

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 С и С++ 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.
Список сокращений и условных обозначений	4
Словарь терминов	5
Введение	7
Раздел 1. Исследовательский раздел	10
1.1 Процесс сертификации ПО на отсутствие НДВ	10
1.2 Классификация НДВ	11
1.2.1 По применению	12
1.2.2 По целям	12
1.3 Степень опасности НДВ	13
1.4 Обзор программных решений для сертификации ПО на отсутствие НДВ	15
1.4.1 Сравнение статических анализаторов	15
1.4.2 Сравнение динамических анализаторов	18
1.5 Постановка задачи ВКР	22
1.6 Выводы по разделу	23
Раздел 2. Конструкторский раздел	24
2.1 Обоснование выбора языка программирования и среды разработки	24
2.1.1 Сравнение языков программирования	24
2.1.2 Сравнение сред разработки	27
2.2 Архитектура ПМ АПНДВ	30
2.2.1 Организация передачи информации между компонентами ПМ АПНДВ	30
2.2.2 Схема данных	34
2.2.3 Алгоритм работы программы	34
2.2.4 Разработка консольного интерфейса ПМ АПНДВ	40
2.2.5 Разработка графического интерфейса ПМ АПНДВ	43
2.3 Выводы по разделу	45

Раздел 3. Технологический раздел	46
3.1 Процесс разработки ПМ АПНДВ	46
3.1.1 Сборка программы на языке Nim	47
3.1.2 Тестирование ПМ АПНДВ	48
3.1.3 Отладка ПМ АПНДВ	52
Список литературы	56

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 → 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 определенного 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 недеklarированных 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.

До разработки ПМ АПНДВ



Рисунок 1 — Процесс проведения сертификации раньше

Получается, что на рынке невозможно найти комбинированных решений, с помощью которых 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 С/С++.

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 59 страниц, 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) определение 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 по несколькими способами, в зависимости от их целей и применения.

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 удешевлению 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 $0 \leq Y \leq 0.3$, то возможность реализации угрозы признается низкой;
- если $0.3 < Y \leq 0.6$, то возможность реализации угрозы признается средней;
- если $0.6 < Y \leq 0.8$, 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 проявления деструктивного эффекта НДВ в работающем ПО.

Затем выносится решение о проведении анализа ПО на НДВ в процесс сертификации или его игнорирование, как не критичного.

Сейчас анализ программы на НДВ происходит вручную:

- 1) с помощью специального ПО проводят статический анализ исходных кодов программного проекта;
- 2) с помощью отладчиков 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 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 конкретных программ. Он будет упомянут как в сравнении статических анализаторов, так и динамических. 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 создании 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.

