ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Санкт-Петербургский государственный университет телекоммуникаций
им. проф. М.А. Бонч-Бруевича»

На правах рукописи

Израилов Константин Евгеньевич

МЕТОД АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ В ТЕЛЕКОММУНИКАЦИОННЫХ УСТРОЙСТВАХ

Специальность 05.13.19 – Методы и системы защиты информации,
информационная безопасность

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель
доктор технических наук, профессор
Буйневич Михаил Викторович

Санкт-Петербург – 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность темы исследования. Безопасность современных телекоммуникационных систем определяется, в том числе, уязвимостями в программном обеспечении входящих в их состав телекоммуникационных устройств. Согласно открытым базам (таким, как NVD, CVE, OSVDB и др.) ежегодно наблюдается рост обнаруживаемых уязвимостей. При этом сфера применяемых в России телекоммуникационных устройств является на 90% «импортозависимой», вынуждая использовать заведомо небезопасное программное обеспечение зарубежного производства.

Существующее же множество способов поиска уязвимостей в исходном коде программного обеспечения в данном случае не подходит, поскольку преобладающее количество телекоммуникационных устройств поставляется с машинным кодом, способы поиска по которому существенно ограничены и в разы более трудоемки. При этом их целью, в основном, являются достаточно формализуемые низкоуровневые уязвимости – ошибки в реализации вычислений и доступе к данным. Средне- и высокоуровневые уязвимости – ошибки в логике работы алгоритмов и общей архитектуре, обнаруживаемые по субъективным признакам и с привлечением экспертов информационной безопасности, на сегодня не имеют развитых способов поиска.

Таким образом, основное противоречие рассматриваемой предметной области заключается в росте количества и разнообразия средне- и высокоуровневых уязвимостей при ограниченном количестве и возможностях экспертов информационной безопасности. Одной из существенных причин, порождающих указанное противоречие, является низкая производительность работы экспертов, обусловленная использованием для поиска средне- и высокоуровневых уязвимостей представлений кода, продуцируемых традиционными средствами реверс-инжиниринга.

В этих условиях требуется принципиально новый вид представления машинного кода, специализированный для поиска средне- и высокоуровневых уяз-

вимостей – так называемое алгоритмизированное представление. Это предполагает разработку соответствующего метода алгоритмизации и реализующего его средства, что и обуславливает актуальность темы настоящего исследования.

Степень разработанности темы. Методологические вопросы информационной безопасности телекоммуникационных систем, задавшие предпосылки к исследованию, рассматривались крупными Российскими учеными, такими как М.В. Буйневич [1-3], Б.С. Гольдштейн [4-6], О.В. Казарин [7], В.И. Коржик [8-9], И.В. Котенко [10-11], Е.А. Крук [12-13], А.Е. Кучерявый [14-16], Н.А. Соколов [17-18], Р.М. Юсупов [19-21], Ю.К. Язов [22], В.А. Яковлев [23] и др. Безопасность программного обеспечения, используемого и в телекоммуникационных устройствах, а также непосредственный поиск в нем уязвимостей были рассмотрены в работах А.П. Баранова [24-26], М.А. Еремеева [27-28], Д.П. Зегжды [29], П.Д. Зегжды [30], В.П. Иванникова [31-32], А.А. Корниенко [33], А.Г. Ломако [34-35], В.Ю. Осипова [36], В.Г. Шведа [37-38] и др. Возможностями реверс-инжиниринга, примененными в интересах алгоритмизации, занимались такие ученые, как А.И. Аветисян [39-40], В.А. Падарян [41], Е.Н. Трошина [42-44], А.В. Чернов [45-46], C. Cifuentes [47-48] и др. Существующие представления программного кода, а также подходы к созданию альтернативных, использованные в интересах метода алгоритмизации, были получены на основании работ С.В. Диасамидзе [49-50], А.С. Маркова [51-53], Ф.А. Новикова [54-55], С.В. Полякова [56], Д.А. Эделя [57], B. Shneiderman [58-59] и др.

Несмотря на достаточное освещение указанных областей, тем не менее, проблемные вопросы алгоритмизации машинного кода в интересах поиска средне- и высокоуровневых уязвимостей оставлены практически без внимания.

