Утверждаю
Директор Института СПИНТех
НИУ МИЭТ
Проф. _____/Гагарина Л.Г./
«____»_____ 2020 г.

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет «Московский институт электронной техники»

Уманский Александр Александрович

Разработка программного модуля для анализа программ на языках C и C++ на недекларированные возможности

Специальность 09.03.04 —
«Программная инженерия»

Отчет по производственной практике
студента института СПИНТЕХ

Научный руководитель:
кандидат технических наук, доцент
Кононова Александра Игоревна

_____

Студент:
Уманский Александр Александрович

_____

Москва, г. Зеленоград — 2020

Содержание

Стр.

Стр.

Список сокращений и условных обозначений

НДВ	Недекларированные возможности
ПМ	Программный модуль
БД	База Данных
ИСПДН	Информационная система персональных данных
ПО	Программное обеспечение
ЯП	Язык программирования
GUI	Graphical User Interface
IDE	Интегрированная среда разработки
JSON	Формат описания структур данных в текстовом виде ключ $\rightarrow$ значение
PID	Уникальный идентификатор процесса в ОС
TDD	Test-driven development
ПМ АПНДВ	Программный модуль анализа на недекларированные возможности

Словарь терминов

Кросс-платформенный : Программа, которая может запускаться на различных операционных системах и/или архитектурах процессоров

Программная закладка : Подпрограмма, либо фрагмент исходного кода, скрытно внедренный в исполняемый файл

Динамическая трасса : Дерево вызванных программой функций во время конкретного ее исполнения

Статическая трасса : Дерево функций программы, которые объявлены для вызова

Отладчик : Программа, в контексте которой запускается другая программа для локализации и устранения ошибок в контролируемых условиях

Отладка : Процесс локализации и устранения ошибок программы в контролируемых условиях

Удаленная отладка : Процесс отладки программы, запущенной вне контекста отладчика

Препроцессор : Программа-макропроцессор, обрабатывающая специальные директивы в исходном коде и запускающаяся до компилятора

Препроцессирование : Процесс обработки исходного кода препроцессором

Открытое ПО : ПО с открытым исходным кодом, который доступен для просмотра, изучения и изменения

Сериализация : Процесс перевода определеного типа данных программы в некоторый формат

Десериализация : Процесс перевода данных, находящихся в некотором формате, во внутренний тип данных программы

Скрипт : Программа, обычно на интерпретируемом языке программирования, выполняющая конкретное действие

Сигнатура функции : Объявление функции, в которое входит имя функции, количество входных параметров и их тип

Сборка : Процесс компиляции, линковки и публикации программного обеспечения из исходных кодов

Рефакторинг : Процесс улучшения кода без введения новой функциональности. Результатом является чистый код с улучшенным дизайном

Релизная сборка : Сборка программы происходит без отладочных символов,

обычно с использованием техник оптимизации кода

Терминал : То же, что и консоль

Сверхвысокоуровневый ЯП : Классификация языков программирования, к данной категории относятся языки программирования, позволяющие описать задачу не на уровне «как нужно сделать», а на уровне «что нужно сделать»

Source-to-source : Компиляция исходного кода некоторого языка в исходный код другого языка. Во время компиляции языка данным способом может происходить несколько итераций преобразования, пока последний язык в цепочке преобразований не будет скомпилирован в машинный код или интерпретирован

Введение

Сертификация – процесс подтверждения соответствия характеристик товара определенным стандартам.

Сертификация не является универсальным способом решения всех существующих проблем в области информационной безопасности, однако сегодня это единственный реально функционирующий механизм, который обеспечивает независимый контроль качества средств защиты информации. При грамотном применении механизм сертификации позволяет достаточно успешно решать задачу достижения гарантированного уровня защищенности автоматизированных систем.

Отсутствие недекларированных возможностей в скомпилированном объектном файле является ключевым аспектом сертификации ПО. Сертификация программного обеспечения необходима для подтверждения требований заказчика к защите информации, к выполнению функциональных и технических задач и к обеспечению работы ПО в целом.

Но существуют опасения, возникающие не на пустом месте, что на любом из этапов сборки программы из исходных кодов, в ней может появиться программная закладка [1; 2]

Чтобы подобные ситуации исключить, применяется техника статического анализа исходных кодов, динамического анализа – анализа пройденных программой трасс и последующее сравнение результатов обоих анализов.

На данный момент не существует открытых программных решений, позволяющих проводить сертификацию программного обеспечения в описанном ранее формате. Самое близкое по назначению ПО это статические анализаторы [3], и так использующиеся как составная часть в процессе сертификации. Помимо них существует свободное программное обеспечение от корпорации Microsoft – Microsoft Application Inspector [4], но оно взаимодействует только с исходными кодами программы, распознавая паттерны и назначение функций. Помимо свободных программ существует утилита анализа ядра Linux от ООО Фирма «Анкад». В ней проводится статический, динамический и сравнительные анализы, но программа не умеет работать с чем либо, кроме ядра Linux и сертифицировать что-либо еще с помощью нее не получится.

Рисунок 1 — Процесс проведения сертификации раньше

Получается, что на рынке невозможно найти комбинированных решений, с помощью кторых было бы возможно провести процесс сертификации любого ПО. Для каждого конкретного проекта приходится использовать различные статические анализаторы, динамические анализаторы, что приводит к дублированию, по своей сути, кода и выполняемых действий, которые нужны для сертификации ПО. Это ведет к разрастанию кодовой базы компании и нарастающим трудностям в последующей поддержке каждого отдельного решения, что в свою очередь ведет к увеличению затрат компании.

Чтобы унифицировать разрабатываемое ПО для сертификации, было решено разбить ПМ АПНДВ на модули, разделенные по ответственности и не знающие друг о друге. Это обеспечивает удобство в редактировании, замене и изменении модулей, а при сохранении формата выдаваемой информации – инкапсуляцию изменений только на конкретном модуле.

Так как модули не знают друг о друге, то и работают они в условиях ограниченной информации. Модуль статического анализа обрабатывает только исходные коды, выдавая список статических вызовов. Модуль динамического анализа работает с программой без отладочных символов, собирая информацию на уровне машинных инструкций.

Данный подход помогает приблизить процесс сертификации к «боевым» условиям

Целью данной работы является унификация проведения процесса сертификации программ написанных на языках программирования C/C++.

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

1) анализ текущих программных решений для проведения статического и динамического анализа, выбор наиболее подходящего в плане универсальности и расширяемости;
2) анализ языков программирования для выбора наиболее производительного, надежного и легкоподдерживаемого;
3) разработка алгоритма работы программы;
4) разработка структур данных;
5) разбиение функционала ПМ АПНДВ на модули по ответственности;
6) разработка алгоритма передачи данных между модулями.

Практическая значимость проекта состоит в унификации процесса сертификации ПО на отсутствие НДВ.

Рисунок 2 — Процесс проведения сертификации сейчас

Полный объём отчета составляет 64 страницы, включая 16 рисунков и 14 таблиц. Список литературы содержит 56 наименований.

Раздел 1. Исследовательский раздел

Сертификация программного обеспечения проводится, когда необходимо подтвердить соответствие разрабатываемой продукции требованиям защиты информации. Частым объектом сертификации является ПО, разработанное:

- для обеспечения информационной безопасности;
- для осуществления коммуникации между людьми или программно-аппаратными комплексами;
- для техники военного назначения;
- крупными разработчиками программного обеспечения.

Подтверждение безопасности программного обеспечения – важный этап в продвижении программного продукта.

1.1 Процесс сертификации ПО на отсутствие НДВ

Сертификационная процедура состоит из следующих этапов:

1) готовность документации ПО, доступность исходных текстов;
2) опредление объема исходных текстов, подлежащих анализу;
3) обращение заявителя в испытательную лабораторию с собранной информацией;
4) анализ документации;
5) разработка «Программы и методик проведения сертификационных испытаний»;
6) проведение испытаний;
7) экспертиза результатов.

Сертификация должна выявить присутствие в исполняемом файле недекларированных возможностей, которые могут являться как злым умыслом разработчиков компилятора, линкера и других вспомогательных программ, так и методами оптимизации ПО, которые применяются для более рационального потребления ресурсов программой.

Впервые теоретическая возможность создания таких программ была описана создателем языка Си, инженером Bell Labs – Кеном Томпсоном в 1984 году, в журнале ACM под названием «Reflections on Trusting Trust» («Размышления о

доверии») [2]. В ней была описан компилятор, содержащий в себе следующий функционал:

- компилятор знает, когда компилирует сторонние программы, а когда себя или другой компилятор;
- при компиляции любой программы, в исполняемый файл внедряется некий вредоносный код;
- если компилятор компилирует себя или другой компилятор, то он добавляет функционал внедрения вредоносного кода в другие компиляции программ и компилятора;

Такой компилятор разбивает уверенность в принципе, который можно описать как «я не доверяю скомпилированному бинарному файлу, я все собираю сам», так как нельзя быть полностью уверенным, что в программе не появилось НДВ на стадии компиляции.

Эта статья так и оставалась бы чистым размышлением, если бы в 2009 году специалисты из лаборатории по информационной безопасности – SophosLabs не обнаружили компилятор Delphi [1], умеющий добавлять в исполняемые файлы вредоносную составляющую. За день специалисты получили в свое распоряжение более 3000 уникальных исполняемых файлов, зараженных W32/Induc-A. Это является серьезной угрозой безопасности, так как распространителем вредоносных файлов может быть легитимный источник, например известная компания по производству программного обеспечения.

Выявить данные расхождения между необработанными исходными кодами и поведением программы во время исполнения позволяет разработанный мной программный модуль.

Дадим определение термину «Недекларированные возможности»:

Недекларированные возможности [5] — намеренно измененная часть ПО, с помощью которой можно получить незаметный несанкционированный доступ к безопасной компьютерной среде.

1.2 Классификация НДВ

Классифицировать НДВ можно несколькими способами, в зависимости от их целей для компрометации и способу использования.

1.2.1 По применению

Использование НДВ может реализовываться в:

– перехвате данных;
– подмене данных;
– выводе компьютерной системы из строя;
– полном доступе к удаленной компьютерной системе.

Причем, при полном доступе к компьютерной системе, вредоносные программы программы могут быть использованы злоумышленниками для всех вышеперечисленных целей.

1.2.2 По целям

Использование НДВ может быть направлено на:

– **Персональные компьютеры и рабочие станции**
  Целью могут быть как персональные компьютеры широкого числа пользователей, так и отдельные рабочие станции, которые могут являться точкой входа в защищенную компьютерную систему, так и использоваться для перехвата важной информации;
– **Серверы**
  Серверы обслуживают большое количество клиентов, а значит проникновение на сервер может существенно повлиять на работу всех компьютеров, работающих с данным сервером;
– **Встраиваемые системы**
  Благодаря постоянному удешелению микроконтроллеров и периферийных устройств, все больше и больше повседневных вещей обзаводятся «умной» функциональностью. Погоня производителей за прибылями отражается на безопасности прошивок умных устройств;
– **Промышленные компьютеры**
  Программные закладки в такие системы чреваты шпионажем или диверсией, как, например вирус [6]. Не смотря на то, что данная программа является вирусом, а не программой с НДВ, случившееся ярко показывает реальное применение подобных техник для деструктивных действий.

1.3 Степень опасности НДВ

Для определения опасности НДВ будем пользоваться следующими нормативными документами [7]:

– приказ ФСТЭК России от 18 февраля 2013 г. № 21;

– федеральный закон "О персональных данных"от 27.07.2006 N 152-ФЗ.

Тип угроз безопасности персональных данных определяется в зависимости от комбинаций критичности угроз в ИСПДн (табл. 1):

– наличием НДВ в системном программном обеспечении (ПО), используемом в ИСПДн;

– наличием НДВ в прикладном ПО, используемом в ИСПДн.

Согласно п.6 Требований к защите персональных данных при их обработке в ИСПДн, утвержденных постановлением Правительства РФ от 01.11.2012 №1119 [8], установлены 3 типа актуальных угроз безопасности персональных данных. Самый низкий тип угроз – третий, самый высокий – первый.

Таблица 1 — Тип актуальных угроз

Угрозы	Тип актуальных угроз		
	1 Тип	2 Тип	3 Тип
Наличие НДВ в системном ПО, используемом в ИСПДн	критично	некритично	некритично
Наличие НДВ в прикладном ПО используемом в ИСПДн	критично или некритично	критично	некритично

Порядок определения актуальных угроз безопасности персональных данных в ИСПДн осуществляется в соответствии с Методикой определения актуальных угроз безопасности персональных данных при их обработке в информационных системах персональных данных, утвержденных ФСТЭК России, 2008 год.

Актуальной считается угроза, которая может быть реализована в ИСПДн и представляет опасность для персональных данных. Подход к составлению перечня актуальных угроз состоит в следующем. Для оценки возможности реализации угрозы применяются два показателя:

– Y_1 - уровень исходной защищенности ИСПДн;

– Y_2 - частота (вероятность) реализации рассматриваемой угрозы;

Коэффициент реализуемости угрозы Y определяется соотношением:

$$Y = \frac{Y_1 + Y_2}{20}$$

По значению коэффициента реализуемости угрозы Y интерпретация реализуемости угрозы следующим образом:

- если $0 \leqslant Y \leqslant 0.3$, то возможность реализации угрозы признается низкой;
- если $0.3 < Y \leqslant 0.6$, то возможность реализации угрозы признается средней;
- если $0.6 < Y \leqslant 0.8$, то возможность реализации угрозы признается высокой;
- если $Y > 0{,}8$, то возможность реализации угрозы признается очень высокой.

Далее оценивается опасность каждой угрозы. Этот показатель имеет три значения:

- низкая опасность – если реализация угрозы может привести к незначительным негативным последствиям для субъектов персональных данных;
- средняя опасность – если реализация угрозы может привести к негативным последствиям для субъектов персональных данных;
- высокая опасность – если реализация угрозы может привести к значительным негативным последствиям для субъектов персональных данных.

Затем осуществляется выбор из общего перечня угроз безопасности тех, которые относятся к актуальным для данной ИСПДн, в соответствии с правилами, приведенными в табл. 2

Таблица 2 — Правила отнесения угрозы безопасности персональных данных к критичной

Возможность реализации угрозы	Показатель опасности угрозы		
	Низкая	Средняя	Высокая
Низкая	некритичная	некритичная	критичная
Средняя	некритичная	критичная	критичная
Высокая	критичная	критичная	критичная
Очень высокая	критичная	критичная	критичная

Как видно из таблицы, на критичность угрозы НДВ влияет не только её степень опасности, но и вероятность проявлении деструктивного эффекта НДВ в работающем ПО.

Затем выносится решение о проведении анализа ПО на НДВ в процессе сертификации или его игнорирование, как некритичного.

Сейчас этап анализа программы на НДВ происходит вручную:

1) с помощью специального ПО проводят статический анализ исходных кодов программного проекта;
2) с помощью отладчиков, профилировщиков или эмуляторов проводят динамический анализ исполняемого файла, сохраняя трассы выполнения;
3) данные статического и динамического анализа приводятся к общему виду;
4) с помощью программы сравнения ищутся несовпадения или их отсутствие.

1.4 Обзор программных решений для сертификации ПО на отсутствие НДВ

На сегодняшний день не существует в открытом доступе комплексных разработок по сертификации программного обеспечения на предмет наличия НДВ. Однако, существуют программы, специализирующиеся отдельно на анализе исходных кодов и отдельно исполняемого файла. В ООО Фирма «Анкад» существует узконаправленный пакет утилит для анализа ядра Linux и пакетов пользовательского пространства, но он не приспособлен для анализа каких-либо других программ. Он будет упомянут как в сравнении статических анализаторов, так и динамических, потому что умеет выполнять все виды анализов. Так как ПМ АПНДВ будет совмещать и расширять функционал данных программных средств, то рассмотрим их по отдельности.

1.4.1 Сравнение статических анализаторов

Статический анализ исходных кодов проводится без надобности в сборке и запуске анализируемой программы.

