 24. Пример загрузки результатов в систему Svace.

5.4 Результаты работы программного средства Futag

В данном разделе описываются результаты работы программного средства Futag по генерации фаззинг-оберток для библиотек. В подразделе 5.4.1 описываются случаи, когда Futag не может сгенерировать фаззинг-оберток; в подразделах 5.4.2 и 5.4.3 даются оценки результатов работы Futag в сравнении с альтернативными

инструментами в случае, когда средство успешно генерирует фаззинг-обертки; в подразделе 5.4.4 перечислены найденные ошибки в популярных библиотеках.

5.4.1 Случаи, когда программное средство Futag не может сгенерировать фаззинг-обертки

Процесс генерации фаззинг-оберки записывается в лог-файл для анализа. Ниже перечислены некоторые случаи, когда Futag не может сгенерировать фаззинг-обертки.

1. Генерация фаззинг-обертки для консрукторов класса, которые имеют параметр сложного типа: данный случай объясняется в шаге 11 алгоритма построения фаззинг-оберток для функций библиотеки в условиях отсутствия информации о контекстах использования (раздел 3.3).

2. Аргумент функции имеет тип данных, анализ которого пока не поддерживается.

В следующем листинге показан лог процесса генерации фаззинг-обертки для функции «json_object_deep_copy», которая принимает параметр функционального типа «json_c_shallow_copy_fn». Программное средство Futag пока не анализирует и не генерирует переменную данного типа данных.

```
Log for function: json_object_deep_copy
- Can not generate for object: {
   'base_type_name': '',
   'gen_type': 13,
   'gen_type_name': '_FUNCTION',
   'length': 0,
   'local_qualifier': '',
   'type_name': 'json_c_shallow_copy_fn'
}
```

3. Аргумент функции имеет сложный тип и инициализируется с помощью функции, которая имеет аргумент сложного типа.

В нижеприведенном листинге показано определение структуры «array_list» в библиотеке «json-c».

```
    struct array_list
    {
        void **array;
        size_t length;
```

```
        size_t size;
        array_list_free_fn *free_fn;
    };
```

Существует функция «array_list_new», которая возвращает переменную данного типа, однако данная функция принимает аргумент функционального типа, который не анализируется программным средством Futag.

```
struct array_list *array_list_new(array_list_free_fn
*free_fn)
{
    return array_list_new2(free_fn,
ARRAY_LIST_DEFAULT_SIZE);
}
```

В итоге для повышения возможности генерации фаззинг-обертки необходимо добавить генератор для типов функции или метода (случай 1) и генератор для неподдерживаемых типов данных (случаи 2 и 3).

5.4.2. Количественная оценка работы программы Futag

Для оценки работы программы Futag проводился анализ ее работы для разных библиотек. В качестве объекта анализа были выбраны широко известные библиотеки:

- libjson-c,
- libpostgres,
- curl,
- openssl,
- pugixml,
- libopus.

Результаты анализа приведены в таблице 2. Колонки таблицы это:

- время генерации — это время требуется для анализа и генерации фаззинг-оберток для функций библиотеки;
- время компиляции — это время требуется для компиляции всех фаззинг-оберток;
- количество фаззинг-оберток — это количество скомпилированных фаззинг-оберток;
- количество строк кода — это общая сумма строк всех сгенерированных фаззинг-оберток.

За 180 секунды (3 минуты) Futag сгенерировал 612 фаззинг-оберток для libjson-c, которые состоят из 280.019 строк кода; за 105 секунд (около 2 минут) Futag сгенерировал 29 фаззинг-оберток для libpostgres, которые состоят из 84.387 срок кода и т.п.

Таблица 2. Результат генерации фаззинг-оберток для библиотек.

Библиотека	Время генерации (секунды)	Кол-во фаззинг-оберток	Время компиляции (секунды)	Кол-во строк кода
libjson-c	180	612	3 111	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

Согласно книге Стива МакКоннелла «Software estimation: demystifying the black art» [56], 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 3, где:

- мин.время написания (человеко-день) - минимальное количество дней для написания фаззинг-оберток.
- макс.время написания (человеко-день) - максимальное количество дней для написания фаззинг-оберток.

Библиотека	Кол-во фаззинг-оберток	Кол-во строк кода	Мин.время написания (человеко-день)	Макс.время написания (человеко-день)
libjson-c	3 111	280 019	2240	11200

libpostgres	749	84 387	675	3375
curl	152	9 617	77	385
openssl	269	19 458	155	778
pugixml	61	2 815	22	113
libopus	422	12 606	100	504

Таблица 3. Время написания фаззинг-оберток вручную.

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

5.4.3 Качество фаззинг-обертки в сравнении с другими инструментами

В настоящее время доступ к альтернативным инструментам FUDGE и Intelligence закрыт, поэтому для оценки качества сгенерированных фаззинг-оберток в данной работе проводится сравнение результатов работы программного средства Futag с инструментами FuzzGen и OzzFuzz.