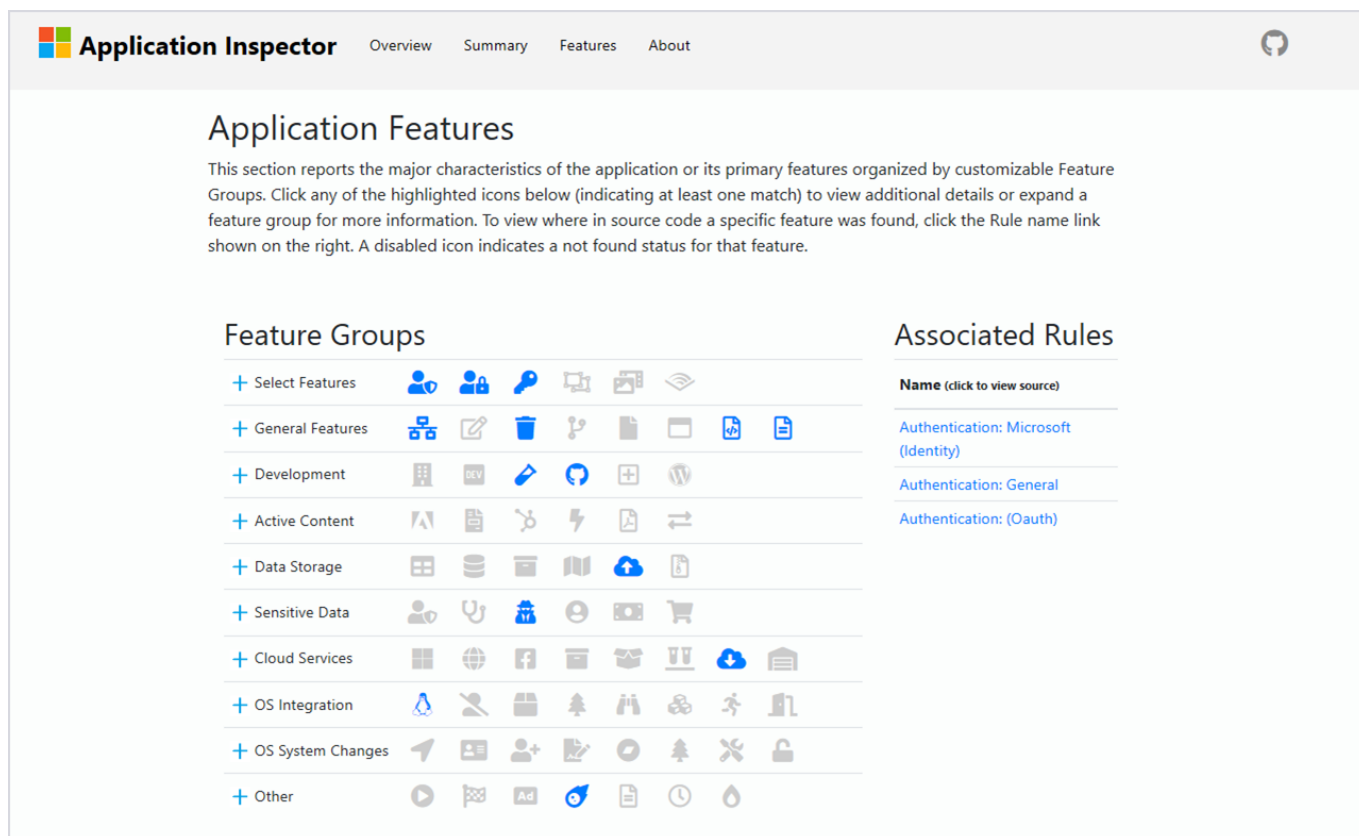


Рисунок 1.1 — Отчет Microsoft Application Inspector

Является бесплатным ПО, с открытым исходным кодом. Позволяет защититься от НДВ только на переднем плане, так как анализирует исходные коды подключенных модулей, но бессилён 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 различные их редакции. 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

Утилита анализа ядра Linux от ООО Фирма «Анкад». Помимо статического и динамического анализа имеет сравнительный анализ, а так же команды для проверки специфических ситуаций, таких как накопление информации в ядре и других.

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 возможный 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

Свойства \ Название программы	Gcov [12]	GDB [13]	QEMU [14]	Kernel analyzer
Кросс-платформенность	Да	Да	Да	Да
Открытость исходного кода	Да	Да	Да	Да
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
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 lscov 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;
- ARM;
- AVR;
- H8/300;
- Altera Nios/Nios II;
- System/370;

LCOV - code coverage report

Current view: [top level](#) - [example/methods](#) - [iterate.c](#) ([source](#) / [functions](#))

Test: [Basic example](#) ([view descriptions](#))

Date: 2019-03-04 16:39:23

Legend: Lines: hit not hit | Branches: + taken - not taken # not executed

	Hit	Total	Coverage
Lines:	8	8	100.0 %
Functions:	1	1	100.0 %
Branches:	4	4	100.0 %

	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

- System 390;
- X86 и X86-64;
- IA-64 "Itanium";
- Motorola 68000;
- MIPS;
- PA-RISC;
- PowerPC;
- SuperH;
- SPARC;
- VAX.

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.

```

test.c
48
49     int main ()
50     {
51     child_pid = fork();
52     if (child_pid > 1) {
53         printf("Parent %d is waiting...\n", getpid());
54         sleep(3);
55
56         if (sigterm() == 0)
57         {
58             printf("Parent exit\n");
59             //exit(0);
60         }
61
62         printf("Starting alarm\n");
63         signal(SIGALRM, sigkill);
64         alarm(3);
65
66         wait(NULL);
67         sleep(4);
68         printf("Parent exit\n");
69         exit(0);
70     }
71     else if (child_pid == 0) {
72         printf("Starting child process - %d\n", getpid());

```

```

native process 19598 In: main
<http://www.gnu.org/software/gdb/documentation/>.
--Type <RET> for more, q to quit, c to continue without paging--

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...
(gdb) b main
Breakpoint 1 at 0x400822: file test.c, line 51.
(gdb) run
Starting program: /home/pc/Coding/a.out

Breakpoint 1, main () at test.c:51
(gdb)

```

Рисунок 1.3 — Терминальный интерфейс GDB

QEMU

QEMU – Быстрый эмулятор процессоров, поддерживает множество процессорных архитектур, предоставляет возможность сохранять сгенерированный машинный код. К недостаткам можно отнести медленную, по сравнению с отладчиком работу, так как эмулятору приходится преобразовывать каждую инструкцию запущенной программы в машинный код процессора, на котором он запущен.