Статические анализаторы, в основном, явно или побочно используются разработчиками через программы-линтеры и компиляторы для обнаружения нежелательного поведения или нарушения стиля программирования, не связанного с корректностью исходного кода грамматике языка. Данные статические

анализаторы будут сообщать, например, если выражение потенциально может вызывать переполнение стека или условное выражение всегда будет исполнять только одну из своих веток.

Не смотря на всю полезность данных статических анализаторов, в контексте сертификации программного обеспечения на НДВ они имеют малое практическое применение и скорее относятся к повышению качества или читаемости кода.

Поэтому рассмотрим статические анализаторы, специализирующиеся на созданиях карт исходного кода.

Таблица 3 — Сравнительная таблица статических анализаторов

Свойства \ Название программы	Microsoft Application Inspector	SCI Tools Understand [9]	GNU cflow [10]	Kernel analyzer
Кросс-платформенность	Да	Да	Да	Да
Открытость исходного кода	Да	Нет	Да	Нет
Препроцессирование кода C/C++	Нет	Да	Да	Да
Представление препроцессорных директив как вызов функций	Нет	Нет	Да	Нет
Создание графа вызовов	Нет	Да	Да	Да
Создание обратного графа вызовов	Нет	Да	Да	Нет
Бесплатность	Да	Нет	Да	Да
Графический интерфейс	Нет	Есть	Нет	Нет

Microsoft Application Inspector

Задача Microsoft Application Inspector – Систематическая и масштабируемая идентификация функций исходного кода. Анализатор написан на .NET Core [11], а это значит, что программа будет работать на всех платформах, для которых реализован .NET Core: Windows, Linux и macOS.

Распознает паттерны поведения функций не только в 34 языках, но так же и в их смешениях – когда взаимодействующие части программы написаны на разных языках. Умеет замечать отличия в поведении между различными версиями инспектируемого программного модуля.

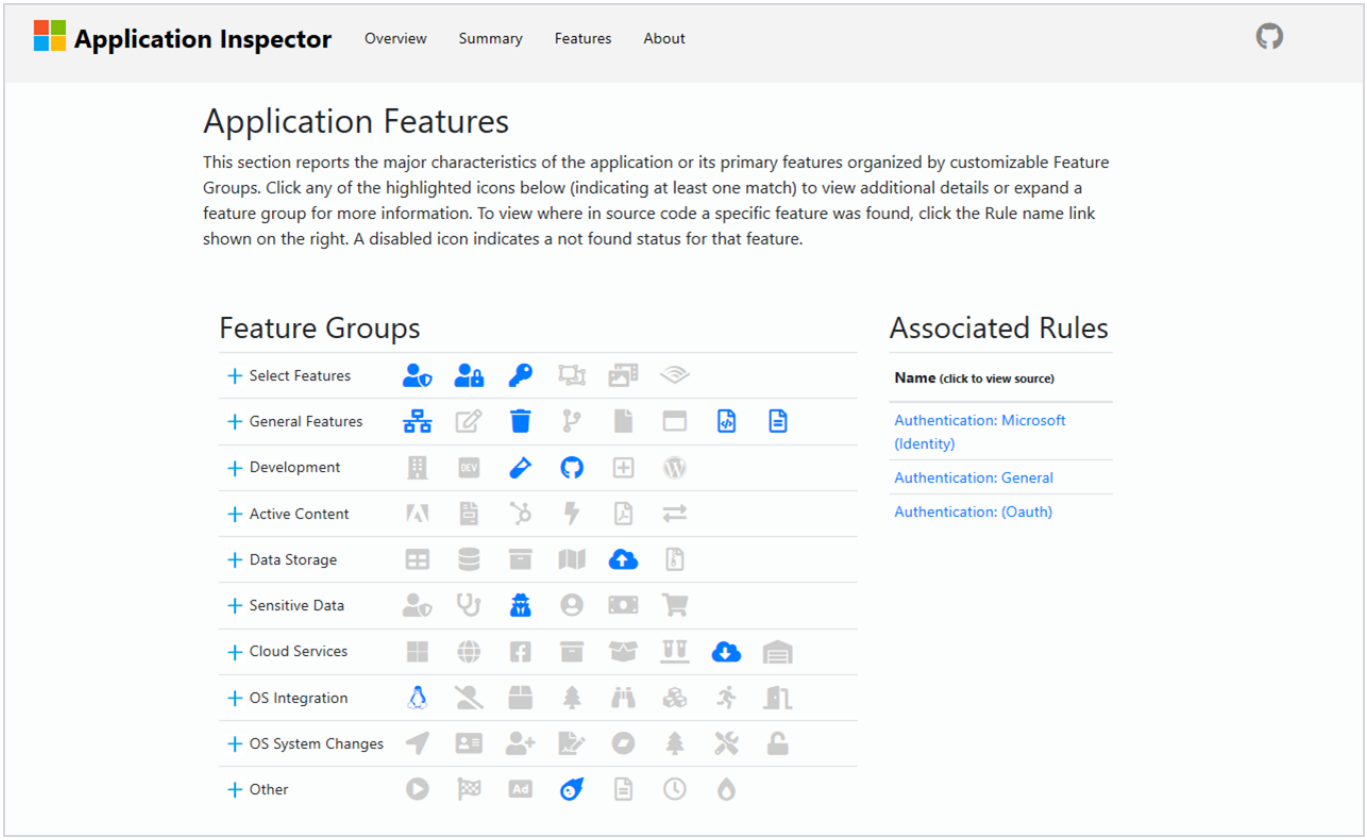


Рисунок 1.1 — Отчет Microsoft Application Inspector

Является бесплатным ПО, с открытым исходным кодом. Позволяет защититься от НДВ только на переднем плане, так как анализирует исходные коды подключенных модулей, но бессилен при появлении НДВ на этапе компиляции.

В качестве недостатков для использования как составной части ПМ АПНДВ можно отметить предметный анализ функций, который ничего не говорит о последовательности их вызова.

SCI Tools Understand

SCI Tools Understand – кросс-платформенный, быстрый статический анализатор больших объемов кода, имеющий хорошие возможности в визуализации отношений модулей программы, имеет встроенный расчет различных метрик программного кода.

Поддерживает около 20 языков программирования, а так же распознает код, написанный под разными стандартами. Недостатки SCI Tools Understand – платность и закрытый исходный код. Но купив лицензионную копию программы, пользователь получает возможность писать скрипты манипуляции БД анализируемого проекта, генерирования отчетов и собственных метрик.

GNU cflow

Быстрый и минималистичный статический анализатор, с открытым исходным кодом, позволяющий создавать как прямые, так и обратные графы вызовов. Командный интерфейс приближен к командному интерфейсу компилятора. Поддерживает языки C и C++, а так же LEX и YACC. К достоинствам так же можно отнести удобный и емкий формат отчета, который легко разбирать регулярными выражениями.

Kernel analyzer

Статический анализ утилита анализа ядра линукс проводит через компиляцию исходного кода ядра компиляторами clang и gcc, с последующим разбором сгенерированных ими абстрактных синтаксических деревьев, что является гарантированно честным, но не самым производительным способом генерации статических диаграмм.

Помимо статического и динамического анализа имеет сравнительный анализ, а так же команды для проверки специфических ситуаций, таких как накапливание информации в ядре и трассирование процесса запуска системы.

Вывод

Так как ПМ АПНДВ ориентирован на анализ C/C++ программ, то проанализировав табл. 3 приходим к выводу, что функционал Microsoft Application Inspector не покрывает нужные сценарии использования, а SCI Tools Understand не подходит из-за своей закрытости и платности, Kernel analyzer – узконаправленности. Единственный возможнный выбор – GNU cflow.

1.4.2 Сравнение динамических анализаторов

Динамический анализ программного обеспечения может проводиться в реальном или эмулированном окружении. Проведения динамического анализа, требует от себя тестирования максимального количества вариантов ввода данных для прохождения исследуемой программой как можно большего количества путей генерации выходных данных. Динамический анализ, необходимый в рамках сертификации программного обеспечения, может проводиться программами различного назначения, до тех пор, пока программа умеет сама или побочно создавать динамическую карту вызовов.

Рассмотрим программы, потенциально пригодные для проведения динамического анализа.

Таблица 5 — Сравнительная таблица программ для динамического анализа

Название программы / Свойства	Gcov [12]	GDB [13]	QEMU [14]	Kernel analyzer
Кросс-платформенность	Да	Да	Да	Да
Открытость исходного кода	Да	Да	Да	Нет
Возможность анализировать память	Нет	Да	Да	Да
Возможность программно управлять	Нет	Да	Да	Нет
Возможность создавать собственные команды	Нет	Да	Нет	Нет
Возможность удаленной отладки	Нет	Да	Нет	Нет
Бесплатность	Да	Да	Да	Да
Поддержка отладки программ, написанных не для x86 архитектуры	Да	Да	Да	Да
Графический интерфейс	Есть	Есть	Есть	Нет

Gcov

Утилита для создания покрытия кода, входит в пакет GCC (GNU Compiler Collection). Генерирует новые исходные файлы, в которых на каждой строчке указано количество раз, сколько была вызвана та или иная функция. Больше не генерирует никакой информации, и работает только с программами, скомпилированными с помощью GCC. С помощью графического интерфейса lcov можно создавать html отчеты рис. 1.2 о пройденных трассах.

GNU Debugger

Отладчик GDB впервые увидел свет в 1986 году и за прошедшие годы обзавелся большим количеством поддерживаемых архитектур процессоров, самые известные:

– Alpha;

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - example/methods - iterate.c (source / functions)	Lines:	8	8	100.0 %
Test:	Basic example (view descriptions)	Functions:	1	1	100.0 %
Date:	2019-03-04 16:39:23	Branches:	4	4	100.0 %
Legend:	Lines: hit not hit | Branches: + taken - not taken # not executed				

```
        Branch data     Line data    Source code
       1                :            : /*
       2                :            :  *  methods/iterate.c
       3                :            :  *
       4                :            :  *  Calculate the sum of a given range of integer numbers.
       5                :            :  *
       6                :            :  *  This particular method of implementation works by way of brute force,
       7                :            :  *  i.e. it iterates over the entire range while adding the numbers to finally
       8                :            :  *  get the total sum. As a positive side effect, we're able to easily detect
       9                :            :  *  overflows, i.e. situations in which the sum would exceed the capacity
      10                :            :  *  of an integer variable.
      11                :            :  *
      12                :            :  */
      13                :            :
      14                :            : #include <stdio.h>
      15                :            : #include <stdlib.h>
      16                :            : #include "iterate.h"
      17                :            :
      18                :            :
      19                :          3 : int iterate_get_sum (int min, int max)
      20                :            : {
      21                :            :         int i, total;
      22                :            :
      23                :          3 :         total = 0;
      24                :            :
      25                :            :         /* This is where we loop over each number in the range, including
      26                :            :            both the minimum and the maximum number. */
      27                :            :
      28        [ +  + ]:      67548 :         for (i = min; i <= max; i++)
      29                :            :         {
      30                :            :                 /* We can detect an overflow by checking whether the new
      31                :            :                    sum would become negative. */
      32                :            :
      33        [ +  + ]:      67546 :                 if (total + i < total)
      34                :            :                 {
      35                :          1 :                         printf ("Error: sum too large!\n");
      36                :          1 :                         exit (1);
      37                :            :                 }
      38                :            :
      39                :            :                 /* Everything seems to fit into an int, so continue adding. */
      40                :            :
      41                :      67545 :                 total += i;
      42                :            :         }
      43                :            :
      44                :          2 :         return total;
      45                :            : }
```

Рисунок 1.2 — Отчет lcov

- ARM;
- AVR;
- H8/300;
- Altera Nios/Nios II;
- System/370;
- System 390;
- X86 и X86-64;
- IA-64 "Itanium";
- Motorola 68000;
- MIPS;
- PA-RISC;
- PowerPC;
- SuperH;

– SPARC;
– VAX.

Существует под все популярные операционные системы. Часто используется в различных IDE в качестве отладчика из за своей надежности и текстового интерфейса GDB/MI (MI расшифровывается как Machine Interface), позволяющего использовать отладчик в качестве компонента некой большой системы. К достоинствам можно отнести возможность описания сценария отладки в командном файле, с последующим исполнением его GDB, расширение возможностей отладчика через программирование на встроенном интерпретаторе python, создание макрокоманд с помощью уже существующих, а так же удаленную отладку.

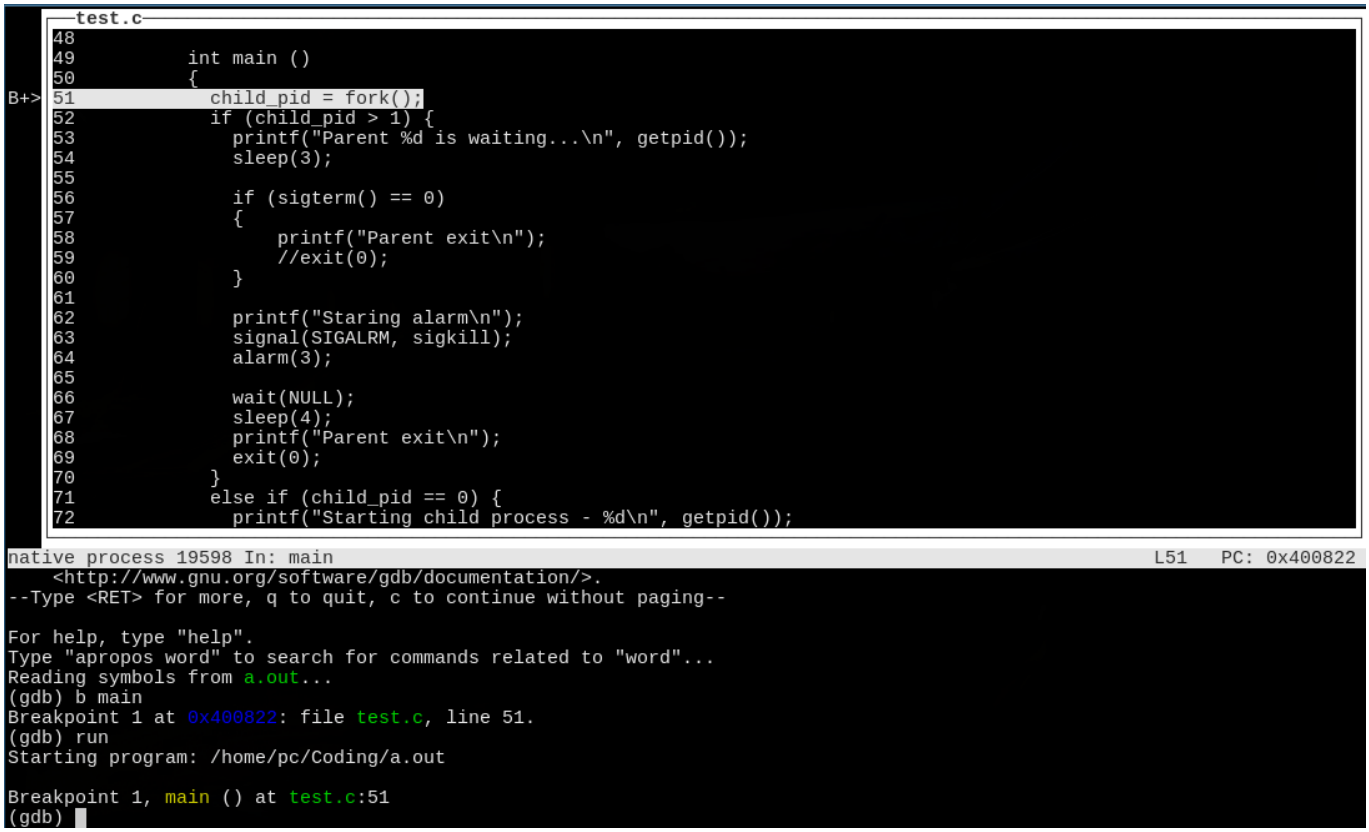


Рисунок 1.3 — Терминальный интерфейс GDB

QEMU

QEMU – Быстрый эмулятор процессоров, поддерживает множество процессорных архитектур, предоставляет возможность сохранять сгенерированный машинный код. Существует под все популярные операционные системы. Помимо эмуляции процессора, QEMU может эмулировать и периферийные устройства: сетевые карты, жесткие диски, видео карты, PCI, USB и др. К недостаткам можно отнести медленную, по сравнению с отладчиком работу, так как эмулятору