Сравнение с инструментом FuzzGen:

В репозитории инструмента на Github (https://github.com/HexHive/FuzzGen) представлена фаззинг-обертка для библиотеки «libopus» для системы Android. Библиотека «libopus» — это библиотека для декодера звуковых файлов. Перед декодированием необходимо создать декодер функцией «opus_decoder_create» и затем вызвать «opus_decode» для декодирования. В фаззинг-обертке [74], сгенерированной системой FuzzGen, переменная «dep_262» инициализируется функцией «opus_decoder_create» в строках 609, 668 и 861, затем с dep_262 как аргументом вызывается функция «opus_decode» в строках 1216, 1272, 2061. В листинге 22 показан сокращенный код фаззинг-обертки для библиотеки, сгенерированный инструментом FuzzGen.

Листинг 22. Фаззинг-обертка для бибилотеки json, сгенерированная инструментом FuzzGen.

```
 112 int32_t dep_270;
 ... ...
```

```
 121 int32_t dep_273;
 122 int32_t *dep_271;
 123 OpusDecoder* dep_262;
 ... ...
 505 dep_270 = (int32_t )channels_bbD_0;
 ... ...
 579 dep_273 = (int32_t )Fs_Nbu_0;
 ... ...
 609 dep_262   =   opus_decoder_create(dep_273,   dep_270,
     dep_271); /* vertex #1 */
 ... ...
 668 dep_262   =   opus_decoder_create(dep_273,   dep_270,
     NULL); /* vertex #2 */
 ... ...
 861 dep_262 = opus_decoder_create(48000, 2, dep_271); /*
     vertex #5 */
 ... ...
1216 opus_decode(dep_262,  (const  uint8_t  *)dep_267,  3,
     (int16_t *)dep_269, 960, 0); /* vertex #15 */
 ... ...
1272 opus_decode(dep_262,  (const  uint8_t  *)dep_267,  1,
     (int16_t *)dep_269, 960, 0); /* vertex #17 */
 ... ...
2061 opus_decode(dep_262, dep_267, 51, (int16_t *)dep_269,
     960, 0); /* vertex #43 */
```

Сгенерированная фаззинг-обертка программным средством Futag для библиотеки «libopus» показана в листинге 23. Перед тем, как аргумент «s__st» передается в функцию «opus_decode» (на строке 56), он инициализируется вызовом функции «opus_decoder_create» (на строке 25), что обеспечивает контекст использования данной библиотеки.

В итоге сравнения видно, что результат работы программы Futag аналогичен результату инструмента FuzzGen.

Листинг 23. Фаззинг-обертка для библиотеки «opus», сгенерированная программой Futag.

```
 1 int LLVMFuzzerTestOneInput(uint8_t * Fuzz_Data, size_t
   Fuzz_Size){
 2     if (Fuzz_Size < sizeof(opus_int32) + sizeof(int) +
   sizeof(int) + sizeof(opus_int32) + sizeof(opus_int16) +
   sizeof(int) + sizeof(int)) return 0;
 3     size_t dyn_cstring_buffer = (size_t) (Fuzz_Size -
   (sizeof(opus_int32) + sizeof(int) + sizeof(int) +
   sizeof(opus_int32) + sizeof(opus_int16) + sizeof(int) +
```

```
    sizeof(int) ));
 4      //generate random array of dynamic string sizes
 5      size_t dyn_cstring_size[1];
 6      dyn_cstring_size[0] = dyn_cstring_buffer;
 7      //end of generation random array of dynamic string
    sizes
 8      uint8_t * Futag_pos = Fuzz_Data;
 9
10      //GEN_BUILTIN
11      opus_int32 b_1_Fs;
12      memcpy(&b_1_Fs, Futag_pos, sizeof(opus_int32));
13      Futag_pos += sizeof(opus_int32);
14      //GEN_BUILTIN
15      int b_1_channels;
16      memcpy(&b_1_channels, Futag_pos, sizeof(int));
17      Futag_pos += sizeof(int);
18      //GEN_BUILTIN
19      int b__1_error;
20      memcpy(&b__1_error, Futag_pos, sizeof(int));
21      Futag_pos += sizeof(int);
22      //GEN_POINTER
23      int * p_b__1_error = & b__1_error;
24      //GEN_VAR_FUNCTION
25      OpusDecoder * s__st =
    opus_decoder_create(b_1_Fs,b_1_channels,p_b__1_error);
26      //GEN_CSTRING
27      unsigned char * rstr_data = (unsigned char *)
    malloc((dyn_cstring_size[0] + 1)* sizeof(char));
28      memset(rstr_data, 0, dyn_cstring_size[0] + 1);
29      memcpy(rstr_data, Futag_pos, dyn_cstring_size[0]);
30      Futag_pos += dyn_cstring_size[0];
31      const unsigned char * str_data = rstr_data;
32
33      //GEN_BUILTIN
34      opus_int32 b_len;
35      memcpy(&b_len, Futag_pos, sizeof(opus_int32));
36      Futag_pos += sizeof(opus_int32);
37
38      //GEN_BUILTIN
39      opus_int16 b__pcm;
40      memcpy(&b__pcm, Futag_pos, sizeof(opus_int16));
41      Futag_pos += sizeof(opus_int16);
42      //GEN_POINTER
```

```
43      opus_int16 * p_b__pcm = & b__pcm;
44
45      //GEN_BUILTIN
46      int b_frame_size;
47      memcpy(&b_frame_size, Futag_pos, sizeof(int));
48      Futag_pos += sizeof(int);
49
50      //GEN_BUILTIN
51      int b_decode_fec;
52      memcpy(&b_decode_fec, Futag_pos, sizeof(int));
53      Futag_pos += sizeof(int);
54
55      //FUNCTION_CALL
56      opus_decode(s__st ,str_data ,b_len ,p_b__pcm
    ,b_frame_size ,b_decode_fec );
57      //FREE
58      if (rstr_data) {
59          free(rstr_data);
60          rstr_data = NULL;
61      }
62      return 0;
63  }
```