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;

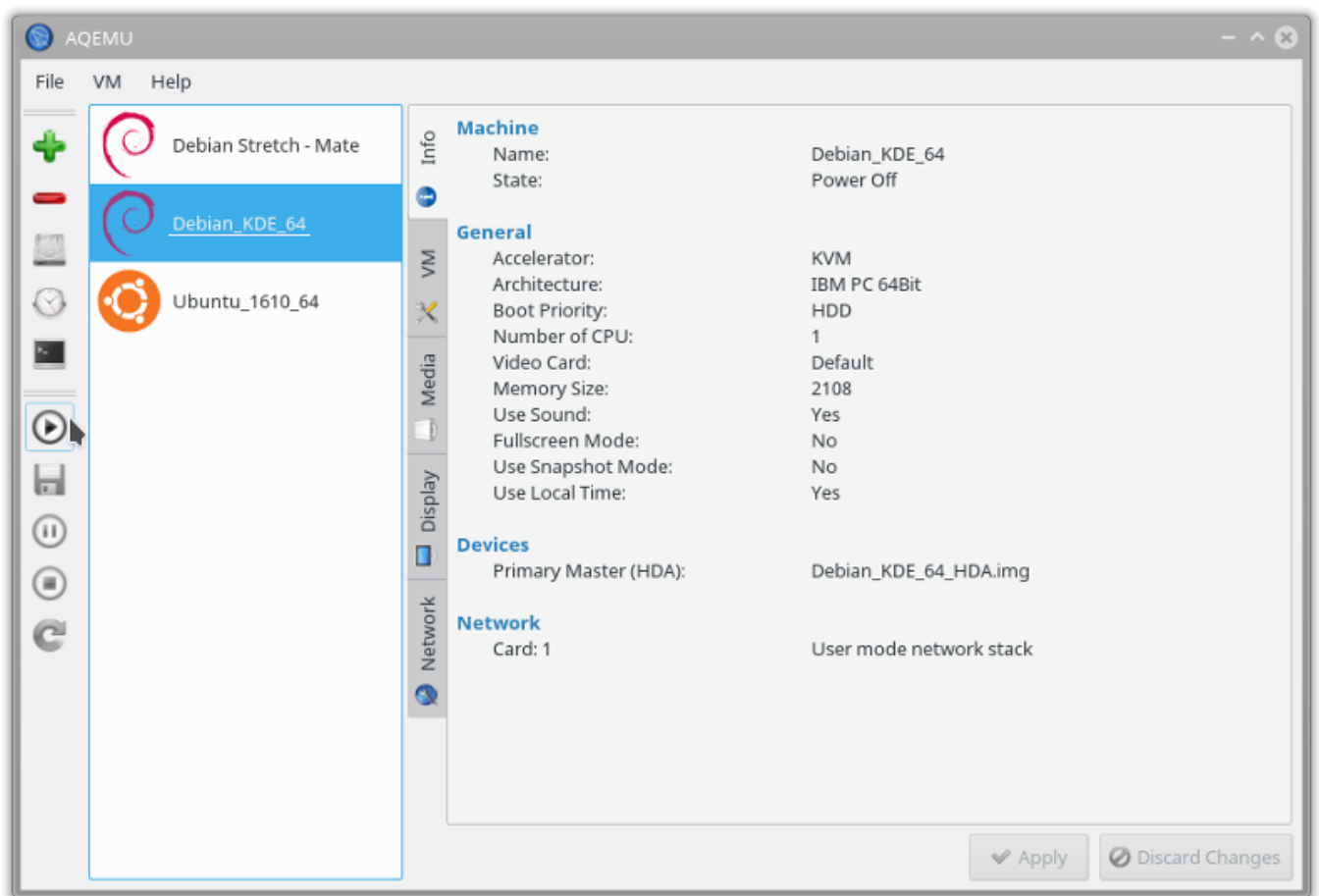


Рисунок 1.4 — Графическая оболочка AQEMU для эмулятора QEMU

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.

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 нормативно-правовые обоснования необходимости проведения данной процедуры, а так же представлены последствия игнорирования возможности внедрения НДС 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.