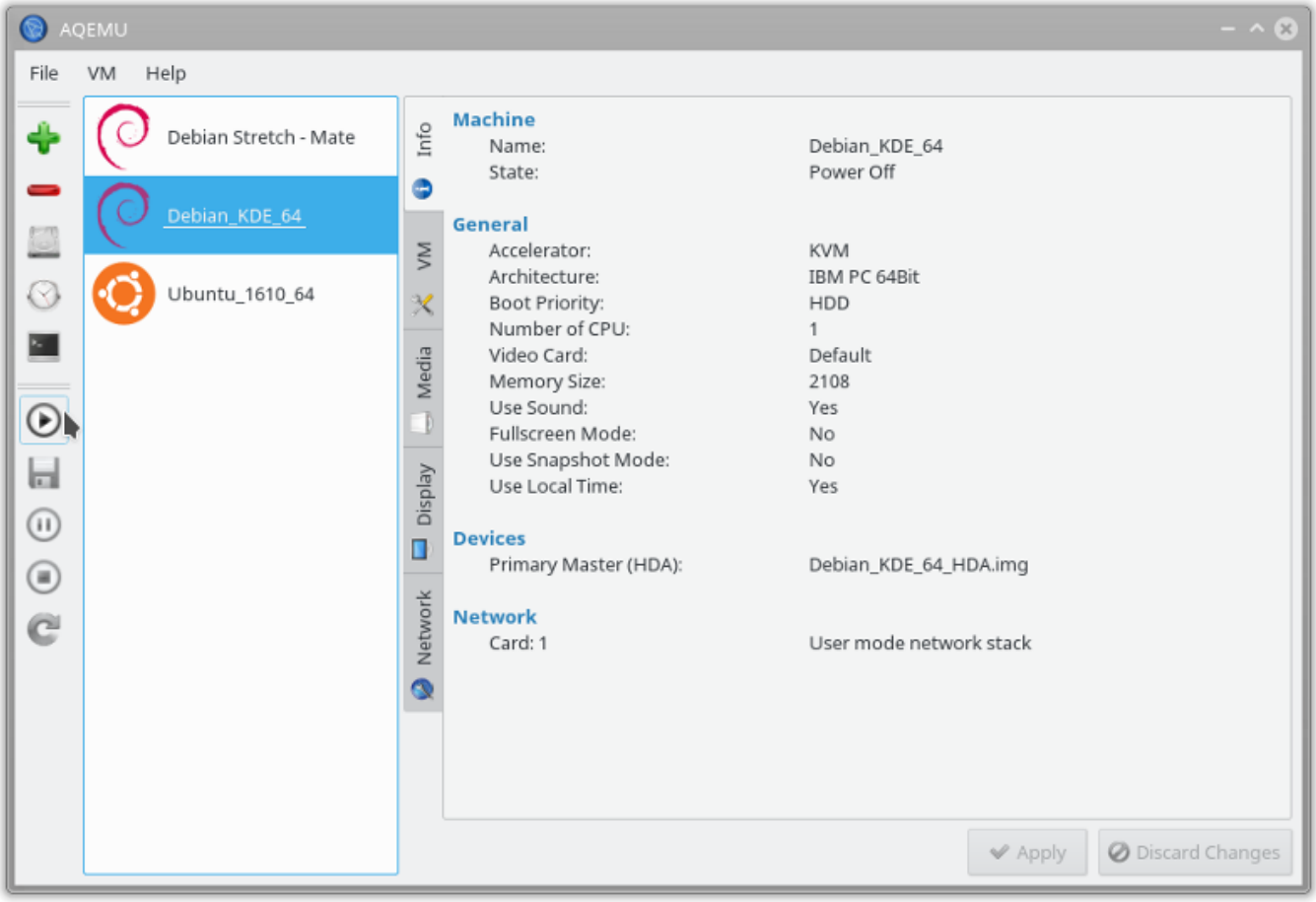


Рисунок 1.4 — Графическая оболочка AQEMU для эмулятора QEMU

приходится преобразовывать каждую инструкцию запущенной программы в машинный код процессора, на котором он запущен.

Работает это следующим образом: инструкции запущенной внутри QEMU программы конвертируются в промежуточный, платформонезависимый код при помощи интерпретатора TCG (Tiny Code Generator), затем этот платформонезависимый код компилируется уже в целевые машинные инструкции.

Kernel analyzer

В динамическом анализе утилита полагается на QEMU и проводит его через разбор call-инструкций в скомпилированном TCG коде.

Вывод

Так как для более точного выполнения задачи сертификации будет полезно получать информацию времени выполнения программы, такую, как:

1) значения регистров перед вызовом функции;
2) состояние стека перед вызовом функции;

3) стек вызовов;
4) экспертиза результатов;
5) информацию о сегментах и функциях в них определенных.

А так же расширять возможность динамического анализатора с помощью скриптов, то из табл. 5 следует, что удобнее всего это можно будет сделать с помощью отладчика GDB, нежели эмулятора QEMU или генератора покрытия кода Gcov. Kernel analyzer – по тем же причинам, что и QEMU.

1.5 Постановка задачи ВКР

На основе изложенного в разд. 1.1-разд. 1.4 сформированы следующие цели и задачи ВКР. Цель: унификация процедуры сертификации программного обеспечения, написанного на языках C и C++.

Задачи:

1) исследование предметной области (рассмотрено в разд. 1.1);
2) сравнительный анализ существующих программных решений (рассмотрено в разд. 1.4.1-разд. 1.4.2);
3) выбор языка и среды разработки;
4) разработка схемы данных ПМ АПНДВ;
5) разработка схемы алгоритма ПМ АПНДВ;
6) программирование ПМ АПНДВ;
7) отладка и тестирование ПМ АПНДВ;
8) разработка документации к ПМ АПНДВ.

1.6 Выводы по разделу

В исследовательском разделе была рассмотрена предметная область процесса сертификации программного обеспечения, теоретические и нормативные-правовые обоснования необходимости проведения данной процедуры, а так же представлены последствия игнорирования возможности внедрения НДВ через доверенные программы – компиляторы.

Проведен анализ существующих решений данной проблемы, из которого был сделан вывод о том, что программы, аналогичной по универсальности ПМ АПНДВ нет на рынке.

Выбраны сторонние свободные программы в качестве компонентов ПМ АПНДВ.

Обоснована актуальность разработки ПМ АПНДВ.

Поставлены задачи для дальнейшей разработки ПМ АПНДВ.

Раздел 2. Конструкторский раздел

2.1 Обоснование выбора языка программирования и среды разработки

Для удобной, быстрой и эффективной, как по срокам выполнения, так и по качеству конечного продукта, разработки ПМ АПНДВ потребуются правильные инструменты – язык программирования, на котором легче всего описать решение данной задачи и среда разработки, не только поддерживающая данный язык, но и позволяющая эффективно с ним работать.

2.1.1 Сравнение языков программирования

Для разработки ПМ АПНДВ понадобится сверхвысокоуровневый язык с кросс-платформенной стандартной библиотекой, который позволит точно и лаконично описать этапы анализа, а так же имеющий высокую скорость исполнения, для анализа больших объемов исходного кода и исполняемых файлов.

Рассмотрим подробно каждый из представленных в таблице языков.

Таблица 7 — Сравнительная таблица языков программирования

Язык программирования / Свойства	Nim [15]	Python [16]	Perl [17]	C/C++
Сверхвысокоуровневость	Да	Да	Да	Нет
Компилируется в машинный код	Да	Нет	Нет	Да
Количество функции в стандартной библиотеке	5585	638	1338	1224
Портируемость	Есть	Есть	Есть	Есть, но неудобная
Встроенная генерация документации	Есть	Есть	Есть	Нет
Статическая типизация	Есть	Нет	Нет	Есть
Автоматическое управление памятью	Есть	Есть	Есть	Есть
Обобщенное программирование	Есть	Есть	Есть	Есть
Мета-программирование	Есть	Есть	Есть	Есть
Опыт использования	Есть	Есть	Нет	Есть

C++

Мультипарадигменный высокоуровневый язык программирования, разработанный в 1983 году Бьёрном Страуструпом. Является практически полным надмножеством языка C. Статически типизирован.
Отличается высокой производительностью и неплохой гибкостью при написании кода. К минусам языка можно отнести сложность освоения и перегруженность «наследием» 80-х годов прошлого века, а так же низкую скорость компиляции, по сравнению с предшественником – C.
Портируемость языка на различные платформы обеспечивается пере- или кросс-компиляцией исходного кода под нужную платформу.

Python

Мультипарадигменный сверхвысокоуровневый язык программирования, разработанный в 1991 году Гвидо Ван Россумом. Является интерпретируемым языком, имеет слабую динамическую типизацию, что позволяет легко писать обобщенный код и использовать мета-программирование, но так же ведет к трудноулавливаемым ошибкам. Негативное влияние можно сгладить с помощью указания типов при объявлении перемнных и аргументов функций, а так же программы, проверяющей эти типы – линтера. Например pylint [18] или pyflakes [19].
Благодаря своей популярности, python так же портирован на большое количество платформ. Большим плюсом языка является его обширная стандартная библиотека, позволяющая легко писать комплексные приложения, не прибегая к установке дополнительных библиотек – такие программы, как и сам python, следуют философии «в комплекте с батарейками» («batteries included» [20]), суть которой заключается в самодостаточности программ. Помимо этого вместе с python поставляется менеджер пакетов pip [21], позволяющий удобно устанавливать требуемые библиотечные модули вместе с зависимостями.
К минусам языка можно отнести медлительность эталонного интерпретатора языка – cpython [22]. Код, исполняемый им, в определенных задачах медленнее кода на C в сотни раз. Не смотря на то, что есть более быстрые интерпретаторы: PyPy [23], Jython [24], Iron Python [25], они не смогут достичь скорости исполнения программ, компилируемых в машинный код.

На данный момент существует две, между собой несовместимые, версии языка: python 2, поддержка которого закончилась 1 января 2020 г. и python 3.

Perl

Мультипарадигменный сверхвысокоуровневый язык программирования, разработанный в 1987 году Ларри Уоллом. Является интерпретируемым языком, имеет слабую динамическую типизацию.
Полное название языка – «Practical Extraction and Report Language» («Практический Язык для Извлечения Данных и Составления Отчётов»), отражает его суть: в языке реализованы обширные возможности для работы с текстом, в синтаксис интегрированы регулярные выражения, как и в языках, которые оказали на него наибольшее влияние – AWK [26] и sed [27]. Но это же и я является его слабой частью, так как Perl скорее предназначен для однострочных команд в терминале, как AWK и sed.

Nim

Мультипарадигменный сверхвысокоуровневый язык программирования, разработанный в 2004 году Андреасом Румпфом. Является компилируемым языком, имеет строгую статическую типизацию.
Заметно, что на синтаксис языка повлиял Python, что сделало его выразительным и понятным. Язык использует промежуточную компиляцию, которая несколько замедляет процесс компиляции программ, но позволяет запускать nim-программы на различных платформах. На данный момент поддерживается компиляция в JavaScript [28] и оптимизированный C-код с несколькими моделями управления памятью:

- Сборщики мусора, основанные на:
    1) подсчете ссылок;
    2) подсчете ссылок с оптимизацией move-семантикой [29];
    3) boehm [30];
    4) go [31];
- ручном освобождении памяти;
- модель, в которой вся выделенная память высвобождается только по завершению программы (не рекомендуется к использованию).

Компиляции Nim в C означает не только высокую скорость работы, но и прозрачный программный интерфейс при взаимодействии с C библиотеками. Это значит, что можно писать Nim-код, взаимодействующий с C библиотекой так же, как если бы это была Nim-библиотека, в отличие от, например, Python.
Так же вместе с компилятором языка поставляется пакетный менеджер nimble [32] и генератор документации из комментариев, написанных на reStructuredText [33].

Вывод

Из всего вышесказанного следует, что для ПМ АПНДВ лучше всего подойдет язык Nim благодаря его скорости, выразительности и портируемости на различные платформы. Кроме того, для подготовки динамического анализа программы будут использованы утилиты, умеющие разбирать заголовки исполняемого файла, а именно objdump и readelf. Форматирование входных данных для данных утилит будет осуществляться с помощью Bash-скриптов. Не смотря на то, что данные программы имеются только на UNIX системах, есть возможность использовать их и в операционной системе Windows, через Cygwin [34].

2.1.2 Сравнение сред разработки

Для разработки на Nim существует несколько IDE и огромное количество текстовых редакторов, часть которых рассмотрим ниже:

Рассмотрим подробно каждый из представленных в таблице редакторов.

Aporia

Простая IDE, написанная на Nim, для редактирования исходного кода Nim, с использованием GTK2. В настоящее время не поддерживается, так как большая часть Nim-программистов перешла на Visual Studio Code.

Atom

Графический редактор с открытым исходным кодом от GitHub Inc., написан с использованием Electron [40] – фреймворка для разработки кросс-платформенных приложений с помощью HTML, JavaScript и CSS. Из-за архитектурных и технологических решений, все программы, написанные на данном фреймворке, являются очень требовательными к ресурсам.

Таблица 9 — Сравнительная таблица IDE и редакторов кода

Свойства \ IDE/Редактор	Aporia [35]	Atom [36]	Sublime Text [37]	Visual Studio Code [38]	Vim [39]
Поддержка плагинов	Нет	Да	Да	Да	Да
Требователен к ресурсам	Нет	Да	Нет	Да	Нет
Имеет продвинутую систему редактирования текста	Нет	Нет	Нет	Нет	Да
Кросс-платформенность	Есть	Есть	Есть	Есть	Есть
Может работать без GUI	Нет	Нет	Нет	Нет	Да
Восстановление после сбоев	Нет	Есть	Есть	Есть	Есть
Возможность выделять ключевые слова с помощью регулярных выражений	Нет	Есть	Есть	Есть	Есть
Опыт использования	Нет	Нет	Есть	Есть	Есть

Sublime Text

Проприетарный графический текстовый редактор написан на C++ и python, возможности которого могут быть расширены с помощью плагинов на python.

Visual Studio Code

Графический редактор с открытым исходным кодом от Microsoft. Так же, как и Atom, написан с использованием Electron. Имеет встроенный «магазин» плагинов. На текущий момент является самым популярным редактором кода.

Vim

Текстовый редактор с открытым исходным кодом и большими возможностями к быстрому редактированию текстов. Является наследником редактора vi, который, в свою очередь, создавался с оглядкой на редактор ed. Управление делится на режим ввода и режим команд, благодаря чему есть возможность управлять редактором только с помощью клавиатуры, что, при должном умении, повышает скорость не только из-за отсутствия необходимости в использовании компьютерной мыши, но и более коротким сочетаниям «горячих клавиш». Поддерживает программирование необходимого функционала с помощью языка

vimscript или python. Встроенный функционал позволяет проводить сложное редактирование в автоматическом формате, что имеет свое применение в скриптах. Легко поддается модифицированию с помощью плагинов. Есть под множество платформ.

Так как Vim, не смотря на его расширяемость с помощью плагинов, все равно остается текстовым редактором, то для комфортной разработки требуется дополнить его функционал возможностью компилировать, запускать и отлаживать ПМ АПНДВ. Так как Vim используется мной в качестве редактора из консоли, как компиляция, отлаживание и запуск ПМ АПНДВ, то для удобной работы, будем усовершенствовать именно консоль. Для этого в систему был установлен `tmux` – терминальный мультиплексор, позволяющий открывать в одном терминале несколько окон, и удобно переключаться между ними. Помимо этого, tmux является клиент-серверным приложением, в котором окнами управляет tmux-сервер, а видит их tmux-клиент. Такое разделение функционала позволяет настроить выделенный tmux-сервер, к которому можно будет подключаться удаленно с любого устройства, поддерживающего SSH, и возвращаться, управлять сессиями, которые были открыты с других устройств.