Сравнение с инструментом OSS-Fuzz:

OSS-Fuzz [57] - это бесплатный инструмент для тестирования безопасности и стабильности программного обеспечения с открытым исходным кодом. Он был разработан командой Google, чтобы помочь сообществу открытых исходных кодов создавать более надежное и безопасное программное обеспечение.

OSS-Fuzz автоматически запускает тысячи тестов на исходном коде программного обеспечения, используя инфраструктуру облачных вычислений. Это помогает обнаруживать потенциальные ошибки, уязвимости и другие проблемы, которые могут быть опасными для конечных пользователей.

В листинге 24 показывается фаззинг-обертка для функции «json_tokener_parse_ex», которая написана разработчиком и выложена в системе «OSS-Fuzz». Перед тем, как переменная «tok» передается в функцию «json_tokener_parse_ex», она инициализируется вызовом функции «json_tokener_new», что обеспечивает контекст использования

библиотеки «json-c». Аналогичный результат для Futag можно посмотреть в листинге 14 на строке 13. Кроме того, Futag генерировал и другие обертки, которые инициализировали переменную «json_tokener *tok» другим способом (с помощью функции json_tokener_new_ex(int), см. листинг 25 — строка 15).

Листинг 24. Фаззинг-обертка для библиотеки json на OSS-Fuzz.

```
 1 #include <json.h>
 2
 3 extern "C" int LLVMFuzzerTestOneInput(const uint8_t
   *data, size_t size) {
 4     const char *data1 = reinterpret_cast<const char
   *>(data);
 5     json_tokener *tok = json_tokener_new();
 6     json_object *obj = json_tokener_parse_ex(tok,
   data1, size);
 7
 8     json_object_put(obj);
 9     json_tokener_free(tok);
10     return 0;
11 }
```

Листинг 25. Инициализация переменной s__tok другим способом.

```
 1 #include <json.h>
 2 int LLVMFuzzerTestOneInput(uint8_t * Fuzz_Data, size_t
   Fuzz_Size){
 3     if (Fuzz_Size < sizeof(int)) return 0;
 4     size_t dyn_cstring_buffer = (size_t) (Fuzz_Size -
   sizeof(int));
 5     //generate random array of dynamic string sizes
 6     size_t dyn_cstring_size[1];
 7     dyn_cstring_size[0] = dyn_cstring_buffer;
 8     //end of generation random array of dynamic string
   sizes
 9     uint8_t * futag_pos = Fuzz_Data;
10     //GEN_BUILTIN
11     int b_1_depth;
12     memcpy(&b_1_depth, futag_pos, sizeof(int));
13     futag_pos += sizeof(int);
14     //GEN_VAR_FUNCTION
15     struct json_tokener * s__tok =
```

```
    json_tokener_new_ex(b_1_depth);
16      //GEN_CSTRING1
17      char * rstr_str = (char *)
    malloc((dyn_cstring_size[0] + 1)* sizeof(char));
18      memset(rstr_str, 0, dyn_cstring_size[0] + 1);
19      memcpy(rstr_str, futag_pos, dyn_cstring_size[0]);
20      futag_pos += dyn_cstring_size[0];
21      const char * str_str = rstr_str;
22      //GEN_SIZE
23      int sz_len = (int) dyn_cstring_size[0];
24      //FUNCTION_CALL
25      json_tokener_parse_ex(s__tok ,str_str ,sz_len );
26      //FREE
27      if (rstr_str) {
28          free(rstr_str);
29          rstr_str = NULL;
30      }
31      return 0;
32  }
```

В итоге сравнения видно, что результат работы программы Futag аналогичен результату инструмента «OSS-Fuzz». Если фаззинг-обертки в «OSS-Fuzz» написаны разработчиками и их количество небольшое, то программа Futag может автоматизированно генерировать больше фаззинг-оберток.

5.4.4 Найденные ошибки

В данном разделе перечисляются 2 ошибки в популярных библиотеках, найденные программой Futag.