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 трудноулаживаемым ошибкам. Негативное влияние можно сгладить с помощью указания типов при объявлении переменных 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) gc [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

IDE/Редактор Свойства	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 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.

```

<ssing.nim> 6:[parse_log.nim] 7:[set_breakpoints.nim] 8:[gdb.nim] 9:[comparative_analysis.nim] 10:[aggregation.nim]
18 import os
17 import posix
16 import tables
15 import osproc
14 import streams
13 import parseopt
12 import strutils
11 import strformat
10
9 import aux/parsing
8
7 from memfiles import open, close
6
5 var parser = init_opt_parser(commandline_params())
4 # B -e передается путь до исследуемой программы
3 # B -p передаются аргументы для программы
2
1 var cmd_arguments = {"e" : "",
30 "p" : ""}.to_table()
1 while true:
2   parser.next()
3   case parser.kind
4   of cmd_end:
5     break
6   of cmdLongOption:
7     if parser.key == "-e":
8       cmd_arguments["e"] = parser.value
9     else if parser.key == "-p":
10      cmd_arguments["p"] = parser.value
11      break
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
26
```

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.

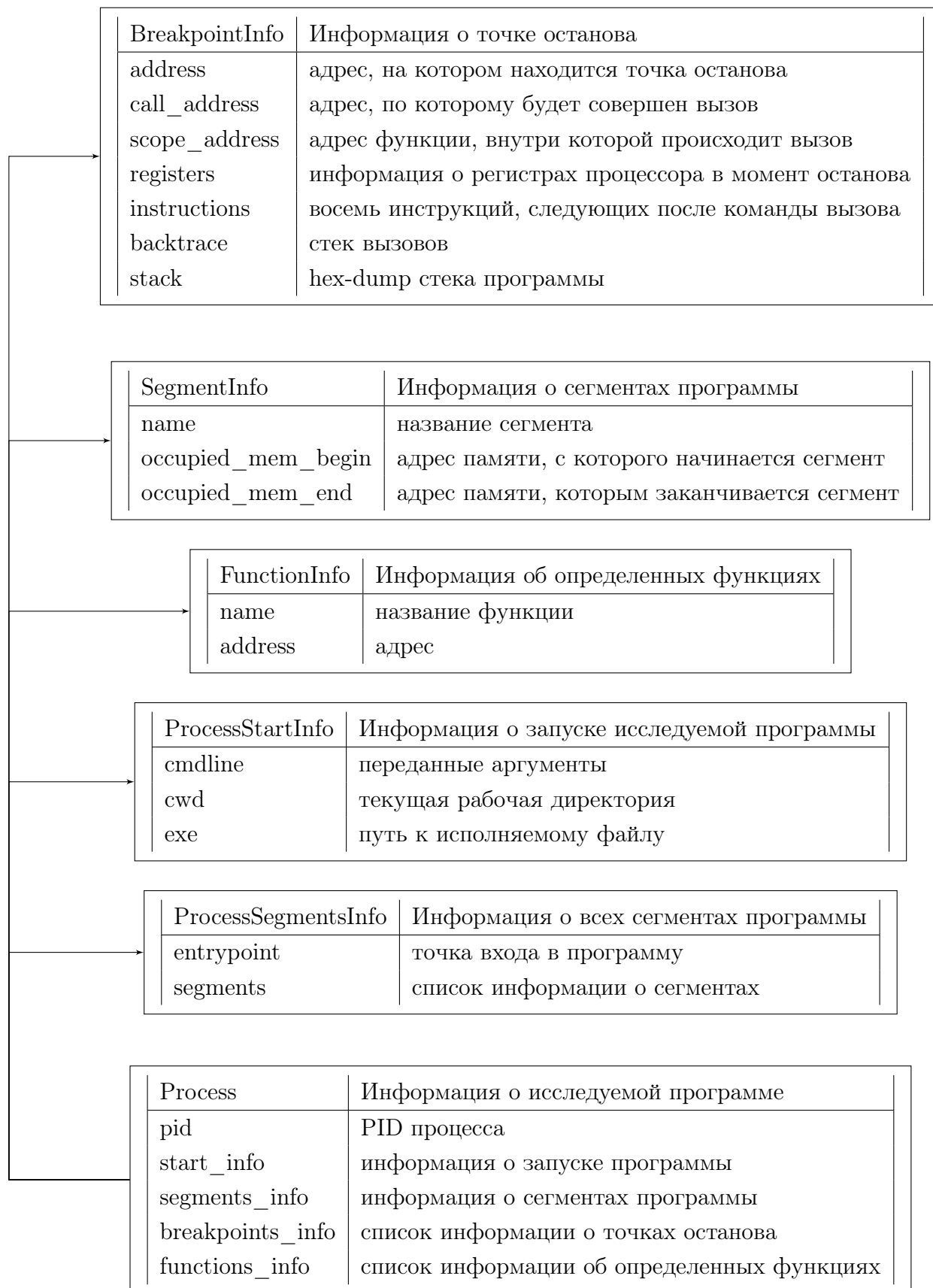


Рисунок 2.2 — Сохраняемые структуры динамического анализа

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 которой 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 анализа. 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 функции 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.