Вывод

Из всего вышесказанного и личного опыта следует, что для разработки ПМ АПНДВ лучше всего подойдет текстовый редактор Vim, так как он поддерживает добавление плагинов, не требователен к ресурсам и позволяет очень быстро редактировать текст. Для расширения его его функциональности использовался терминальный мультиплексор tmux и следующие плагины:

1) NERDTree [41] – улучшает просмотр каталогов;
2) Tabular [42] – позволяет быстро выравнивать текст для улучшения читаемости;
3) vim-polyglot [43] – подсветка синтаксиса большого числа языков;
4) undotree [44] – просмотр истории изменений в виде дерева;
5) rainbow [45] – подсветка вложенных скобок разными цветами, для улучшения читаемости.

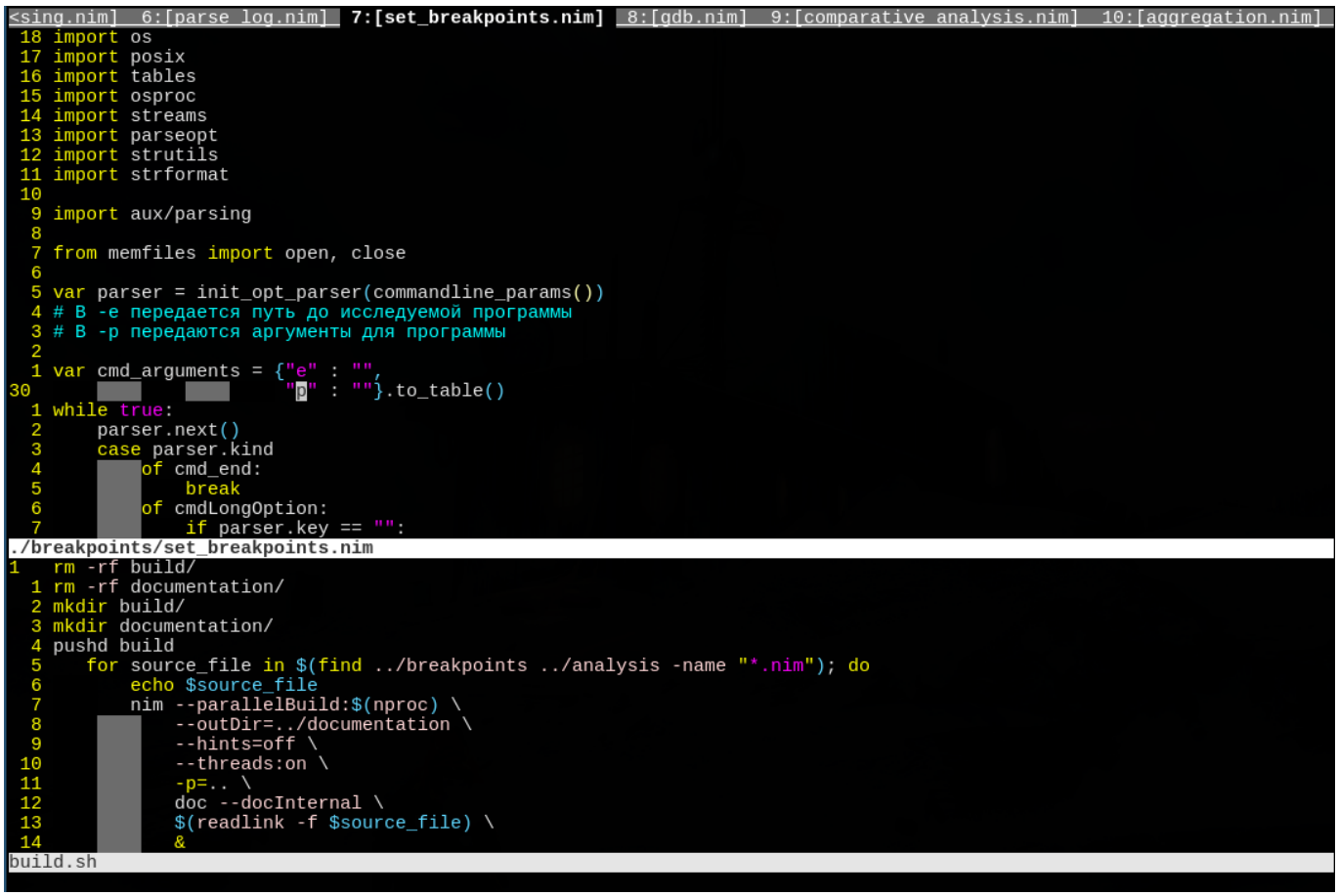


Рисунок 2.1 — Интерфейс Vim ПМ АПНДВ

2.2 Архитектура ПМ АПНДВ

Архитектура программного обеспечения это система, объединяющая внутрение компоненты, их связи между собой и с окружением, а так же принципы, использующиеся при проектировании и эволюции программы [46].

При проектировании ПМ АПНДВ была выбрана UNIX-философия [47], заключающаяся в следующих основополагающих принципах:

– создавать маленькие программы;

– программы делают одно дело, но делают его хорошо;

– хранить данные в текстовом, читаемом для людей формате.

Поэтому было принято решение разрабатывать под каждую подзадачу проведения сертификации ПО самостоятельную программу, которая была бы маленькой и хорошо бы справлялась со своим назначением.

2.2.1 Организация передачи информации между компонентами ПМ АПНДВ

Передача информации между компонентами ПМ АПНДВ осуществляется посредством сериализации внутренних структур (рис. 2.2 и рис. 2.3) конкретного

модуля в формате JSON. JSON удобен тем, что является простым для чтения как человеком, так и компьютером, что позволяет оператору анализировать так же и промежуточные результаты работы, для вынесения вердикта.

Виды сериализуемых данных

В ПМ АПНДВ сериализуются данные после прохождения этапа:

– статического анализа исходных кодов;

– динамического анализа сертифицируемой программы.

Структура данных помогает иерархически организовать доступ к собранной, во время динамического анализа, информации.

Данные с расставленных точек останова, содержатся в структуре `BreakpointInfo`, которая заполняется непосредственно во время выполнения машинных инструкций программы, а значит важно в них получить максимальное количество информации текущем мгновенном состоянии программы. В структуре содержится:

– адреса:
  – `call`-инструкции, на которой находится точка останова;
  – по которому собирается сделать вызов `call`-инструкция;
  – функции, в котором находится данная `call`-инструкция;

  Которые необходимы для последующего сравнительного анализа;

– регистры, в которых могут содержаться передаваемые параметры (fastcall convention [48]);

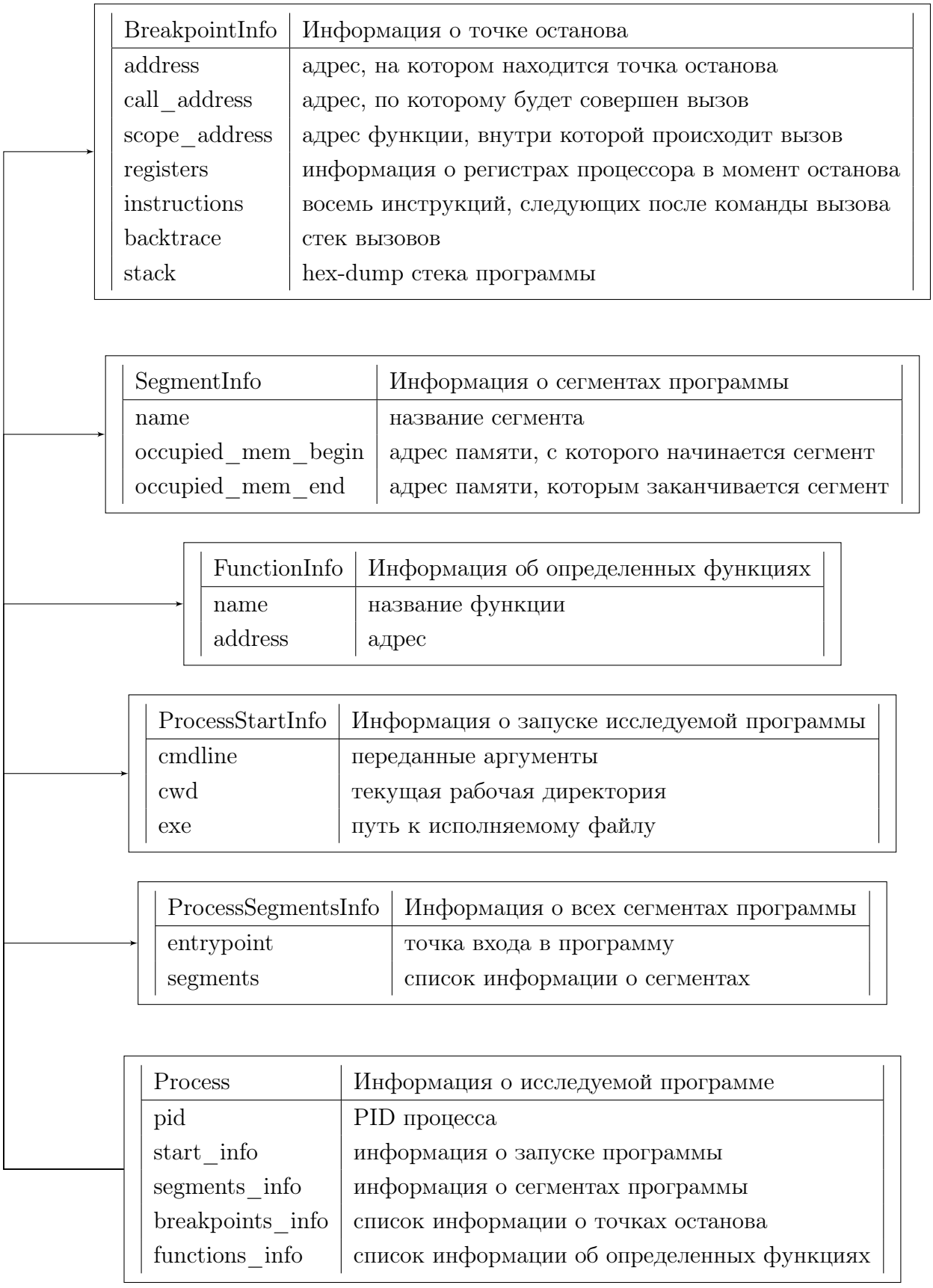
– следующие за `call` 8 инструкций, в которых может содержаться код, обрабатывающий возвращенное значение;

– стек вызовов, позволяет посмотреть ветку исполнения исследуемой программы.

Информация о сегментах в `SegmentInfo` позволяет определить, к какому сегменту относится вызываемая, или текущая функция. Например, это может быть сегмент динамически загружаемой библиотеки.

`FunctionInfo` содержит информацию, которую предоставляет GDB при загрузке программы: список известных функций и их адреса.

`ProcessStartInfo` сохраняет параметры запуска, `ProcessSegmentsInfo` – агрегирует информацию по всем сегментам программы. Структура `Process` же агрегирует в себе всё вышеперечисленное.



BreakpointInfo	Информация о точке останова
address	адрес, на котором находится точка останова
call_address	адрес, по которому будет совершен вызов
scope_address	адрес функции, внутри которой происходит вызов
registers	информация о регистрах процессора в момент останова
instructions	восемь инструкций, следующих после команды вызова
backtrace	стек вызовов
stack	hex-dump стека программы

SegmentInfo	Информация о сегментах программы
name	название сегмента
occupied_mem_begin	адрес памяти, с которого начинается сегмент
occupied_mem_end	адрес памяти, которым заканчивается сегмент

FunctionInfo	Информация об определенных функциях
name	название функции
address	адрес

ProcessStartInfo	Информация о запуске исследуемой программы
cmdline	переданные аргументы
cwd	текущая рабочая директория
exe	путь к исполняемому файлу

ProcessSegmentsInfo	Информация о всех сегментах программы
entrypoint	точка входа в программу
segments	список информации о сегментах

Process	Информация о исследуемой программе
pid	PID процесса
start_info	информация о запуске программы
segments_info	информация о сегментах программы
breakpoints_info	список информации о точках останова
functions_info	список информации об определенных функциях

Рисунок 2.2 — Сохраняемые структуры динамического анализа

UnitInfo	Информация об одном файле исходного кода
arguments	список аргументов компиляции
directory	папка с файлом исходного кода
file	имя файла

BuildInfo	Информация о сборке программы
units_info	список файлов исходного кода

CflowConstruct	Описание функции в статическом анализе
name	имя функции
nesting	уровень вложенности
signature	сигнатура функции
path	путь до файла, в котором используется функция
line	номер строки
recursive	рекурсивность функции
text_offset	отступ в сегменте .text

Рисунок 2.3 — Сохраняемые структуры статического анализа

Структуры данных, относящиеся к статическому анализу косвенно связаны друг с другом. Их можно разделить на структуры времени компиляции программы и структуры времени статического анализа. К структурам времени компиляции относятся:

- `UnitInfo` содержит информацию о сборке одного файла исходного кода; В нее входит:
  - аргументы компилятору – указание заголовочных файлов, параметры генерации машинного кода, указание макросов и т.д.;
  - папка, в котрой находится файл исходного кода;
  - название файла.
- `BuildInfo` агрегирует все `UnitInfo`, полученные при компиляции проекта и записанные в compilation database [49];

К структурам времени анализа относится `CflowConstruct`, которая содержит в себе уже разобранную и типизированную информацию, предоставляемую Cflow – программой статического анализа:

- имя функции;

– уровень вложенности вызова – уровень дерева, на котором располагается конкретная функция, относительно точки входа – функции с нулевым уровнем вложенности;
– сигнатура функции, в данном случае вместе с возвращаемым типом;
– путь до файла, в котором функция была использована;
– номер строки, где функция была использована;
– рекурсивность функции – значение принимающее либо «ложь», либо «истина», в зависимости, есть ли в определении функции вызов самой себя;
– отступ в области .text – количество в байтах от начала .text-сегмента уже скомпилированной программы до начала функции.

Все значения, кроме `text_offset`, заполняются непосредственно во время проведения статического анализа.

`text_offset` заполняется на стадии агрегации результатов линковки и результатов статического анлиза. Это необходимо, чтобы на стадии сравнительного анализа можно было сопоставить адреса вызываемых функций в динамическом и статическом анализе, полагаясь на разность между началом сегмента .text и адресом функции. Как на стадии линковки, так и в динамическом анализе для конкретной фунции он будет одинаков.

2.2.2 Схема данных

Из схемы данных на рис. 2.4 видно, что работу ПМ АПНДВ можно разбить на параллельные задачи.

2.2.3 Алгоритм работы программы

Работу ПМ АПНДВ можно разделить на функциональные этапы:

1) сборка анализируемой программы;
2) статический анализ результатов сборки;
3) динамический анализ собранной программы;
4) сравнительный анализ результатов статического и динамического анализа.

Причем п. 2) и п. 3) могут выполняться одновременно, так как не имеют зависимости по данным.

Рассмотрим подробнее каждый из этапов.