1. **Ошибка в функции «png_convert_ from_time_t» библиотеки «libpng».**

В библиотеке libpng функция png_convert_from_time_t используется для преобразования системного времени в формате «time_t» в формат «png_time». Код функции png_convert_from_time_t в библиотеке libpng версии 1.6.37 показан в листинге 26. Данная функция преобразует входные данные «ttime» с помощью системного вызова «gmtime» (вызывается в 8-ой строке) и записывает результат в указатель «ptime». После вызова «gmtime» отсутствует проверка на нулевое значение, что проводит к аварийному завершению при выполнении следующего вызова «png_convert_from_struct_tm(ptime, tbuf)».

Листинг 26. Код функции png_convert_from_time_t в библиотеке libpng до исправления.

```
 1 void PNGAPI
 2 png_convert_from_time_t(png_timep ptime, time_t ttime)
 3 {
 4    struct tm *tbuf;
 5
 6    png_debug(1, "in png_convert_from_time_t");
 7
 8    tbuf = gmtime(&ttime);
 9    png_convert_from_struct_tm(ptime, tbuf);
10 }
```

Фаззинг-обертка для функции png_convert_from_time_t, сгенерированная программой Futag, показана в листинге 27. При выполнении данной фаззинг-обертки LibFuzzer выдает ошибку «AddressSanitizer: SEGV on unknown address». Эта ошибка была передана разработчикам библиотеки libpng 08 февраля 2021 г. Ошибка исправлена 13 сентября 2022г. Код функции «png_convert_from_time_t» после исправления показан в листинге 28.

Листинг 27. Фаззинг-обертка для функции png_convert_from_time_t

```
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include<png.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
size_t Size) {
    if(Size < sizeof(png_time) + sizeof(time_t)) return 0;
    uint8_t * pos = (uint8_t *) Data;

    png_timep ptime = (png_timep)
malloc(sizeof(png_time));
    memset( ptime, 0, sizeof(ptime));
    memcpy(ptime, pos, sizeof(png_time));
    pos += sizeof(png_time);

    time_t ttime = {0};
    memcpy(&ttime, pos, sizeof(time_t));
```

```
    png_convert_from_time_t(ptime, ttime);
    free( (png_timep) ptime);
    return 0;
}
```

Листинг 28. Код функции «png_convert_from_time_t» после исправления.

```
void PNGAPI
png_convert_from_time_t(png_timep ptime, time_t ttime)
{
    struct tm *tbuf;
    png_debug(1, "in png_convert_from_time_t");

    tbuf = gmtime(&ttime);
    if (tbuf == NULL)
    {
        /* TODO: add a safe function which takes a png_ptr
argument and raises
        * a png_error if the ttime argument is invalid and
the call to gmtime
        * fails as a consequence.
        */
        memset(ptime, 0, sizeof(*ptime));
        return;
    }
    png_convert_from_time_t(ptime, ttime);
}
```

Подробную информацию можно посмотреть на сайте разработки библиотеки libpng по ссылке: https://github.com/glennrp/libpng/issues/362.

2. **Ошибка в функции «get_file_size» библиотеки «pugixml».**

В библиотеке pugixml при чтении XML-документа из файла используется функция *get_file_size* для выделения необходимой памяти. В функции *get_file_size* используются системные вызовы «fseek» и «ftell» для получения размера файла, и в разных системах эти вызовы возвращают разные значения при работе со специальными файлами, такими, как «.», «/» и т.д. Код функции *get_file_size* в библиотеке *pugixml* версии 1.23 отображается в листинге 29.

Листинг 29. Код функции png_convert_from_time_t в библиотеке libpng до исправления.

```
PUGI__FN xml_parse_status get_file_size(FILE* file,
size_t& out_result){
#if defined(PUGI__MSVC_CRT_VERSION) &&
PUGI__MSVC_CRT_VERSION >= 1400
    // there are 64-bit versions of fseek/ftell, let's use
them
    typedef __int64 length_type;
    _fseeki64(file, 0, SEEK_END);
    length_type length = _ftelli64(file);
    _fseeki64(file, 0, SEEK_SET);
#elif defined(__MINGW32__) && !defined(__NO_MINGW_LFS) &&
(!defined(__STRICT_ANSI__) ||
defined(__MINGW64_VERSION_MAJOR))
    // there are 64-bit versions of fseek/ftell, let's use
them
    typedef off64_t length_type;

    fseeko64(file, 0, SEEK_END);
    length_type length = ftello64(file);
    fseeko64(file, 0, SEEK_SET);
#else
    // if this is a 32-bit OS, long is enough; if this is
a unix system, long is 64-bit, which is enough; otherwise
we can't do anything anyway.
    typedef long length_type;

    fseek(file, 0, SEEK_END);
    length_type length = ftell(file);
    fseek(file, 0, SEEK_SET);
#endif

    // check for I/O errors
    if (length < 0) return status_io_error;
    // check for overflow
    size_t result = static_cast<size_t>(length);
    if (static_cast<length_type>(result) != length) return
status_out_of_memory;
    // finalize
    out_result = result;
    return status_ok;
}
```

Фаззинг-обертка для метода load_file, который вызывает функцию *get_file_size*, показана в листинге 30. В результате обработки данной фаззинг-обертки при передаче входного файла с названием «.» в системе Linux функция *get_file_size* возвращает размер, который равен **INT64_MAX**, что приводит к ошибке «*AddressSanitizer: allocation-size-too-big*». Данная ошибка исправлена 15 апреля 2023, код функции после исправления показан в листинге 31.