Цель и задачи. Целью работы является повышение эффективности поиска уязвимостей в машинном коде телекоммуникационных устройств путем его алгоритмизации.

Для достижения цели исследования в работе были поставлены и решены следующие задачи:

1) Проанализировать способы нейтрализации уязвимостей в программном коде;

2) Идентифицировать метаданные в машинном коде, соответствующие парадигмам разработки программного кода для телекоммуникационных устройств;

3) Исследовать модели машинного кода с позиции структурных метаданных;

4) Построить гипотетическую схему алгоритмизации машинного кода на базе его модели;

5) Синтезировать метод, реализующий схему алгоритмизации;

6) Разработать архитектуру программного средства автоматизации метода алгоритмизации;

7) Реализовать прототип программного средства алгоритмизации и произвести его базовое тестирование;

8) Сформировать критерии и создать методики для оценки свойств алгоритмизации машинного кода;

9) Произвести комплексную оценку эффективности алгоритмизации в интересах поиска уязвимостей;

10) Предложить перспективы применения и развития метода алгоритмизации.

Объект исследования – машинный код с уязвимостями.

Предмет исследования – восстановление алгоритмов работы программного кода.

Научно-техническая задача – разработка метода и средства алгоритмизации машинного кода в интересах поиска средне- и высокоуровневых уязвимостей в программном обеспечении телекоммуникационных устройств.

Научная новизна. Научная новизна работы определяется новой предметной областью и новизной полученных результатов и состоит в том, что:

– предложено представление машинного кода, отличное от классического (линейного) наличием иерархически-связанных структурных элементов – структурных метаданных и разноуровневых уязвимостей;

– в отличие от традиционных методов реверс-инжиниринга машинного кода, используется концептуально новый подход к алгоритмизации путем моделирования машинного кода с последующей итерационной оптимизацией модели и генерацией оригинального представления;

– впервые разработана архитектура программного средства реверс-инжиниринга машинного кода, ориентированная на создание и обработку его модели, а также использующая оригинальные модули выделения сигнатур подпрограмм, оптимизации и лаконизации внутреннего представления, поиска среднеуровневых уязвимостей;

– создан оригинальный комплекс методик и критериев оценки алгоритмизации машинного кода и ручного поиска средне- и высокоуровневых уязвимостей.

Теоретическая и практическая значимость работы. Теоретическая значимость научных положений, изложенных в работе, состоит в следующем:

– установлено соответствие между инструкциями машинного кода и его структурными метаданными;

– расширен класс методов для восстановления исходного кода программ в части их алгоритмов;

– разработаны классы объектов и процедур для моделирования низкоуровневого представления программы;

– сформированы критерии понятности представления и алгоритмизированности машинного кода.

Практическая значимость результатов проведенных исследований состоит в следующем:

– структурная модель позволяет формулировать научно-обоснованные требования к решению задач, связанных с восстановлением архитектуры и алгоритмов машинного кода, а также с поиском его уязвимостей;

– метод алгоритмизации может быть использован для восстановления архитектуры и алгоритмов машинного кода в человеко-ориентированный вид, подходящий для поиска средне- и высокоуровневых уязвимостей;

– архитектура предоставляет возможность реализации программных средств для инвариантного преобразования низкоуровневого представления программ в высокоуровневое;

– методики оценки применимы для обоснованного выбора среди методов и средств, используемых для поиска уязвимостей в машинном коде телекоммуникационных устройств.

Часть основных научных результатов используются в учебном процессе на кафедре защищенных систем связи СПбГУТ при подготовке и проведении лекционно-практических занятий для направления 10.00.00 – «Информационная безопасность» по учебным дисциплинам «Ассемблер в задачах защиты информации», «Вредоносное программное обеспечение» и «Реверс-инжиниринг системного программного обеспечения». Другая часть реализована в ООО «Астрософт» при разработке архитектуры системного программного обеспечения и в методическом обеспечении лаборатории тестирования.

Методология и методы исследования. Для решения поставленных задач использовались как классические, так и современные методы исследования, а именно:

– системный, причинно-следственный и сравнительный анализ был применен в равной степени для получения практически всех основных научных результатов;

– сбор, систематизация и анализ научно-технической информации предметной области позволили создать структурную модель машинного кода с уязвимостями;

– функциональный и структурный синтез использовался для создания метода алгоритмизации;

– с помощью методов бальной оценки и анализа иерархий был создан комплекс научно-методических средств, позволяющий оценивать эффективность алгоритмизации машинного кода.