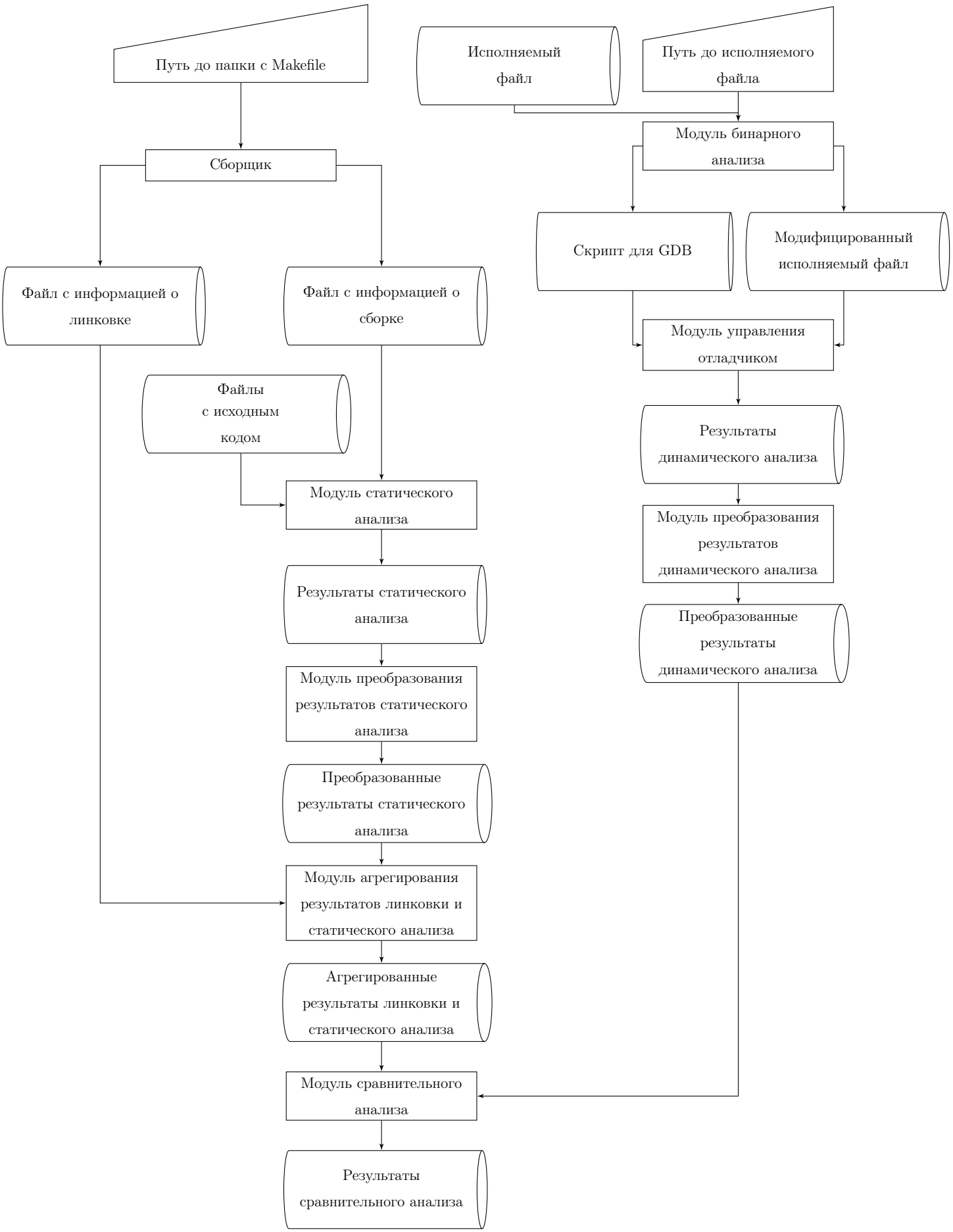


Рисунок 2.4 — Схема данных ПМ АПНДВ

Сборка анализируемой программы

Этап сборки анализируемой программы в ПМ АПНДВ является ключевым для проведения статического, динамического и сравнительного анализов. Не имеет смысла проводить анализ исполняемого файла, скомпилированного не в процессе проведения анализа. Без map-файла нельзя дополнить статический анализ физическими адресами функций в исполняемом файле. Не имея статического и динамического анализа, невозможно провести сравнительный. При сборке программы используется утилита make и BEAR.

Утилита make

Make – утилита для автоматической сборки программ и библиотек из исходного кода. Работает через чтение специальных файлов – «мейкфайлов» (англ. Makefile), в которых описаны «рецепты» сборки. В мейкфайле может находиться любое количество рецептов, они могут быть как зависимы друг от друга, так и быть совершенно непересекающимися.

Отдельный рецепт имеет название, компоненты, от которых он зависит (могут остать пустыми, это будет означать, что рецепт независим) и правила сборки, они тоже могут оставаться пустыми.

Стоит заметить, что использование программы make в UNIX системах не обязательно ограничевается компиляцией программ и библиотек. В мейкфайлах с помощью рецептов так же можно описать различные сценарии, требующие последовательного выполнения команд. В большинстве программ, использующих схему распространения через компиляцию исходного кода, имеются мейкфайлы, в которых определены рецепты clean – очистить и help – помощь. Которые реализуют, соответственно, очистку директорий проекта от временных файлов, полученных в результате выполнения других рецептов мейкфайла и получения информации о доступных рецептах.

По-умолчанию, make выполняет рецепты один за другим, не начиная выполнение нового рецепта, пока не закончится старый. Но при указании определенного аргумента, make может выполнять несвязанные рецепты параллельно, что значительно ускоряет процесс сборки.

Утилита BEAR

Build EAR [50], или сокращенно BEAR позволяет генерировать compilation database, указывая ей команду сборки. Compilation database или CompileDB содержит информацию о том, с какими параметрами компилировались отдельные файлы проекта.

Сборка анализируемой программы происходит посредством программы-обертки, повторяющей интерфейс программы make и запускающая её в контексте программы BEAR, для генерации compilation database. Помимо этого, для make указывается генерация map-файла, файла содержащего информацию о сегментах программы, относительных отступах функций внутри сегментов и др. После окончания компиляции дополнительно происходит разбор сегмента .text map-файла на предмет функций и их относительных адресов внутри сегмента. Полученные данные сохраняются на диск в JSON формате.

Статический анализ результатов сборки

Статический анализ результатов сборки производится с помощью программы Cflow, которой на вход подаются аргументы компиляции, взятые из compilation database, полученной на предыдущем шаге, а так же сами файлы с исходными кодами.

Отчет Cflow состоит из списка функций, определяемых следующим правилом, описанным в 2.1, где описания полей обрамлены косыми чертами:

Листинг 2.1 Формат записи в отчете Cflow

```
{/уровень вложенности/} /имя функции/() </сигнатура функции вместе
    с возвращаемым значением/ at /абсолютный путь до файла/:/номер
    строки в файле/>:
{/уровень вложенности вызываемой функции/} /имя вызываемой функции
   /() </сигнатура вызываемой функции вместе с возвращаемым значен
   ием/ at /абсолютный путь до файла/:/номер строки в файле/>:

    ...

```

Данный формат файла легко поддается разбору с помощью регулярных выражений. В ПМ АПНДВ использовалась библиотека регулярных выражений PCRE [51]. Не смотря на то, что Cflow умеет генерировать отчет, в которых представлен не граф вызываемых функций, а список функций, вызывавших данную,

этот формат, не смотря на удобство, страдает большим количеством повторений, что в свою очередь вызывает слишком большой объем отчета и замедляет его разбор, из-за чего в ПМ АПНДВ решено было использовать стандартную версию отчета.

Листинг 2.2 Пример генерации отчета Cflow

```
{   0} printsel() <void printsel (const arg *arg) at /st/st.c:1988>:
{   1}  tdumpsel() <void tdumpsel (void) at /st/st.c:1994>:
{   2}    getsel() <char *getsel (void) at /st/st.c:590>:
{   3}      xmalloc() <void *xmalloc (size_t len) at /st/st.c:253>:
{   4}        malloc()
{   4}        die() <void die (const char *errstr, ...) at /st/st.c:654>:
```

Динамический анализ собранной программы

Подготовка к динамическому анализу собранной программы начинается сразу после завершения этапа сборки разд. 2.2.3. Путь до исполняемого файла передается в модуль расстановки точек останова для первичного модифицирования. Модифицирование заключается в том, что с помощью программ objdump и readelf, о которых говорилось в разд. 2.1.2 и небольших скриптов, написанных на bash, происходит следующее:

1) находятся все `call`-инструкции, сохраняя их относительные адреса от начала сегмента .text;
2) узнается отступ сегмента .text в байтах от начала файла;
3) сохраняется байт по адресу, полученным на предыдущем шаге;
4) заменяется байт по адресу, полученным на предыдущем шаге, на 0xCC в шестнадцатиричной системе счисления. Это машинный код инструкции `int 3` – программного прерывания, которое используется в отладчиках для установки точек останова;
5) генерируется скрипт для отладчика GDB, по расстановке точек останова на все `call`-инструкции, восстановлению изменений в файле и снятию состояний программы.

Процесс исполнения данного скрипта:

1) `file абсолютный-путь-до-файла` – загружается исполняемый файл по абсолютному пути;
2) выставляется формат выводимых данных:

1) `set disassembly-flavor intel` – выставляется отображение синтаксиса ассемблерных мнемоник в стиль intel;
2) `set input-radix 10` – выставляется десятичная система для ввода;
3) `set args аргументы-программе` – анализируемой программе передаются аргументы;
4)
```
define xxd
    dump binary memory dump.bin $arg0 $arg0+$arg1
    shell xxd dump.bin >> gdb.log
end
```
– определеяется команда `xxd`, которая будет добавлять в лог динамического анализа дамп заданного места памяти;

3) `run` – запускается исследуемая программа;
4)
```
info proc
info files
info functions
```
– выводится информация о процессе, сегментах и обнаруженных функциях;
5) программа останавливается на первом байте сегмента .text, 0xCC, кодирующем программную точку останова;
6) `set $pc--` – счетчик команд уменьшается на единицу;
7) `set *(char*)$pc=байт` – по адресу, указанном в счетчике команд записывается ранее сохраненный первый байт сегмента .text;
8) расставляются относительные точки останова;
9) программа выходит из останова и продолжает работу, собирая информацию с точек останова.

Информация с прошедших точек останова собирается с помощью следующих команд GDB:

```
commands
    info registers
    x/8i $pc
    bt
    xxd $sp-256 256
```

```
    continue
end
```

Нужно отметить, что в отладчике GDB существует команда `starti`, которая запускает программу и останавливается на первой инструкции, что позволяет отлаживать программу прямо с точки входа. Но проблема использования `starti` состоит в том, что первой инструкцией программы может оказаться не .text-сегмент, а какой-нибудь другой, а значит относительная расстановка точек будет неверной. Поэтому приходится на уровне исполняемого файла удостоверяться, что исполнение программы прервется именно на первой инструкции .text-сегмента.

Сравнительный анализ результатов статического и динамического анализа

Модуль сравнительного анализа запускается после того, как становятся готовы результаты статического и динамического анализа. Он загружает результаты с диска в описанные ранее структуры разд. 2.2.1, а так же информацию о функциях из map-файла. Это позволяет дать отчет по нескольким вариантам несовпадения:

- несовпадение функций в map-файле и функций, объявленных в статическом анализе (каких имен из множества функций, полученных из map-файла нет среди функций, определенных в исходникых текстах);
- несовпадение распознанных отладчиком GDB функций и функций, полученных в динамическом анализе (каких имен из множества функций, определенных GDB нет среди функций, полученных из map-файла);
- несовпадение функций в статическом и динамическом анализе.

2.2.4 Разработка консольного интерфейса ПМ АПНДВ

Консольный интерфейс программы, или программа, поддерживающая интерфейс командной строки – компьютерная программа, обрабатывающая аргументы, переданные ей в определенном формате. Консольный интерфейс не может существовать без командного интерпретатора – другой компьютерной программы, которая обрабатывает команды компьютеру, заданные в виде текста. Один из самых старых видов взаимодействия человека и компьютера. Появившись в середине 1960-х, он используется и по сей день.

Для запуска ПМ АПНДВ с консольным интерфейсом нужно перейти в папку с собранным ПМ АПНДВ, после чего использовать bash-скрипт 2.3, который принимает следующие аргументы:

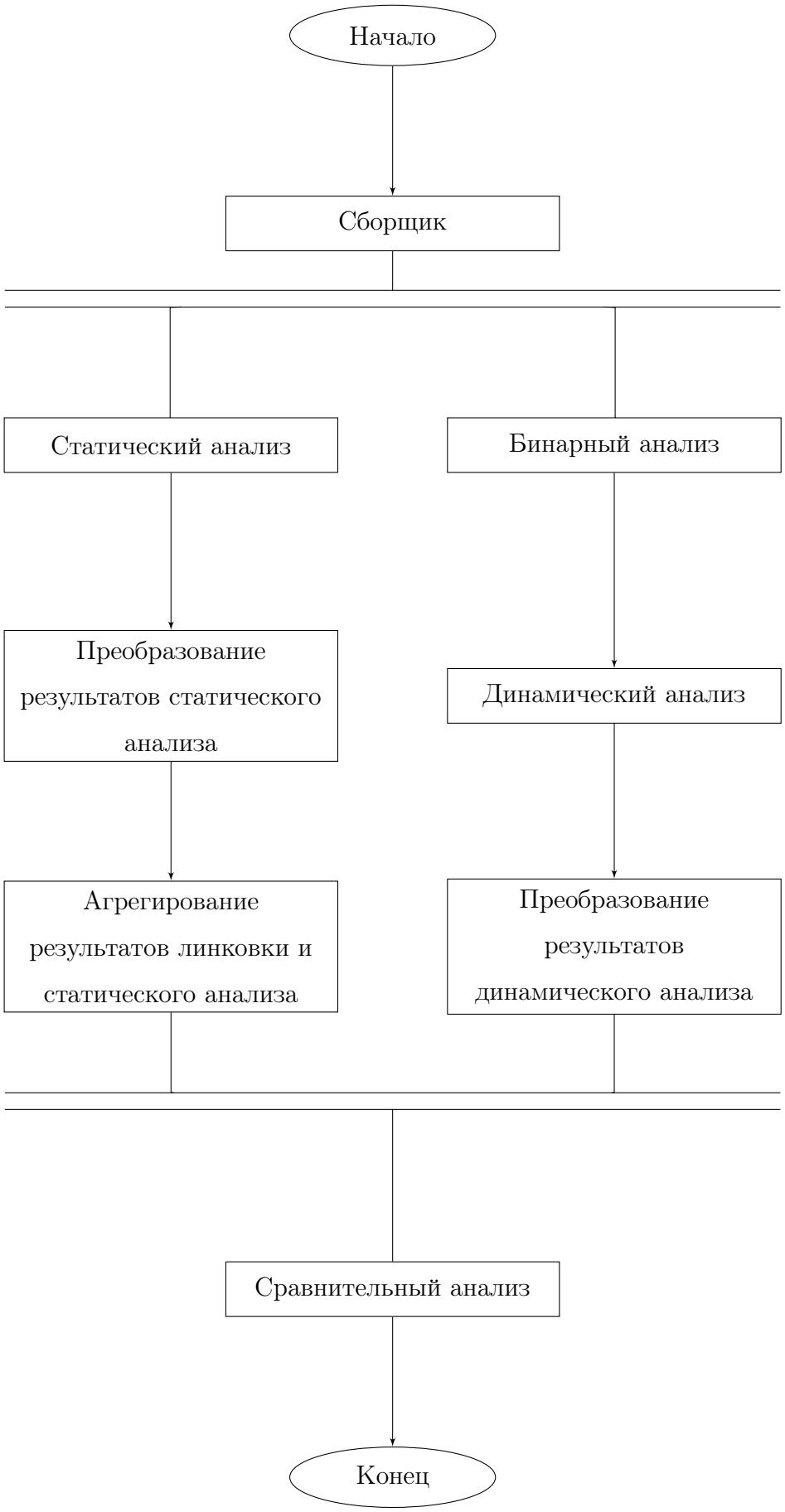


Рисунок 2.5 — Алгоритм работы ПМ АПНДВ

1) `$1` – путь до папки, в которой хранится мейкфайл проекта;
2) `$2` – путь до исследуемого исполняемого файла.

Результаты сравнительного анализа выводятся на экран.