Листинг 30. Код функции png_convert_from_time_t в библиотеке libpng до исправления.

```
PUGI_IMPL_FN xml_parse_status get_file_size(FILE* file,
size_t& out_result)
{
#if defined(__linux__) || defined(__APPLE__)
    // this simultaneously retrieves the file size and
file mode (to guard against loading non-files)
    struct stat st;
    if (fstat(fileno(file), &st) != 0) return
status_io_error;

    // anything that's not a regular file doesn't have a
coherent length
    if (!S_ISREG(st.st_mode)) return status_io_error;

    typedef off_t length_type;
    length_type length = st.st_size;
#elif defined(PUGI_IMPL_MSVC_CRT_VERSION) &&
PUGI_IMPL_MSVC_CRT_VERSION >= 1400
    // there are 64-bit versions of fseek/ftell, let's use
them
    typedef __int64 length_type;

    _fseeki64(file, 0, SEEK_END);
    length_type length = _ftelli64(file);
    _fseeki64(file, 0, SEEK_SET);
#elif defined(__MINGW32__) && !defined(__NO_MINGW_LFS) &&
(!defined(__STRICT_ANSI__) ||
defined(__MINGW64_VERSION_MAJOR))
    // there are 64-bit versions of fseek/ftell, let's use
them
    typedef off64_t length_type;
```

```
    fseeko64(file, 0, SEEK_END);
    length_type length = ftello64(file);
    fseeko64(file, 0, SEEK_SET);
#else
    // if this is a 32-bit OS, long is enough; if this is
a unix system, long is 64-bit, which is enough; otherwise
we can't do anything anyway.
    typedef long length_type;
    fseek(file, 0, SEEK_END);
    length_type length = ftell(file);
    fseek(file, 0, SEEK_SET);
#endif
    // check for I/O errors
    if (length < 0) return status_io_error;
    // check for overflow
    size_t result = static_cast<size_t>(length);
    if (static_cast<length_type>(result) != length) return
status_out_of_memory;
    // finalize
    out_result = result;
    return status_ok;
}
```

Листинг 31. Фаззинг-обертка для функции png_convert_from_time_t

```
extern "C" int LLVMFuzzerTestOneInput(uint8_t * Fuzz_Data,
size_t Fuzz_Size){
    if (Fuzz_Size < sizeof(unsigned int)) return 0;
    size_t dyn_cstring_buffer = (size_t) (Fuzz_Size -
sizeof(unsigned int));
    //generate random array of dynamic string sizes
    size_t dyn_cstring_size[1];
    dyn_cstring_size[0] = dyn_cstring_buffer;
    //end of generation random array of dynamic string
sizes
    uint8_t * Futag_pos = Fuzz_Data;
    //GEN_CSTRING1
    char * rstr_path_ = (char *)
malloc((dyn_cstring_size[0] + 1)* sizeof(char));
    memset(rstr_path_, 0, dyn_cstring_size[0] + 1);
    memcpy(rstr_path_, Futag_pos, dyn_cstring_size[0]);
    Futag_pos += dyn_cstring_size[0];
    const char * str_path_ = rstr_path_;
```

```
    //GEN_SIZE
    unsigned int sz_options = (unsigned int)
dyn_cstring_size[0];
    //GEN_ENUM
    unsigned int e_encoding_enum_index;
    memcpy(&e_encoding_enum_index, Futag_pos,
sizeof(unsigned int));
    enum pugi::xml_encoding e_encoding = static_cast<enum
pugi::xml_encoding>(e_encoding_enum_index % 10);
    //declare the RECORD first
    pugi::xml_document Futag_target;
    //METHOD CALL
    Futag_target.load_file(str_path_ ,sz_options
,e_encoding );
    //FREE
    if (rstr_path_) {
        free(rstr_path_);
        rstr_path_ = NULL;
    }
    return 0;
}
```

Подробно можно посмотреть на сайте разработки по ссылке: https://github.com/zeux/pugixml/issues/560.

Заключение

В результате проведенной работы были получены следующие научные результаты:

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 применить генератор только к используемым интерфейсам библиотеки. Качество работы данного метода зависит от качества кода пользовательской программы. Для улучшения метода при обнаружении контекстов необходимо анализировать сложные конструкторы, 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.