Помимо общей методологии программирования, основой архитектуры программного средства алгоритмизации послужили теории компиляции и графов; также использовалось компьютерное моделирование для поддержки в нем структурной модели.

Положения, выносимые на защиту. Соискателем лично получены следующие основные научные результаты, выносимые на защиту:

1) Структурная модель машинного кода с уязвимостями;
2) Метод алгоритмизации машинного кода;
3) Архитектура программного средства алгоритмизации машинного кода;
4) Комплекс научно-методических средств оценки алгоритмизации машинного кода в интересах поиска уязвимостей.

Степень достоверности. Достоверность основных полученных результатов обеспечивается корректностью постановки научно-технической задачи исследования, строго обоснованной совокупностью ограничений и допущений, представительным библиографическим материалом, опорой на современную научную базу, корректным применением апробированных классических и современных методов исследования; и подтверждается непротиворечивостью полученных результатов практики восстановления программного кода и поиска в нем уязвимостей, широкой апробацией результатов на представительных научных форумах, а также получением свидетельства о государственной регистрации программы для ЭВМ.

Апробация результатов. Основные научные положения и результаты диссертации докладывались, обсуждались и получили одобрение на Пятом научном конгрессе студентов и аспирантов «ИНЖЭКОН-2012» (Санкт-Петербург, СПбГИЭУ, 25-26 апреля 2012 г.), Всероссийской научно-методической конференции «Фундаментальные исследования и инновации в национальных исследовательских университетах» (Санкт-Петербург, СПбГПУ, 17-18 мая 2012 г.), научно-практической конференции «Информационные технологии и непрерывность

бизнеса» (Санкт-Петербург, СПбГИЭУ, 8 ноября 2012 г.), X Международной научно-технической конференции «Новые информационные технологии и системы (НИТиС-2012)» (Пенза, ПГУ, 27–29 ноября 2012 г.), II и V Международных научно-технических и научно-методических конференциях «Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО-2013 и АПИНО-2016)» (Санкт-Петербург, СПбГУТ, 26–27 февраля 2013 г. и 10-11 марта 2016 г.), Шестом научном конгрессе студентов и аспирантов «ИНЖЭКОН-2013» (Санкт-Петербург, СПбГИЭУ, 18-19 апреля 2013 г.), Международной научно-технической конференции «Фундаментальные и прикладные исследования в современном мире» (Санкт-Петербург, СПбГПУ, 20-22 июня 2013 г.), 16-19-ой Международных конференциях «Передовые коммуникационные технологии (ICACT-2014, 2015, 2016, 2017)», (Пхёнчхан, Ю. Корея, 16-19 февраля 2014 г., 1-3 июля 2015 г., 31 января – 3 февраля 2016 г., 19-22 февраля 2017 г.), XIV Санкт-Петербургской Международной конференции «Региональная информатика (РИ-2014)» (Санкт-Петербург, 29-31 октября 2014 г.), IX Санкт-Петербургской межрегиональной конференции «Информационная безопасность регионов России (ИБРР-2015)» (Санкт-Петербург, 28-30 октября 2015 г.).

Публикации. Основные результаты диссертационного исследования опубликованы в 31-ом научном труде [60-90], из них: 9 – в рецензируемых научных изданиях из Перечня ВАК; 5 – в изданиях, входящих в международную систему цитирования Scopus; 1 – свидетельство о государственной регистрации программы для ЭВМ; 7 – статей в научных журналах; 9 – в сборниках научных статей, трудов, тезисов докладов и материалах конференций; 2 – отчеты о НИР.

Структура и объем работы. Диссертационная работа состоит из введения, основной части (содержащей 4 раздела), заключения, списка литературы и приложений.

Общий объем работы − 261 страница, из них основного текста – 185 страниц. Работа содержит 69 рисунков и 26 таблиц. Список литературы включает 175 библиографических источников.