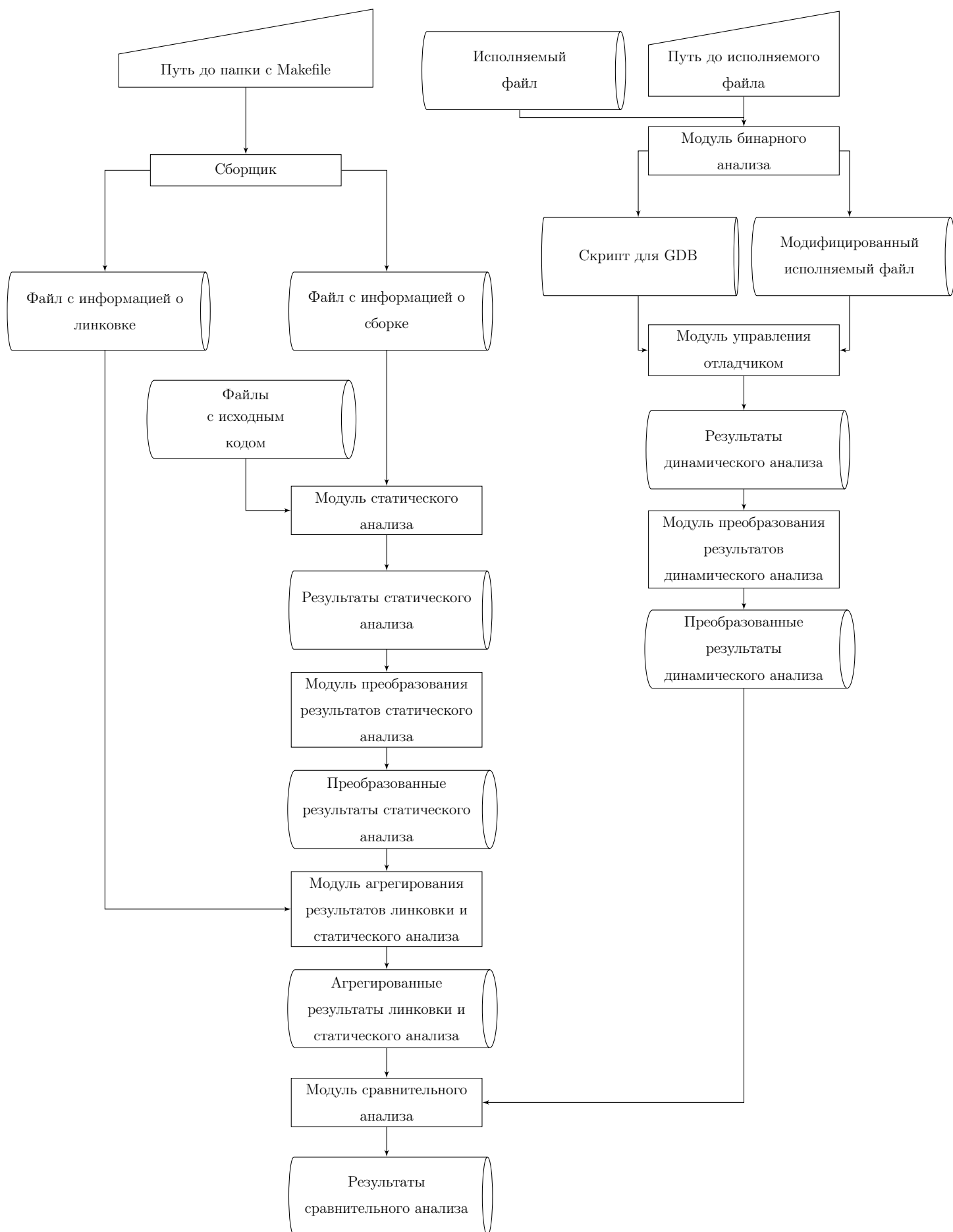


Рисунок 2.4 — Схема данных ПМ АПНДВ

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 tar-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 остаться 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 ограничивается 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 Sflow, которой на вход подаются аргументы компиляции, взятые из compilation database, полученной на предыдущем шаге, а так же сами файлы с исходными кодами.