Список литературы

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. - URL: https://istina.msu.ru/conferences/presentations/ 459124457/.
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).
6. Fuzzing: State of the Art / H. Liang [и др.] // IEEE Transactions on Reliability. — 2018. — Т. 67, No 3. — С. 1199—1218.
7. Boehme, M. Fuzzing: Challenges and Reflections / M. Boehme, C. Cadar, A. Roychoudhury // IEEE Software. — 2020. — Авг. — Т. PP.
8. Jerry Gao H.S. Jacob Tsao, Y. W. Testing and Quality Assurance for Component-Based Software / Y. W. Jerry Gao H.S. Jacob Tsao. — Artech House, 2003.
9. Linden, P. van der. Expert C Programming: Deep C Secrets / P. van der Linden. — Pearson, 1994.
10. Robillard, M. P. A field study of API learning obstacles / M. P. Robillard, R. DeLIne // Empirical Software Engineering. — 2011. — Дек. — Т. 16. — С. 703—732. — URL:

https://www.microsoft.com/en-us/research/publication/ field-study-api-learning-obstacles/.
11. Robillard, M. What Makes APIs Hard to Learn? Answers from Developers / M. Robillard // Software, IEEE. — 2009. — Нояб. — Т. 26. — С. 27—34.
12. Kaur, M. Software Testing and Quality Assurance / M. Kaur. — Excel Books Private Limited, 2012.
13. Feedback-Directed Random Test Generation / C. Pacheco [и др.] // 29th International Conference on Software Engineering (ICSE'07). — 2007. — С. 75—84.
14. How Do Automatically Generated Unit Tests Influence Software Maintenance? / S. Shamshiri [и др.] // 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). — 2018. — С. 250—261.
15. Fuzzing: Art, Science, and Engineering / V. J. M. Man`es [и др.] // CoRR. — 2018. — Т. abs/1812.00140. — arXiv: 1812.00140. — URL: http://arxiv.org/ abs/1812.00140.
16. Goh, B. W. H. American fuzzy lop (AFL) fuzzing / B. W. H. Goh. — 2019.
17. Serebryany, K. Continuous fuzzing with libfuzzer and addresssanitizer / K. Serebryany // 2016 IEEE Cybersecurity Development (SecDev). — IEEE. 2016. — С. 157—157.
18. Open sourcing clusterfuzz / A. Arya [и др.] // Google, Inc. Feb. — 2019.
19. Eddington, M. Peach fuzzing platform / M. Eddington // Peach Fuzzer. — 2011. — Т. 34. — С. 32—43.
20. Amini, P. Sulley fuzzing framework / P. Amini, A. Portnoy. — 2010.
21. Kozlowski, M. P. URL: http://www. powerfuzzer. com / M. P. Kozlowski // Ultimo acesso em fevereiro de. — 2012.
22. ISP-Fuzzer: Extendable Fuzzing Framework / S. Sargsyan [и др.] // 2019 Ivannikov Memorial Workshop (IVMEM). — 2019. — С. 68—71.
23. Grammar-Based Fuzzing / S. Sargsyan [и др.] // 2018 Ivannikov Memorial Workshop (IVMEM). — 2018. — С. 32—35.
24. Parmesan: Sanitizer-guided greybox fuzzing / S. O¨sterlund [и др.] // Proceedings of the 29th USENIX Conference on Security Symposium. — 2020. — С. 2289—2306.

25. Addresssanitizer: A fast address sanity checker / K. Serebryany [и др.]. — 2012.
26. Serebryany, K. ThreadSanitizer: data race detection in practice / K. Serebryany, T. Iskhodzhanov // Proceedings of the workshop on binary instrumentation and applications. — 2009. — С. 62—71.
27. Developers, L. Undefined behavior sanitizer / L. Developers. — 2017.
28. Team, C. LeakSanitizer-clang 12 documentation / C. Team. — 2020.
29. FUDGE: Fuzz Driver Generation at Scale / D. Babic [и др.] // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2019.
30. FuzzGen: Automatic Fuzzer Generation / K. Ispoglou [и др.] // 29th USENIX Security Symposium (USENIX Security 20). — USENIX Association, 08.2020. — С. 2271—2287. — URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.
31. Security Assessment and Fuzzer Improvement for Libtorrent. — URL: http://www.libtorrent.org/2020%20Q4%20Mozilla%20Libtorrent%20Report% 20Public%20Report.pdf (дата обр. 09.11.2020).
32. IntelliGen: Automatic Driver Synthesis for Fuzz Testing / M. Zhang [и др.] // 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). — 2021. — С. 318—327.
33. Louridas, P. Static code analysis / P. Louridas // Ieee Software. — 2006. — Т. 23, No 4. — С. 58—61.
34. Using static analysis for finding security vulnerabilities and critical errors in source code / A. Avetisyan [и др.] // Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). — 2011. — Т. 21.
35. ASTLOG: A Language for Examining Abstract Syntax Trees. / R. F. Crew [и др.] // DSL. Т. 97. — 1997. — С. 18—18.
36. CodeQL. — URL: https://codeql.github.com (дата обр. 30.09.2021).
37. Kremenek, T. Finding software bugs with the clang static analyzer / T. Kremenek // Apple Inc. — 2008. — С. 2008—08.
38. Dulaney, E. Linux All-in-One For Dummies / E. Dulaney. — John Wiley & Sons, 2018.