Листинг 2.3 run.sh

```
pushd $1
    make clean
popd
pushd build
    ./build -C=$1
    (./set_breakpoints -e=$2 &&
     ./gdb ;
     ./parse_log &&
     ./dynamic_analysis;) &
    (./static_analysis &&
     ./aggregation) &
     wait $(jobs -p)
    ./comparative_analysis
popd
```

Если же ПМ АПНДВ еще не собран, то нужно воспользоваться скриптом 2.4:

Листинг 2.4 build.sh

```
rm -rf build
mkdir build
pushd build
   for source_file in $(find ../breakpoints ../analysis -name "*.nim"); do
       echo $source_file
       nim --parallelBuild:$(nproc) \
           --outDir=. \
           -p=.. \
           --threads:on \
           c $(readlink -f $source_file) &
   done
   wait $(jobs -p)
popd
```

2.2.5 Разработка графического интерфейса ПМ АПНДВ

GUI, или графический пользовательский интерфейс впервые был показан широкой публике еще в 1968 году, на презентации, впоследствии названной «The Mother of All Demos» (или «Мать всех демонстраций»)[52]. На ней Дуглас Энгельбарт взаимодействовал с текстовым документом посредством устройства, позднее названного компьютерной мышью, участвовал в видео конференции и совершал другие манипуляции, предвосхитив технологии на несколько десятков лет вперед. Первым реально использовавшейся системой с графическим интерфейсом был компьютер Xerox Alto, разработанный в 1973 году исследователями из центра исследований Xerox в Пало-Альто.

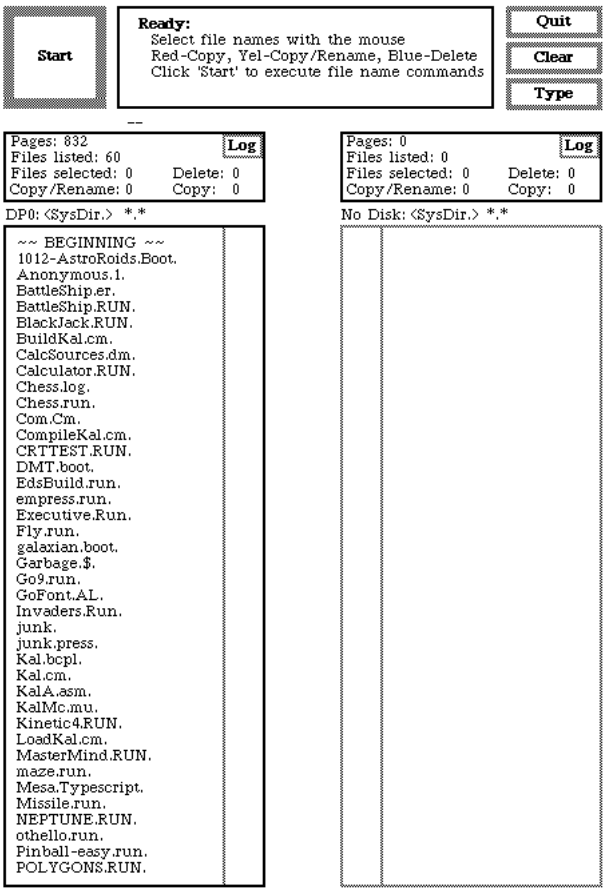


Рисунок 2.6 — Интерфейс файлового менеджера Xerox Alto

И сейчас, не смотря на удобстово консольного интерфейса как для программирования, так и для использования, например, в «Fire-and-forget» задач, коей и старается сделать процесс сертификации ПМ АПНДВ, графический интерфейс остается важным элементом для обучения работы оператора с любым программным обеспечением. Поэтому, для ПМ АПНДВ был создан графический интерфейс с помощью библиотеки Gooey [53].

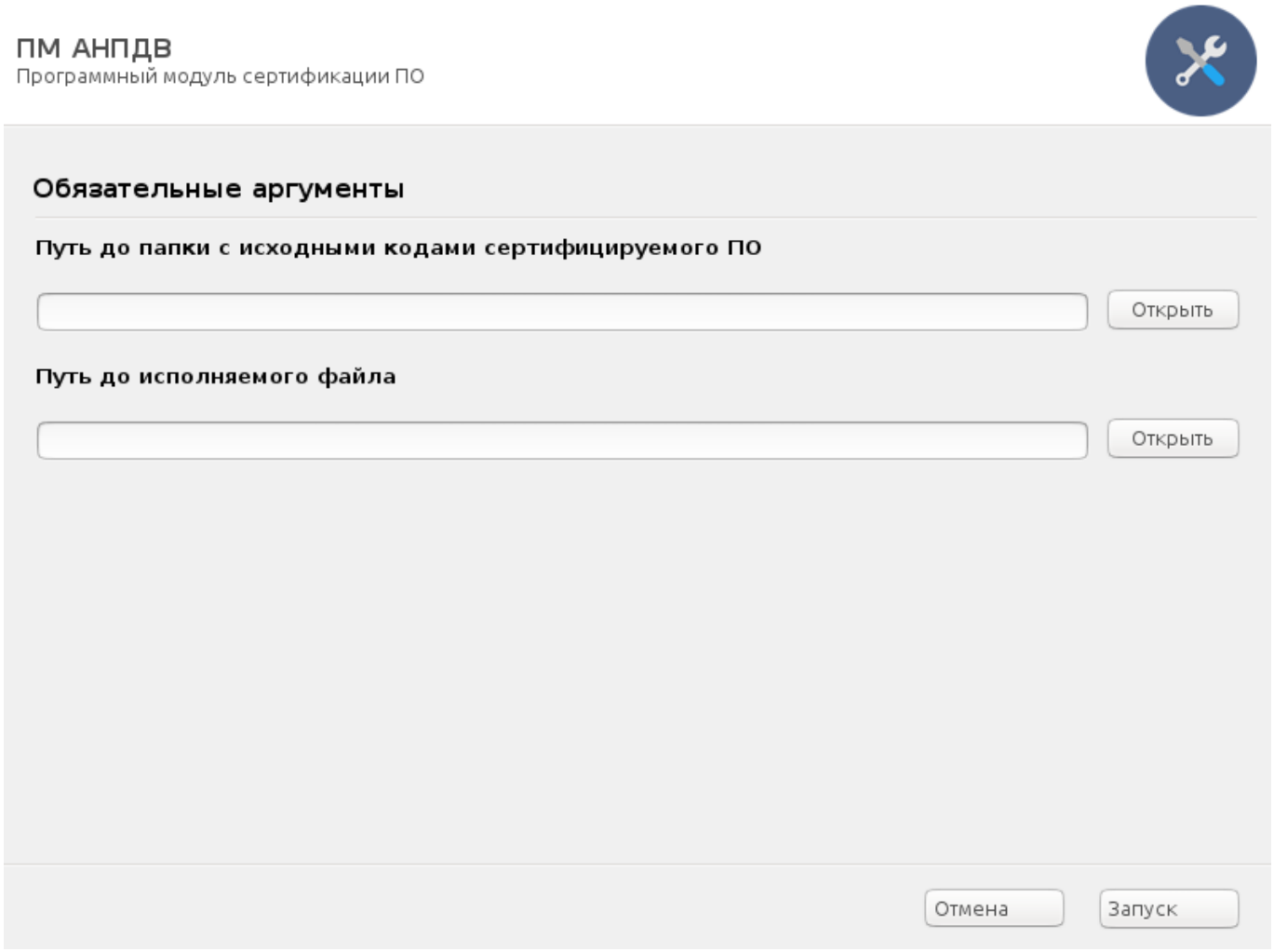


Рисунок 2.7 — Графический интерфейс ПМ АПНДВ

Gooey позволяет преобразовывать строку аргументов python-скрипта, созданную с помощью модуля `argparse` стандартной библиотеки в графический интерфейс, что позволяет «бесплатно» добавить GUI в уже имеющуюся кодовую базу. Помимо стандартного `argparse`, Gooey предоставляет собственный парсер аргументов командной строки, который расширяет графические возможности приложения, позволяя использовать специфичные поля, вроде выбора даты или ввода пароля. Так как в python удобно создавать кросс-платформенные приложения, то данная библиотека не исключение – внутри нее используется обертка wxPython над библиотекой wxWidgets, что позволяет программам, использующим Gooey быть написанными всего лишь раз и выглядеть как родные приложения той или иной платформы.

2.3 Выводы по разделу

В конструкторском разделе было проведено сравнение и обоснование выбора языка программирования и среды разработки для ПМ АПНДВ. Разработана архитектура ПМ АПНДВ. Также были описаны:

1) алгоритм передачи данных между модулями ПМ АПНДВ;
2) формат данных, передающихся между модулями ПМ АПНДВ;
3) используемые сторонние программы и форматы данных, обрабатываемые ими.

Составлена схема данных, алгоритм работы ПМ АПНДВ. Описаны командный и графический интерфейс ПМ АПНДВ. Подробно рассмотрены шаги выполнения процесса сертификации с помощью ПМ АПНДВ

Раздел 3. Технологический раздел

3.1 Процесс разработки ПМ АПНДВ

Разработка ПМ АПНДВ происходила на языке программирования Nim, с использованием системы контроля версий git [54].

Система контроля версий – это система, сохраняющая изменения в файлае или наборе файлов в течение времени и позволяющая возвращаться к их определенным версиям. Это позволяет вернуть файлы в состояние, в котором они были до внесения изменений, возвратить проект к исходному состоянию, защитить себя и проект от безвозвратной потери работающей версии программы вследствие ломающих изменений, удаления или другой утери файлов. Помимо этого системы контроля версий значительно облегчают параллельную разработку программ, позволяя нескольким разработчикам одновременно работать в разных «ветках» – это специальные указатели на конкретное изменение файлов в системе контроля версий, которые позволяют добавлять изменения не затрагивая иерархию изменений других веток. Все это возможно, так как система контроля версий хранит не сами файлы, а лишь изменения в виде набора «заплаток» (патчей, от анг. patch) к ним. Заплатки это небольшие файлы содержащие только информацию о изменениях, произошедших между стадиями фиксации изменений – «коммитами».

Пример заплатки:

Листинг 3.1 git diff

```
diff --git a/analysis/static/aggregation.nim b/analysis/static/aggregation.nim
index 1623a36..13ac345 100644
--- a/analysis/static/aggregation.nim
+++ b/analysis/static/aggregation.nim
@@ -1,4 +1,4 @@
-## Этот модуль занимается агреггированием результатов
+## Этот модуль занимается агрегированием результатов
 ## линковки и статического анализа:
 ## Всем CflowConstruct выставляется ‘‘text_offset‘‘ -- адрес функции в бинарнике
 ##
```

В качестве парадигмы программирования, где было возможно это сделать без ущерба производительности ПМ АПНДВ, использовалось функциональное программирование.

Суть функционального программирования заключается в том, что любую программу можно описать как вычисление, в математическом понимании, значений некоторых функций и их суперпозиций. В функциональном программировании функции являются объектами первого класса – это значит, что функции могут быть:

– переданы в качестве аргумента другой функции;
– возвращены из функции как результат;
– созданы во время исполнения программы;

В императивном программировании программный код описывает то, как должна выполниться задача. Это происходит через изменение состояния программы, а функции, во время вычисления результатов могут основываться на внешних, относительно функции, переменных и иметь побочные эффекты, например изменяя состояние внешних переменных. Из этого следует, что вызов императивной процедуры с одинаковыми параметрами, может возвращать отличающиеся данные.

Функциональное программирование, в свою очередь, обходится вычислением результатов функций от некоторых исходных данных без изменения внешнего состояния.

«Функциональное программирование» иногда служит синонимом к «чистому функциональному программированию», подмножеству функционального программирования, в котором все функции являются детерминированными математическими функциями, или «чистыми функциями». Чистые функции всегда возвращают одинаковый результат при одинаковых входных данных и не зависят от внешнего изменяемого состояния или других побочных эффектов. Написание чистых функций позволяет снизить колчество ошибок в программе, а сами программы становятся легкотестируемыми и легкоотлаживаемыми, легче поддаются распараллеливанию из-за отсутствия зависимости по данным.

То, как написаны функциональные программы, позволяет компиляторам и интерпретаторам использовать запоминание результатов функции при некоторых аргументах и возврат запомненных результатов, без повторного их вычисления.

3.1.1 Сборка программы на языке Nim

Язык Nim, как уже говорилось ранее в разд. 2.1.1, для преобразования исходных кодов использует source-to-source (S2S) компиляцию, а в качестве языка

компиляции используется язык Си, либо JavaScript. Для сборки ПМ АПНДВ компилятору задаются такие параметры, что исходные коды компилируются в Си, ради высокой скорости исполнения программы. Компилятор Nim, помимо всего прочего, имеет возможность генерировать документацию, в формате `.html` и `.json` из файлов с исходным кодом. Для генерации документации используются комментарии в формате reStructuredText. Для привязки комментария к функции, типу или определению класса, комментарий должен быть написан сразу после объявления функции, типа или определения класса.

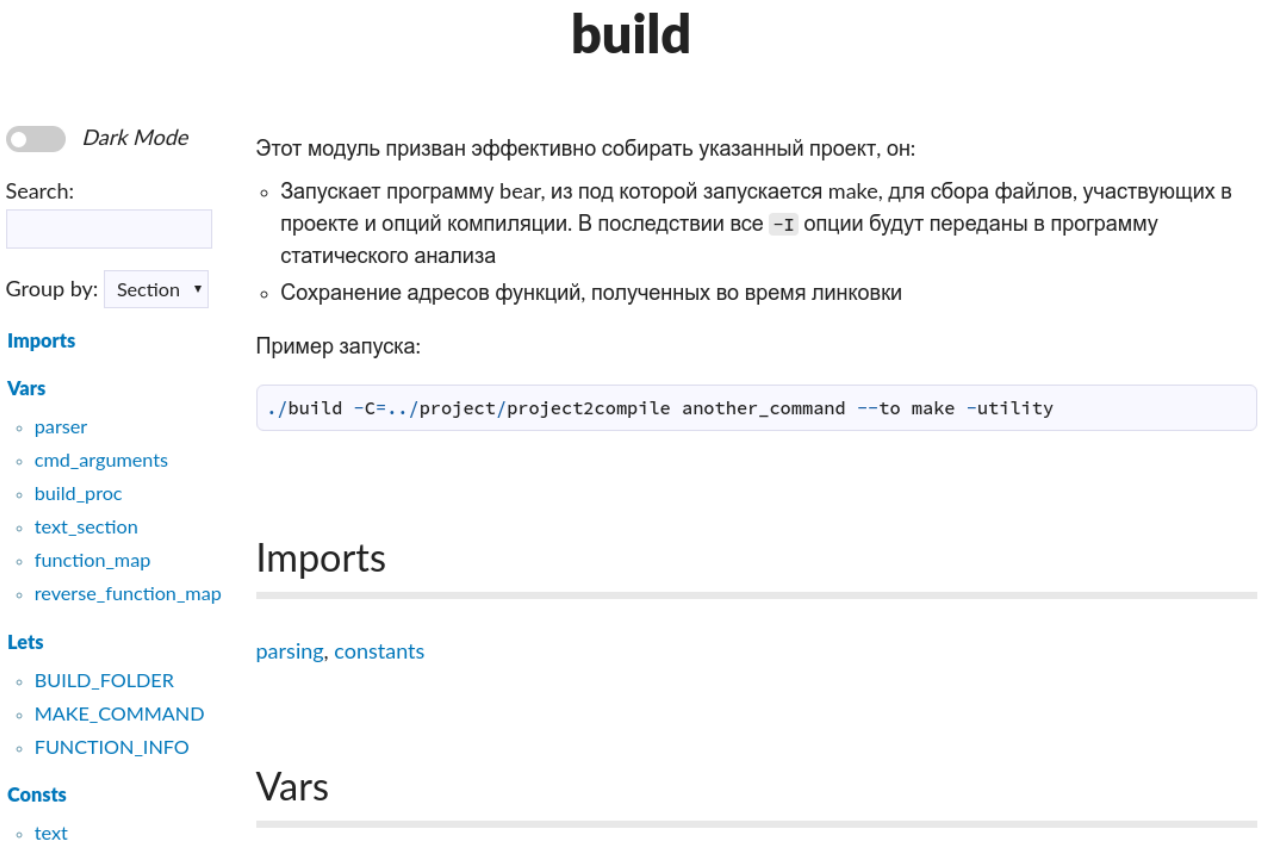


Рисунок 3.1 — Сгенерированная документация по одному из модулей ПМ АПНДВ

В ПМ АПНДВ, в качестве системы сборки программы используется самописный скрипт на Bash, листинг 2.4, выполняющий следующие действия:

1) удаляется и создается заново папка `build`, содержащая скомпилированные модули;
2) текущая папка меняется на `build`;
3) ищутся все файлы с расширением `.nim`, после чего для каждого файла запускается процесс компиляции.