1 АНАЛИЗ СТРУКТУРНЫХ МЕТАДАННЫХ МАШИННОГО КОДА

1.1 Анализ современных способов нейтрализации уязвимостей в программном коде телекоммуникационных устройств

Информатизация, стремительно проникающая во все сферы жизни, всецело поддерживается информационными технологиями. Единоличное использование информации редко имеет значимый смысл, что приводит к необходимости ее передачи на расстояние. Используемые для этого телекоммуникационные сети представляют собой распределенную систему телекоммуникационных устройств (далее – ТКУ) [89-90]. Работа алгоритмов последних, как правило, реализуются с помощью программного обеспечения (далее – ПО). Для этого создается исходный код программы (далее – ИК), собирается в машинный код образа (далее – МК), загружается в ТКУ и исполняется в его аппаратной среде. Наличие ошибок или уязвимостей (что более верно в рассматриваемом контексте) в таком программном коде (далее – ПрК) является причиной возникновения угроз передаваемой информации [91-94]. На сегодняшний же день не существует способа получения (в процессе разработки или эксплуатации) абсолютно безопасного кода – т. е. не имеющего уязвимостей, что является актуальной проблемой. Рассмотрим проблемные вопросы, решение которых гипотетически позволяет получить безопасный код для ТКУ.

1.1.1 Проблемные вопросы обеспечения безопасности программного кода

Произведем классификацию проблемных вопросов безопасности ПК [75, 80] на основании особенностей взаимосвязи защищаемой информации и ПрК, обрабатывающего эту информацию в ТКУ.

Вначале определим техническую составляющую этимологии понятия «уязвимость ПрК ТКУ», для чего рассмотрим типичную схему возникновения угроз информационной безопасности. Традиционно ею являются действия источника,

эксплуатирующего уязвимости [86], хотя могут быть и альтернативные точки зрения [61]. Источник угроз при этом может быть двух типов. Во-первых, ПрК может выступать сам как источник, если его уязвимость инициирует угрозу без каких-либо внешних воздействий – так называемый внутренний источник угроз. И, во-вторых, в ПрК содержится уязвимость, которой может сознательно воспользоваться злоумышленник – так называемый внешний источник угроз.

Уязвимостью ПрК ТКУ условимся считать некую его особенность (ошибку, слабость, недостаток и т. д.), используя которую возможно нарушение безопасности обрабатываемой информации. В таком случае уязвимость указывает на отличие реализации ПрК от той, которая была заложена в его изначальную идею – с этой позиции полностью безопасную. То есть, если код в точности исполняет свое изначальное предназначение, то он в принципе не должен иметь уязвимостей; в противном случае они являются производными его изначально небезопасной задумки.

С учетом рассмотренных особенностей проведем обзор основных доступных способов получения безопасного кода для ТКУ. Все они достаточно хорошо классифицируются с помощью следующей тривиальной схемы взаимосвязи информации и ПрК ТКУ ее обработки (рисунок 1.1).

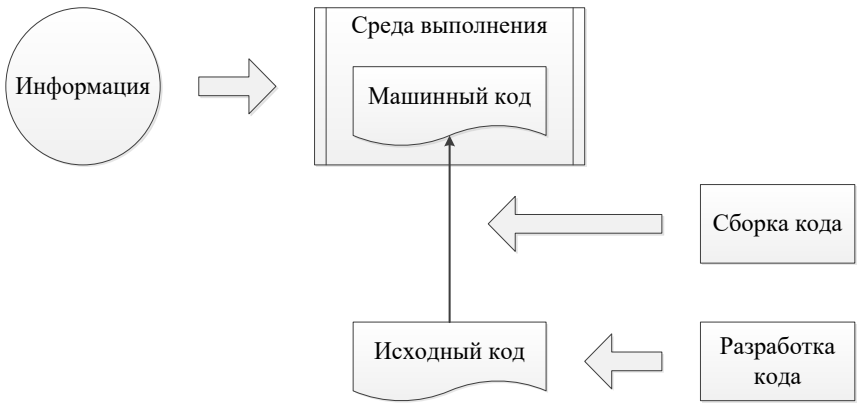

Рисунок 1.1 – Схема взаимосвязи информации, программного кода ее обработки и процесса создания кода

Согласно схеме, вначале происходит разработка ИК, затем его сборка в исполняемый МК; средой выполнения выступает аппаратное обеспечение (процес-

сор, память и т. п.) ТКУ, обрабатывающего информацию. Можно выделить следующие способы разрешения проблемы безопасности кода, связанные с соответствующими элементам схемы – это криптографическая защита информации, разработка безопасного ИК, сборка ИК в безопасный, поиск уязвимостей в ИК, поиск уязвимостей в исполняемом коде, и, наконец, создание безопасной среды выполнения. Рассмотрим границы возможностей способов, определяющих соответствующие проблемные вопросы [95].