39. Stroustrup, B. A Tour of C++ / B. Stroustrup. — Pearson, 2019.
40. Reese, R. M. Understanding and using C pointers: Core techniques for memory management / R. M. Reese. — "O'Reilly Media, Inc.", 2013.
41. Kelly, M. A Case Study on Automated Fuzz Target Generation for Large Codebases / M. Kelly, C. Treude, A. Murray // CoRR. — 2019. — T. abs/1907.12214. — arXiv: 1907.12214. — URL: http://arxiv.org/abs/ 1907.12214.
42. Manolios, P. Termination Analysis with Calling Context Graphs / P. Manolios, D. Vroon // Computer Aided Verification / под ред. T. Ball, R. B. Jones. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. — C. 401—414.
43. Aho, A. V. Compilers: Principles, Techniques, & Tools / A. V. Aho, R. Sethi, J. D. Ullman. — Addison-wesley Reading, 2007. — (Addison-Wesley series in computer science). — URL: https://books.google.ru/books?id= RheqQgAACAAJ.
44. Allen, F. E. Control Flow Analysis / F. E. Allen // Proceedings of a Symposium on Compiler Optimization. — Urbana-Champaign, Illinois : Association for Computing Machinery, 1970. — C. 1—19. — URL: https://doi.org/10.1145/ 800028.808479.
45. Khedker, U. Data flow analysis: theory and practice / U. Khedker, A. Sanyal, B. Sathe. — CRC Press, 2017.
46. A comparison of data flow path selection criteria / L. A. Clarke [и др.] // International Conference on Software Engineering. — 1985.
47. Kim, D. K. Enhancing code clone detection using control flow graphs. / D. K. Kim // International Journal of Electrical & Computer Engineering (2088-8708). — 2019. — T. 9, No 5.
48. Anomaly detection using program control flow graph mining from execution logs / A. Nandi [и др.] // Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. — 2016. — C. 215—224.
49. Bruschi, D. Detecting self-mutating malware using control-flow graph matching / D. Bruschi, L. Martignoni, M. Monga // Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3. — Springer. 2006. — C. 129—143.
50. Kennedy, K. A survey of data flow analysis techniques / K. Kennedy. — IBM Thomas J. Watson Research Division, 1979.

51. Allen, F. E. A program data flow analysis procedure / F. E. Allen, J. Cocke // Communications of the ACM. — 1976. — Т. 19, No 3. — С. 137.
52. A formal evaluation of data flow path selection criteria / L. A. Clarke [и др.] // IEEE Transactions on Software Engineering. — 1989. — Т. 15, No 11. — С. 1318—1332.
53. Static analyzer Svace for finding defects in a source program code / V. Ivannikov [и др.] // Programming and Computer Software. — 2014. — Т. 40. — С. 265—275.
54. Design and development of Svace static analyzers / A. Belevantsev [и др.] // 2018 Ivannikov Memorial Workshop (IVMEM). — IEEE. 2018. — С. 3—9.
55. Debugging with GDB / R. Stallman, R. Pesch, S. Shebs [и др.] // Free Software Foundation. — 1988. — Т. 675.
56. McConnell, S. Software estimation: demystifying the black art / S. McConnell. — Microsoft press, 2006.
57. Serebryany, K. OSS-Fuzz-Google’s continuous fuzzing service for open source software / K. Serebryany // USENIX Security symposium. — USENIX Association. 2017.
58. Зыбин Р.С., Кулямин В.В., Пономаренко А.В., Рубанов В.В., Чернов Е.С. АВТОМАТИЗАЦИЯ МАССОВОГО СОЗДАНИЯ ТЕСТОВ РАБОТОСПОСОБНОСТИ.- Программирование. 2008. Т. 34. № 6. С. 64-80.
59. Зыбин Р.С., Кулямин В.В., Пономаренко А.В., Рубанов В.В., Чернов Е.С. ТЕХНОЛОГИЯ AZOV АВТОМАТИЗАЦИИ МАССОВОГО СОЗДАНИЯ ТЕСТОВ РАБОТОСПОСОБНОСТИ.- Труды Института системного программирования РАН. 2008. Т. 14. № 2. С. 83-108.
60. P. Godefroid. Compositional Dynamic Test Generation. ACM SIGPLAN Notices, 42(1):47-54, 2007. doi: 10.1145/1190215.1190226
61. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. Proc. of 29-th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 206–215, 2008. doi: 10.1145/1375581.1375607
62. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing.- Proc. of Network and Distributed System Security Symposium, pp. 151-166, 2008.