Рассмотрим процесс компиляции. Компилятор запускается со следующими параметрами:

- `--parallelBuild:$(nproc)` – говорит компилятору использовать параллельную компиляцию, с количеством параллельных процессов равному количеству процессоров в компьютере;
- `--outDir=.` – файлы, полученные в процессе компиляции, будут сохранены в текущую папку;
- `-p=..` – указывает родительский каталог;
- `--threads:on` – говорит компилятору разрешить компилируемой программе использовать механизм потоков, компилятор дополнительно проверяет потокобезопасность программы;
- `c $(readlink -f $source_file)` – говорит компилятору, какой файл компилировать, а команда `readlink -f` предоставляет компилятору абсолютный путь до файла в системе.

3.1.2 Тестирование ПМ АПНДВ

Тестирование програмного обеспечение – это процесс проверки соответствия поведения ПО и ожидаемых результатов на некотором множестве тестов. Может проводиться как вручную, так и с помощью специализированных программ, призванных автоматизировать процесс. Тестирование может различаться как по типам, так и по методам тестирования. В ПМ АПНДВ применялись следующие типы тестирования:

1) юнит-тестирование – тестирование модулей ПМ АПНДВ и их частей;
2) интеграционное тестирование – тестирование взаимодействия модулей ПМ АПНДВ между собой;
3) системное тестирование – тестирование ПМ АПНДВ как системы в целом.

Методов тестирования три:

1) тестирование методом белого ящика – ПО тестируется с учетом работы внутренних механизмов программы;
2) тестирование методом черного ящика – ПО тестируется без учета работы внутренних механизмов программы;
3) тестирование методом серого ящика – ПО тестируется с неполным знанием работы внутренних механизмов программы.

Таблица 11 — Сравнение методов тестирования для разработки ПМ АПНДВ

Метод тестирования	Потребность в использовании
Метод белого ящика	Нет потребности, так как ПМ АПНДВ разрабатывался с применением TDD и ориентацией на интерфейс между модулями
Метод черного ящика	Есть потребность, для тестирования корректности работы каждого модуля ПМ АПНДВ
Метод серого ящика	Есть потребность, для тестирования корректности работы связи между модулями ПМ АПНДВ

ПМ АПНДВ разрабатывался с использованием техники test-driven development [55] – разработки через тестирование, или TDD. Данный подход особенно удобен при написания программ в функциональном стиле.

Разработка через тестирование – это подход к разработке программного обеспечения, основывающийся на очень коротком цикле разработки: требования к ПО становятся специальными вариантами тестирования, а код улучшается до того момента, пока тест не будет проходить успешно.

Плюсы данного принципа разработки состоят в следующем:

1) подход помогает разработчикам быть уверенным в коде, который они пишут;
2) правильное написание тестов позволяет реже использовать отладку для поиска ошибок в ПО;
3) при фокусировании на создании тестовых сценариев, разработчик в первую очередь представляет функциональность с позиции клиентов. А значит он будет ставить в разработку интерфейса над разработкой функциональности, что является еще одним кирпичиком в хорошем дизайне программы.

Такой принцип разработки позволяет не только

Цикл разработки ПО с помощью методологии TDD состоит из следующих шагов:

1) Добавить тест.

   В TDD каждая новая функциональнаость должна начинаться с написания теста для нее. Для написания теста разработчик должен хорошо понимать специфику добавляемой функциональности и требования, на-

кладываемые на нее. Это помогает разработчику фокусироваться на важных вещах при разработке функционала.

2) Запустить все тесты и проверить, что новый тест завершился с ошибкой.
   Это подтверждает, что все тесты работают корректно, а так же показывает, что новый тест не срабатывает без написания нового кода, в случае, если требуемая функциональность уже имеется и исключает возможность некачественного теста. Так же этот шаг увеличивает уверенность разработчика в качестве теста.
3) Написать код.
   На этом шаге требуется написать код для вводимой функциональности, который заставит тест завершиться успехом. Не обязательно стараться написать код, который будет хорошо работать, качество кода будет повышено на следующих шагах.
4) Запустить все тесты.
   Если все тестовые сценарии проходят, то разработчик становится уверенным, что код удовлетворяет тестовым требованиям и не ломает текущий функционал ПО. Если это не так, то новый код дорабатывается, пока не будут проходить все тесты.
5) Рефакторинг.
   Увеличивающаяся кодовая база должна регулярно подчищаться при использовании TDD. Новый код может быть перемещен из мест, где он требовался для срабатывания тестов, в место, где он будет более органичен. Постоянно перезапуская тесты во время рефакторинга, разработчик может быть уверен, что своими действиями не повлиял на существующую функциональность
6) Повторение.
   Каждый новый тест продвигает функционал не меньше, чем написанный код. Шаг, с которым редактируется код и запускается тест всегда должен быть маленьким, от 1 до 10 редактирований между запусками. Если новый код никак не может удовлетворить требованиям теста, или другие тесты перестали проходить, то вместо отладки TDD советует откатить внесенные изменения и начать работу заново.

Так как все общение между модулями происходит посредством файлов, то тестированию подвергался данный интерфейс. Тестирование отдельных методов внутри модулей, а так же корректность разбора аргументов модулей, в которых

имеется данный интерфейс, производились, но не будут упомянуты в таблице из-за их примитивности и массовости.

При разработке ПМ АПНДВ, для написания юнит-тестов использовался модуль стандартной библиотеки – `unittest`. Данный модуль позволяет удобно описывать действия, которые должны быть сделаны до, во время и после выполнения теста.

Сначала, с помощью шаблона `suite` дается название набору тестов и обозначается начало области описания набора: «`suite "описание набора тестов":`». Шаблоны `setup` и `teardown` дают начало описанию подготовки окружения перед началом тестирования и изменению окружения после завершения тестирования. Описание каждого теста начинается с шаблона `test`: «`test "описание теста":`», после чего следует набор выражений языка Nim, являющихся телом теста. Запускаются тесты следующей командой компилятору: `nim c -r test "имя теста"`, причем имен тестов может быть сколько угодно.

3.1.3 Профилирование ПМ АПНДВ

Профилирование – вид динамического анализа программы, задачей которого является измерение характеристик программы во время её выполнения. Зачастую профилирование служит инструментом для выявления мест оптимизации программ. Профилировщики могут измерять количество выделяемой программе памяти, частоту и время выполнения отдельных функций и др.

При разработке ПМ АПНДВ, для профилирования использовался модуль стандартной библиотеки – `nimprof`. Другое название nimprof – Embedded Stack Trace Profiler или «Встроенный профилировщик стека вызовов», которое раскрывает его суть – профилировщик анализирует стек вызовов и время, затраченное на выполнение каждой конкретной функции. Для включения профилировщика, требуется скомпилировать модуль, который будет профилироваться, с флагами `--profiler:on` и `--stacktrace:on`

Профилировщик работает следующим образом: во время работы основной программы создаются снимки состояния каждой функции, а стек вызовов показывает, каким путем дошла прогамма до вызова конкретной функции. После окончания работы профилируемой программы генерируется файл `profile_results.txt`

Рассмотрим формат отчета 3.2 профилировщика:

Таблица 13 — Сценарии модульного тестирования ПМ АПНДВ

Цель	Ожидаемый результат	Тест
Проверить корректность запуска сборки и сообщение об ошибках сборки	При удачной сборке генерируется compilation db, лог сборки, прямой и обратный map-файл. Код возврата модуля сборки равен 0. При неудачной сборке генерируется только лог сборки, содержащий ошибки. Код возврата модуля сборки равен 1.	Тест запускает сборку трех различных проектов: не содержащего ошибки, содержащего ошибки, пустого проекта. Проверяется наличие файлов и код возврата модуля.
Проверить корректность парсера логов динамического анализа	При удачном разборе генерируется JSON-файл, описывающий процесс. Код возврата модуля сборки равен 0. При неудачной сборке ничего не генерируется. Код возврата модуля сборки равен 1.	Тест запускает парсер на трех вариантах лога: не содержащего ошибки, с нарушеним формата JSON, пустого файла, а так же без лога. Проверяется наличие файла, описывающего процесс и код возврата модуля.
Проверить корректность парсера логов динамического анализа	При удачном разборе генерируется JSON-файл, описывающий процесс. Код возврата модуля сборки равен 0. При неудачной сборке ничего не генерируется. Код возврата модуля сборки равен 1.	Тест запускает парсер на трех вариантах лога: не содержащего ошибки, с нарушеним формата JSON, пустого файла, а так же без лога. Проверяется наличие файла, описывающего процесс и код возврата модуля.
Проверить корректность динамического анализатора	При корректной работе генерируется JSON-файл с описанием процесса, в котором все точки останова стоят на call-инструкциях При некорректной работе хотя бы одна точка останова будет стоять не на call-инструкции.	Тест запускает парсер динамического лога и проверяет расположение точек останова.

Листинг 3.2 Часть лога профилировщика одного из модулей

```
total executions of each stack trace:
Entry: 1/157 Calls: 20/249 = 8.03% [sum: 20; 20/249 = 8.03%]
  /toolchains/nim-1.2.0/lib/system/iterators_1.nim: readLine 29/249 = 11.65%
  /toolchains/nim-1.2.0/lib/pure/streams.nim: fsReadLine 21/249 = 8.43%
  /toolchains/nim-1.2.0/lib/pure/streams.nim: readLine 29/249 = 11.65%
  project/breakpoints/parse_log.nim: parse_log 248/249 = 99.60%
Entry: 2/157 Calls: 7/249 = 2.81% [sum: 27; 27/249 = 10.84%]
  /toolchains/nim-1.2.0/lib/system/iterators.nim: escapeJsonUnquoted 21/249 = 8.43%
  /toolchains/nim-1.2.0/lib/pure/json.nim: escapeJson 62/249 = 24.90%
  /toolchains/nim-1.2.0/lib/pure/json.nim: escapeJson 62/249 = 24.90%
  /toolchains/nim-1.2.0/lib/pure/marshal.nim: storeAny 146/249 = 58.63%
  /toolchains/nim-1.2.0/lib/pure/marshal.nim: storeAny 146/249 = 58.63%
  /toolchains/nim-1.2.0/lib/pure/marshal.nim: storeAny 146/249 = 58.63%
  /toolchains/nim-1.2.0/lib/pure/marshal.nim: storeAny 146/249 = 58.63%
  /toolchains/nim-1.2.0/lib/pure/marshal.nim: storeAny 146/249 = 58.63%
  /toolchains/nim-1.2.0/lib/pure/marshal.nim: $$ 146/249 = 58.63%
  project/breakpoints/parse_log.nim: parse_log 248/249 = 99.60%
  ...
Entry: 42/157 Calls: 2/249 = 0.80% [sum: 134; 134/249 = 53.82%]
  /toolchains/nim-1.2.0/lib/system/arithmetics.nim: +% 21/249 = 8.43%
  /toolchains/nim-1.2.0/lib/system/assign.nim: genericAssignAux 25/249 = 10.04%
  /toolchains/nim-1.2.0/lib/system/assign.nim: genericAssign 27/249 = 10.84%
  /toolchains/nim-1.2.0/lib/system/assign.nim: genericSeqAssign 21/249 = 8.43%
  /toolchains/nim-1.2.0/lib/pure/options.nim: some 8/249 = 3.21%
  /toolchains/nim-1.2.0/lib/impure/nre.nim: matchImpl 15/249 = 6.02%
  /toolchains/nim-1.2.0/lib/impure/nre.nim: find 13/249 = 5.22%
  project/breakpoints/parse_log.nim: parse_log 248/249 = 99.60%
  ...
```

Записи в отчете начинаются со слова `Entry`, сразу после него идет номер записи, количество сделанных вызовов записи в количественном и процентном соотношении относительно всех вызовов. Внутри квадратных скобок видно, сколько вызовов и какое покрытие в количественном и процентном соотношении существует на момент вызова данной функции. После описания конкретной записи идет описание стека вызовов со статистикой использования функций:

- `/toolchains/nim-1.2.0/lib/pure/streams.nim:` – путь до файла, в котором описана функция из стека вызовов;
- `fsReadLine` – имя функции;
- `21/249` – количество вызовов функции относительно всех вызовов;

– `= 8.43%` – процентное соотношение вызовов функции ко всем вызовам.

Представленная часть лога 3.2 относится к модулю разбора результатов динамического анализа и по ней видно, что больше всего времени было затрачено на вызов функции `readLine` – 8% всего времени исполнения модуля, которая отвечает за чтение строки. Это логично и объясняется тем, что хотя необработанный файл динамического анализа мал – в нем содержится всего 25621 строка, операция чтения будет самой частоиспользуемой из-за назначения профилируемого модуля, а загрузка данных с диска сама по себе является затратной операцией, не смотря на поддержание библиотекой streams некоего внутреннего буфера для оптимизации времени получения данных.

3.1.4 Отладка ПМ АПНДВ

Параметры компилятора, разобранные ранее, позволяют создавать релизную сборку ПМ АПНДВ. Для сборки программы с отладочными символами требуется указать параметр `--debugger:on`.

Как и тестирование, отладка может проводиться с помощью различных методов:

– интерактивная отладка;
– отладочная печать;
– post-mortem отладка или отладка «после смерти»;
– отладка методом «волчья ограда»;
– отладка методом записи и воспроизведения.

Интерактивная отладка

Метод отладки предусматривает запуск отлаживаемой программы в контексте отладчика, расставление точек останова во время выполнения программы, просмотр любых переменных окружения, состояние стека вызовов, изменение переменных для тестирования поведения отлаживаемой программы. Интерактивные отладчики очень распространены, зачастую тем или иным образом интегрированы в IDE. Для компилируемых программ требуется наличие отладочных символов – специальной информации, генерируемой компилятором при преобразовании исходного кода в машинный. В ней содержится информация о файле с исходным кодом и позволяет интерактивным отладчикам сопоставлять блоки машинных инструкций с выражениями языка исходных кодов. Могут быть как интегрированы в исполняемый файл, так и сохраняться отдельно ввиду

серьезного раздувания размера исполняемых файлов для больших программ. Одним из первых интерактивных отладчиков был DDT или DEC Debugging Tape, написанный для PDP-1 в 1964.

Отладочная печать

Самый старый из методов отладки. Иногда его называют «printf() отладкой», из-за функции printf() стандартной библиотеки языка Си. Заключается в выводе, обычно на экран, информации о переменных или выполняющихся условиях. Несмотря на свою старость, до сих пор является удобным способом отладки программный проектов при должном количестве выводимой информации. Результаты отладки могут быть направлены в файл, для последующего внимательного разбора.

Отладка «после смерти»

Отладка программы после того, как программа неисправимо сломалась. Суть постмортем отладки заключается в анализе логов, просмотра состояния стека вызовов и слепка памяти на момент появления исключительной ситуации.

Отладка методом «волчья ограда»

Или методом бисекции. Впервые описан в журнале ACM [56]. Суть метода заключается в нахождении места ошибки через отсечение корректно работающих областей кода до момента, пока разработчик не попадет на некорректно работающую область. Удобно применять вместе с просмотром стека вызовов до момента происхождения исключительной ситуации. Яркий пример применения – команда `git bisect`, позволяющая быстро найти коммит, в котором впервые появилась ошибка.

Отладка методом записи и воспроизведения

Данный тип отладки подразумевает предварительную запись работы программы и последующее отлаживание уже сделанной записи. Это позволяет лучше исследовать причины ошибки, а так же отлавливать и анализировать трудновоспроизводимые ситуации, появляющиеся, например, из-за случайных событий.