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

Листинг 2.1 Формат записи в отчете Sflow

```
{/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/>:

...

```

5

Данный формат файла легко поддается разбору с помощью регулярных выражений. В ПМ АПНДВ использовалась библиотека регулярных выражений PCRE [51]. Не смотря на то, что Sflow 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 шестнадцатичной 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
```

– определяется команда `xxd`, которая будет добавлять в лог динамического анализа дампы 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 исходниках 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:

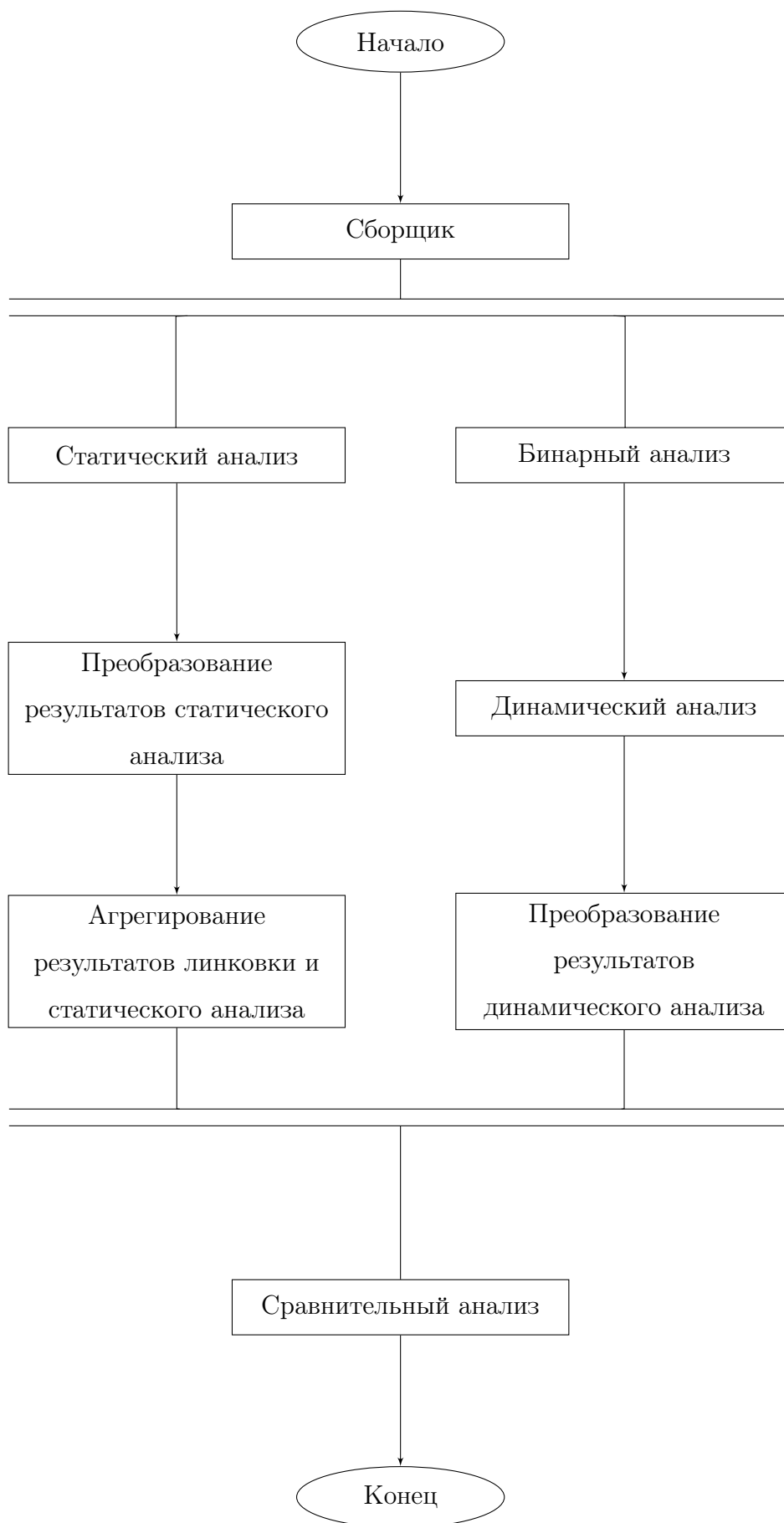


Рисунок 2.5 — Алгоритм работы ПМ АПНДВ

- 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.

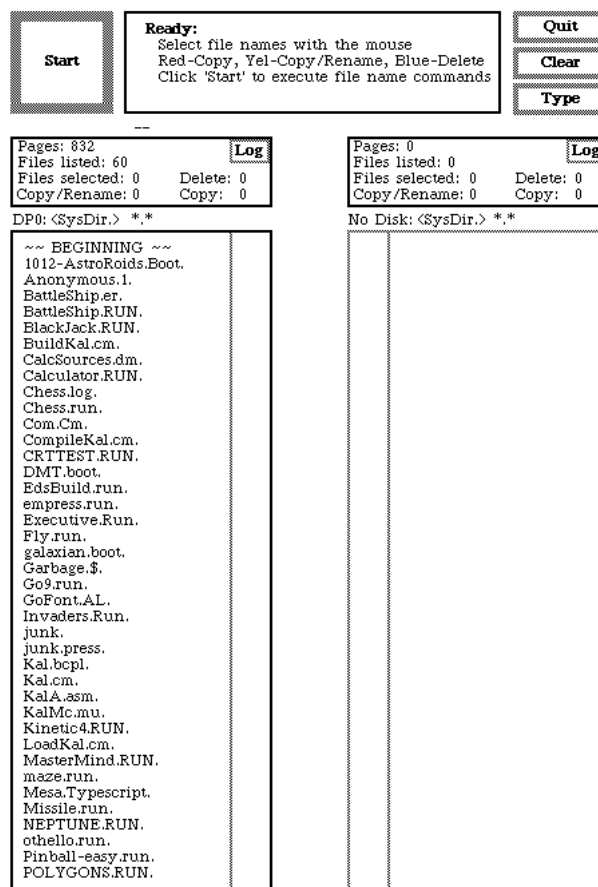


Рисунок 2.6 — Интерфейс файлового менеджера Xerox Alto

И сейчас, не смотря на удобство 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 Gooeu [53].



Обязательные аргументы

Путь до папки с исходными кодами сертифицируемого ПО

Путь до исполняемого файла

Рисунок 2.7 — Графический интерфейс ПМ АПНДВ

Gooeu 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**, Gooeu 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 Gooeu 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 файлах 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, вернуть 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 англ. 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 агрегированием результатов
+## Этот модуль занимается агрегированием результатов
  ## линковки и статического анализа:
  ## Всем CflowConstruct выставляется ‘‘text_offset’’ -- адрес функции в бинарнике
  ##
```