63. P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing.- Communications of ACM, 55(3):40-44, 2012. doi: 10.1145/2093548.2093564
64. E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz-testing in production. Proc. of 35-th International Conference on Software Engineering (ICSE), San Francisco, USA, 2013, pp. 122-131, doi: 10.1109/ICSE.2013.6606558
65. М. В. Мишечкин, В. В. Акользин, Ш. Ф. Курмангалеев. Архитектура и функциональные возможности инструмента ИСП Фаззер. Открытая конференция ИСП РАН им. В.П. Иванникова, 2020.
66. A. Vishnyakov, A. Fedotov, D. Kuts, A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky, S. Kurmangaleev. Sydr: Cutting Edge Dynamic Symbolic Execution. Ivannikov ISPRAS Open Conference (ISPRAS), pp. 46-54, 2020. doi: 10.1109/ISPRAS51486.2020.00014
67. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: a Fast Address Sanity Checker. Proc. of USENIX Annual Technical Conference, pp. 309-318, 2012. doi: 10.5555/2342821.2342849
68. P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation.- Proc. of International Symposium on Software Testing and Analysis (ISSTA'11), pp. 23-33, 2011. doi: 10.1145/2001420.2001424
69. I. K. Isaev, D. V. Sidorov. The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs. Programming and Computer Software, 36(40):225-236, 2010. doi: 10.1134/S0361768810040055
70. М.К. Ермаков, А.Ю. Герасимов. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей.- Труды Института системного программирования РАН, 25:29-38, 2013.
71. B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12), December 1990.
72. Potter, Bruce. "Microsoft SDL threat modelling tool." Network Security 2009.1 (2009): 15-18.

73. Weiser, M. Program slicing / M. Weiser // IEEE Transactions on Software Engineering. – 1984. -P 352−357.
74. Фаззинг-обертка для библиотеки libopus. — URL: https://github.com/HexHive/FuzzGen/blob/master/examples/libopus/opus_decode_fuzzer-LLVMFuzzerTestOneInput%2Btest_opus_api-main%2Bopus_demo-main%2Bopus_encode_regressions-regression_test%2Brepacketizer_demo-main%2Btest_opus_encode-main%2Btest_opus_padding-main%2Btrivial_example-main/opus_decode_fuzzer-LLVMFuzzerTestOneInput%2Btest_opus_api-main%2Bopus_demo-main%2Bopus_encode_regressions-regression_test%2Brepacketizer_demo-main%2Btest_opus_encode-main%2Btest_opus_padding-main%2Btrivial_example-main_fuzzer.cpp

Список рисунков

Список таблиц

Список листингов

Приложение А
Акт внедрения результата диссертационной работы

Российская Федерация
Общество с ограниченной ответственностью
НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР
«ФОБОС-НТ»

Россия, 302006, г. Орел, ул. Московская, 155
ОГРН 1065742017483, ИНН 5751030791
тел./факс (4862) 76-03-56

АКТ
о внедрении результатов диссертационного исследования
Чан Ти Тхиена
на тему «Разработка нового метода автоматизированного тестирования программных библиотек», представленной на соискание ученой степени кандидата технических наук по специальности 2.3.5 - «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Теоретические и практические результаты диссертационной работы Чан Ти Тхиена «Разработка нового метода автоматизированного тестирования программных библиотек», а именно: 1) метод автоматической генерации фаззинг-оберток для функции библиотек в условиях наличия и отсутствия контекстов использования тестируемой библиотеки; 2) программный продукт для автоматической генерации фаззинг-оберток функций библиотеки, были использованы в работе испытательной лаборатории для автоматизации создания фаззинг-оберток при выполнения фаззинг-тестирования исследуемого программного обеспечения, исходные коды которого написаны на языках программирования C/C++, а именно – актуальных версий интерпретаторов lua, php, python, nginx – выполнявшихся в интересах клиентов ООО НТЦ «Фобос-НТ» в ходе сертификационных испытаний в системе сертификации ФСТЭК России.

22.03.2023 г.

Генеральный директор
ООО НТЦ «Фобос-НТ» А. В. Гусаров

Приложение Б
Свидетельство о регистрации программы

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU 2021663344

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

(12) **ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ**

Номер регистрации (свидетельства):
2021663344

Дата регистрации: **16.08.2021**

Номер и дата поступления заявки:
2021662358 06.08.2021

Дата публикации: 16.08.2021

Контактные реквизиты:
+7-903-700-79-86,
m.kalugin@ispras.ru

Авторы:
Чан Ти Тхиен (VN),
Курмангалеев Шамиль Фаимович (RU)

Правообладатель:
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ НАУКИ ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ ИМ. В.П. ИВАННИКОВА РОССИЙСКОЙ АКАДЕМИИ НАУК (ИСП РАН) (RU)

Название программы для ЭВМ:
Futag

Реферат:
Программа - система автоматической генерации фаззинг оберток для программ на языках C/C++. С помощью статического анализа программа восстанавливает зависимости между сущностями программы: типы данных, структуры, функции, классы и т.д. Зависимости используются для автоматической генерации правильного объявления переменных и вызова функции. Поддерживаются зависимости: простой, встроенный или пользовательский тип, результат вызова функции, объект класса. Все возможности генерации аргументов рекурсивно собираются и присваиваются произвольные данные от фаззера для генерации правильного контекста вызова функции. Программа обнаруживает риски безопасности, такие как переполнение буфера, обращение к некорректному адресу памяти, утечка памяти и обеспечивает: генерацию фаззинг оберток для разных платформ, компиляцию оберток с санитайзерами, запуск и сбор результатов, интеграцию с системой Svace. Тип ЭВМ: IBM PC-совмест. ПК; ОС: Windows/Linux.

Язык программирования: C++

Объем программы для ЭВМ: 33 МБ