При разработке ПМ АПНДВ, не смотря на то, что TDD рекомендует писать тесты при любой ошибке, а не заходить в отладчик, комбинирование обоих

подходов повышает производительность разработчика, так как в одних ситуациях удобнее и быстрее всего отладить совсем небольшую ошибку, а не писать для нее тест и запускать после этого все тесты модуля. Данная практика является наиболее распространенной среди разработчиков, применяющих TDD. Из всех перечисленных методов, для отладки ПМ АПНДВ использовалось два: отладочная печать и интерактивная отладка. Отладочная печать была удобна при внесении изменений между тестами, так как дополняла их результаты, а интерактивная отладка при анализе процесса исполнения ПМ АПНДВ. Для интерактивной отладки использовался отладчик GDB рис. 3.2.

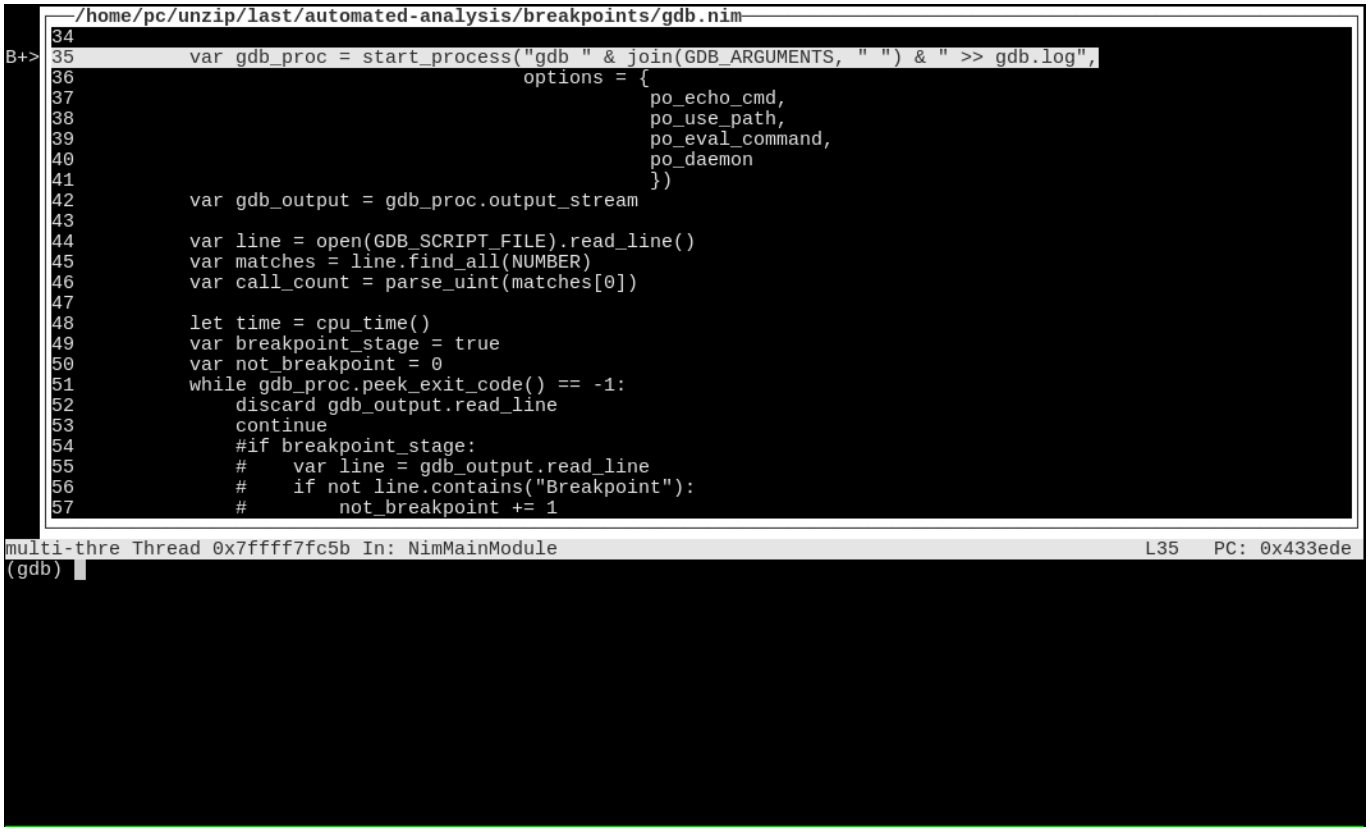


Рисунок 3.2 — Отладка ПМ АПНДВ в GDB

На рис. 3.2 можно увидеть терминальный интерфейс отладчика. Он запускается передачей GDB аргумента `-tui`, или выполнением команды `layout next` из командного интерфейса GDB. Это не единственный вид интерфейса, который предоставляет GDB. Есть, например, `split` интерфейс рис. 3.3, который позволяет одновременно видеть, как исполняемые машинные инструкции, так и исходный код программы.

Независимо от выбранного интерфейса, GDB оставляет командную строку для управления процессом отладки, которая обязательно начинается с `(gdb)`. Через нее происходит управление отладчиком – установка точек останова, просмотр

адресов памяти, переменных, изменение значений. GDB имеет большое количество команд значительно облегчающих отладку программ, а на их основе можно сделать мета-команды, объединяющие функционал нескольких команд.

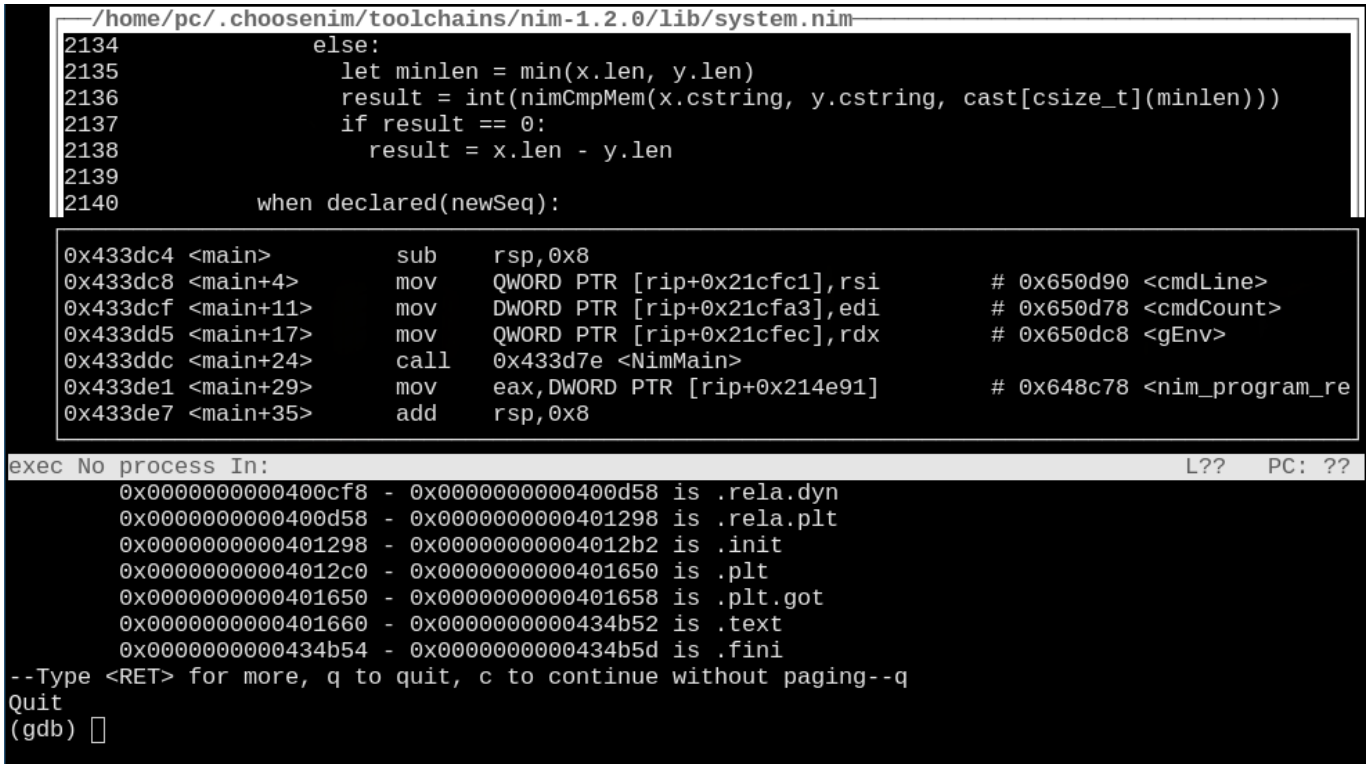


Рисунок 3.3 — layout split в GDB

Начать отлаживать любую программу в GDB можно двумя способами:

– запустить отладчик передав ему в качестве аргумента путь до отлаживаемого файла;
– подключиться к уже работающему процессу.

При как при отладке процесса, так и исполняемого файла GDB запустится с приветствием, содержащим версию и год, в который она была выпущена, а так же попытается найти отладочные символы не только для отлаживаемого процесса или файла, но и для используемых динамических библиотек (если таковые используются программой), пример в листинге 3.3. Если же отладочные символы не нашлись, то GDB сообщит об этом: `(No debugging symbols found in имя-файла)`

Листинг 3.3 Отладка Bash

```
Attaching to process 16108
Reading symbols from /bin/bash...
Reading symbols from /lib/x86_64-linux-gnu/libtinfo.so.5...
Reading symbols from /lib/x86_64-linux-gnu/libdl.so.2...
(No debugging symbols found in /lib/x86_64-linux-gnu/libdl.so.2)
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...
(No debugging symbols found in /lib/x86_64-linux-gnu/libc.so.6)
```

Список литературы

1. *SophosLabs.* Compile-a-virus – W32/Induc-A [Текст] / SophosLabs. — 2009. — URL: https://nakedsecurity.sophos.com/2009/08/18/compileavirus/ (дата обр. 18.08.2009).
2. *Томпсон, К.* Ken Thompson Hack [Текст] / К. Томпсон. — 1984. — URL: http://wiki.c2.com/?TheKenThompsonHack.
3. *Алексеев, А.* Краткий обзор статических анализаторов кода на C/C++ [Текст] / А. Алексеев. — 2016. — URL: https://eax.me/c-static-analysis/ (дата обр. 11.05.2016).
4. *Microsoft.* Application Inspector [Текст] / Microsoft. — 2019. — URL: https://github.com/microsoft/ApplicationInspector.
5. *anti-malware.ru.* Недекларированные возможности [Текст] / anti-malware.ru. — URL: https://www.anti-malware.ru/threats/undeclared-capabilities.
6. *Ализар, А.* Stuxnet был частью операции «Олимпийские игры», которая началась еще при Буше [Текст] / А. Ализар. — 2012. — URL: https://xakep.ru/2012/06/02/58789/ (дата обр. 02.06.2012).
7. Приказ ФСТЭК России №21 [Текст]. — URL: https://fstec21.blogspot.com/2017/07/type-actual-security-threats.html.
8. *РФ, П.* Постановление Правительства РФ от 01.11.2012 № 1119 "Об утверждении требований к защите персональных данных при их обработке в информационных системах персональных данных" [Текст] / П. РФ. — 2012. — URL: http://www.consultant.ru/cons/cgi/online.cgi?req=doc&base=LAW&n=137356&fld=134&dst=1000000001,0&rnd=0.8479428420303414#0004810272834757212.
9. *scitools.* Features [Текст] / scitools. — URL: https://scitools.com/features/.
10. *Позняков, С.* GNU cflow [Текст] / С. Позняков. — URL: https://www.gnu.org/software/cflow/.
11. Introduction to .NET Core [Текст]. — URL: https://docs.microsoft.com/ru-ru/dotnet/core/introduction.

12. Gcov [Текст]. — URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
13. *Free Software Foundation, I.* GNU Debugger [Текст] / I. Free Software Foundation. — URL: https://www.gnu.org/software/gdb/.
14. *Bellard, F.* QEMU [Текст] / F. Bellard. — URL: https://www.qemu.org/.
15. *Rumpf, A.* Nim [Текст] / A. Rumpf. — URL: https://nim-lang.org/.
16. *Rossum, G. van.* python [Текст] / G. van Rossum. — URL: https://www.python.org/.
17. *Wall, L.* Perl [Текст] / L. Wall. — URL: https://www.perl.org/.
18. pylint [Текст]. — URL: https://www.pylint.org/.
19. pyflakes [Текст]. — URL: https://github.com/PyCQA/pyflakes.
20. What "Batteries Included"Means [Текст]. — URL: https://protocolostomy.com/2010/01/22/what-batteries-included-means/ (дата обр. 22.01.2010).
21. pip [Текст]. — URL: https://pypi.org/project/pip/.
22. pip [Текст]. — URL: https://pypi.org/project/pip/.
23. PyPy [Текст]. — URL: https://www.pypy.org/.
24. Jython [Текст]. — URL: https://www.jython.org/.
25. Iron Python [Текст]. — URL: https://ironpython.net/.
26. AWK [Текст]. — URL: http://www.awklang.org/.
27. sed [Текст]. — URL: https://www.gnu.org/software/sed/.
28. JavaScript [Текст]. — URL: https://www.javascript.com/.
29. Move semantics [Текст]. — URL: https://nim-lang.org/docs/destructors.html#move-semantics.
30. A garbage collector for C and C++ [Текст]. — URL: https://www.hboehm.info/gc/.
31. Getting to Go: The Journey of Go's Garbage Collector [Текст]. — URL: https://blog.golang.org/ismmkeynote.
32. Nimble [Текст]. — URL: https://github.com/nim-lang/nimble.
33. reStructuredText. Markup Syntax and Parser Component of Docutils [Текст]. — URL: https://docutils.sourceforge.io/rst.html.

34. Cygwin [Текст]. — URL: https://www.cygwin.com/.

35. Aporia [Текст]. — URL: https://github.com/nim-lang/Aporia/.

36. Atom [Текст]. — URL: https://atom.io/.

37. Sublime Text [Текст]. — URL: https://www.sublimetext.com/.

38. Visual Studio Code [Текст]. — URL: https://code.visualstudio.com/.

39. Vim [Текст]. — URL: https://www.vim.org/.

40. Electron [Текст]. — URL: https://www.electronjs.org/.

41. NERDTree [Текст]. — URL: https://github.com/preservim/nerdtree.

42. Tabular [Текст]. — URL: https://github.com/preservim/nerdtree.

43. vim-polyglot [Текст]. — URL: https://github.com/sheerun/vim-polyglot.

44. undotree [Текст]. — URL: https://github.com/mbbill/undotree.

45. rainbow [Текст]. — URL: https://github.com/luochen1990/rainbow.

46. What is a software architecture? [Текст]. — URL: https://www.ibm.com/developerworks/rational/library/feb06/eeles/index.html (дата обр. 15.02.2006).

47. Unix Design Philosophy [Текст]. — 1995. — URL: https://wiki.c2.com/?UnixDesignPhilosophy.

48. __fastcall [Текст]. — URL: https://docs.microsoft.com/ru-ru/cpp/cpp/fastcall?view=vs-2019.

49. JSON Compilation Database Format Specification [Текст]. — URL: https://clang.llvm.org/docs/JSONCompilationDatabase.html.

50. Build EAR (BEAR) [Текст]. — URL: https://github.com/rizsotto/Bear.

51. PCRE - Perl Compatible Regular Expressions [Текст]. — URL: https://www.pcre.org/.

52. The Mother of All Demos, presented by Douglas Engelbart (1968) [Текст]. — URL: https://www.youtube.com/watch?v=yJDv-zdhzMY.

53. Gooey [Текст]. — URL: https://github.com/chriskiehl/Gooey.

54. git –fast-version-control [Текст]. — URL: https://git-scm.com/.

55. Introduction to Test Driven Development (TDD) [Текст]. — URL: http://agiledata.org/essays/tdd.html.

56. The “Wolf Fence” algorithm for debugging [Текст]. — URL: https://dl.acm.org/doi/10.1145/358690.358695.