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.

build

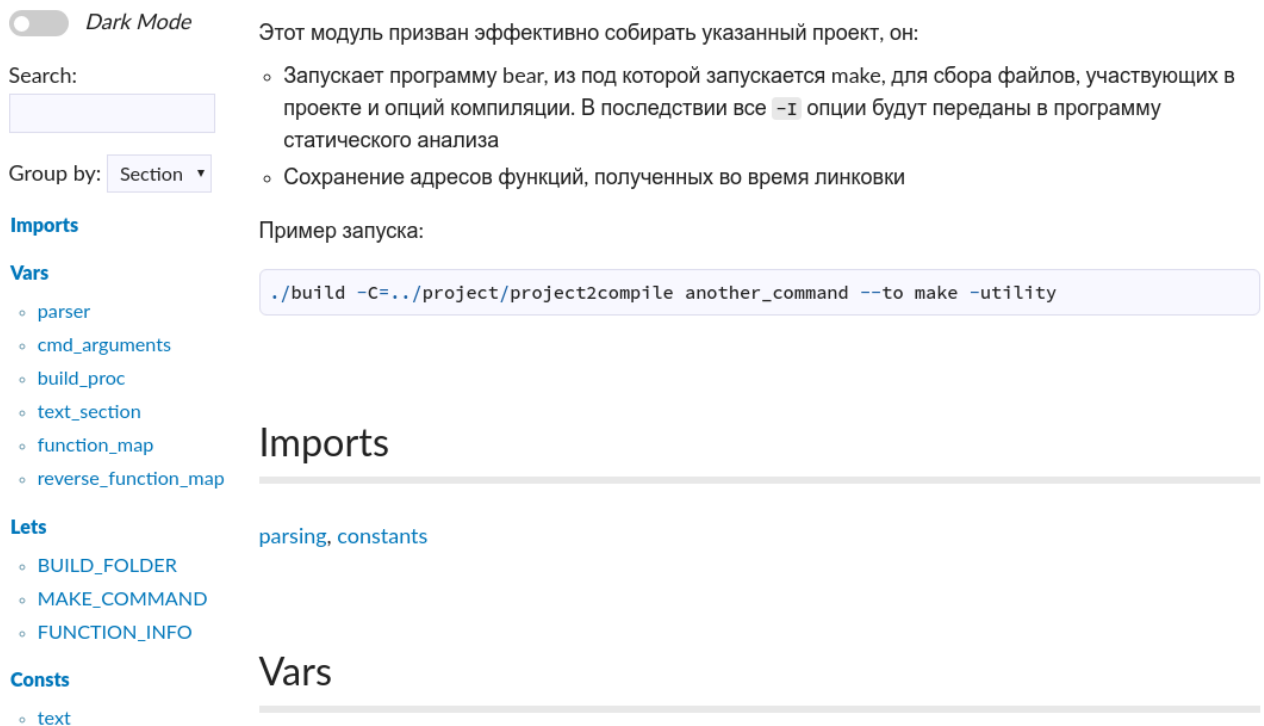


Рисунок 3.1 — Сгенерированная документация по одному из модулей ПМ АПНДВ

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 программного обеспечения – 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;

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

3) тестирование методом серого ящика – ПО тестируется с неполным знанием работы внутренних механизмов программы.

ПМ АПНДВ разрабатывался с использованием техники test-driven development [55] – разработки через тестирование, или TDD.

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 функциональность должна начинаться с написания теста для нее. Для написания теста разработчик должен хорошо

понимать специфику добавляемой функциональности и требования, накладываемые 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 организован. 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.

Таблица 13 — Сценарии модульного тестирования ПМ АПНДВ

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

3.1.3 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.

```

/home/pc/unzip/last/automated-analysis/breakpoints/gdb.nim
34
B+> 35   var gdb_proc = start_process("gdb " & join(GDB_ARGUMENTS, " ") & " >> gdb.log",
36                                     options = {
37                                         po_echo_cmd,
38                                         po_use_path,
39                                         po_eval_command,
40                                         po_daemon
41                                     })
42   var gdb_output = gdb_proc.output_stream
43
44   var line = open(GDB_SCRIPT_FILE).read_line()
45   var matches = line.find_all(NUMBER)
46   var call_count = parse_uint(matches[0])
47
48   let time = cpu_time()
49   var breakpoint_stage = true
50   var not_breakpoint = 0
51   while gdb_proc.peek_exit_code() == -1:
52       discard gdb_output.read_line
53       continue
54       #if breakpoint_stage:
55       #   var line = gdb_output.read_line
56       #   if not line.contains("Breakpoint"):
57       #       not_breakpoint += 1

```

multi-thre Thread 0x7ffff7fc5b In: NimMainModule L35 PC: 0x433ede
(gdb)

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.

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

```

/home/pc/.choosenim/toolchains/nim-1.2.0/lib/system.nim
2134         else:
2135             let minlen = min(x.len, y.len)
2136             result = int(nimCmpMem(x.cstring, y.cstring, cast[csize_t](minlen)))
2137             if result == 0:
2138                 result = x.len - y.len
2139
2140         when declared(newSeq):

0x433dc4 <main>      sub    rsp,0x8
0x433dc8 <main+4>    mov    QWORD PTR [rip+0x21cfc1],rsi    # 0x650d90 <cmdLine>
0x433dcf <main+11>   mov    DWORD PTR [rip+0x21cfa3],edi    # 0x650d78 <cmdCount>
0x433dd5 <main+17>   mov    QWORD PTR [rip+0x21cfec],rdx    # 0x650dc8 <gEnv>
0x433ddc <main+24>   call   0x433d7e <NimMain>
0x433de1 <main+29>   mov    eax,DWORD PTR [rip+0x214e91]    # 0x648c78 <nim_program_re
0x433de7 <main+35>   add    rsp,0x8

exec No process in:                                L??  PC: ??
0x0000000000400cf8 - 0x0000000000400d58 is .rela.dyn
0x0000000000400d58 - 0x0000000000401298 is .rela.plt
0x0000000000401298 - 0x00000000004012b2 is .init
0x00000000004012c0 - 0x0000000000401650 is .plt
0x0000000000401650 - 0x0000000000401658 is .plt.got
0x0000000000401660 - 0x0000000000434b52 is .text
0x0000000000434b54 - 0x0000000000434b5d is .fini
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) █

```

Рисунок 3.3 — layout split в GDB

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 3.2. Если же отладочные символы не нашлись, то GDB сообщит об этом: (No debugging symbols found in имя-файла)

Листинг 3.2 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> (датаabr. 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/rizotto/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>.