Криптографическая защита информации

Криптографические методы защиты информации в связи с глубоко проработанной математической теорией вполне справляются со следующими задачами: невозможностью прочтения информации посторонними, невозможностью незаметного изменения информации, проверкой подлинности авторства и невозможностью отказа от него.

Существуют различные механизмы современной криптографии, подкрепленные целым набором соответствующих алгоритмов – DES [96], RSA [97], MD5 [98], ГОСТ Р 34.11-2012 [99], др. и протоколов – IPSec [100], Диффи-Хеллмана [101], Kerboros [102], SSL [103], др. Тем не менее, для защиты от угроз со стороны ПрК они подходят лишь частично. Так, часть информации обязательно должна быть расшифрована в ТКУ, поскольку она может влиять на дальнейший сетевой маршрут; часть же может быть попросту полностью заменена, что принципиально сделает невозможным проверку ее неизменности. Также криптографические протоколы реализуются алгоритмами посредством ПрК ТКУ, уязвимости в котором могут привести к их некорректной или небезопасной работе. Например, сознательное уменьшение злоумышленником длины общего секретного ключа через модификацию ПрК на любой из стадий разработки кода в разы повысит вероятность успешности атаки с целью раскрытия данных, хотя внешне алгоритм шифрования будет выглядеть работоспособным.

Перспективным направлением является применение гомоморфного шифрования [104], суть которого сводится к возможности операций над данными с кор-

ректно-получаемым результатом без их непосредственной расшифровки. Тем не менее, недостаточная развитость направления и крайне высокие требования к ресурсам аппаратного обеспечения таких криптосистем не позволяют на сегодняшний день полноценно использовать данный подход.

Разработка безопасного исходного кода

Идея разработки ПрК изначально без уязвимостей не только выглядит как принципиально неосуществимая, но и даже она не смогла бы стать разрешением проблемы. Например, созданный подобным образом ПрК ТКУ обеспечивал бы наивысшую степень безопасности информации, но лишь до момента его изменения злоумышленником на любой из следующих стадий жизненного цикла ПрК: в процессе сборки, загрузки в ТКУ или уже в момент его выполнения. Также потенциальным способом внесения уязвимостей является использование утилит сборки (компиляторов, ассемблеров, линкеров и т. п.), содержащих ошибки или вредоносный код [105].

Приведенная идея безопасного ИК может являться лишь направлением теоретического исследования, отдельные ответвления которого возможно дадут практические результаты. Это обосновывается хотя бы тем, что сама уязвимость является крайне субъективной «чертой» ПрК, которая может менять свой статус контекстно от безопасной к небезопасной: со временем, в зависимости от местоположения или условий функционирования ТКУ. Здесь следует отметить, как минимум, две интересные с научной точки зрения ветви: это исследование механизмов понижения человеческого фактора в появлении уязвимостей и разработка парадигм программирования, концепция которых исключала или сильно снижала бы возможность такого появления.

Сборка исходного кода в безопасный

Алгоритмы обнаружения уязвимостей уже сейчас встраиваются многими производителями в утилиты сборки кода; однако они позволяют обнаружить и нейтрализовать (для ряда языков программирования) лишь основные уязвимости

[106-107]. При этом такой способ не лишен упомянутого недостатка предыдущих: возможность модификации исполняемого кода и субъективность суждения относительно некоторых видов уязвимостей.

Поиск уязвимостей в исходном коде

Одним из реалистичных вариантов нейтрализации уязвимостей в ПрК ТКУ является их поиск по ИК в процессе разработки. При этом существует достаточное количество программных средств, производящих такой поиск автоматически [108]. Ручной способ поиска имеет адекватные трудозатраты, поскольку форма кода для поиска совпадает с той, что использовалась при его написании – программно-языковой, понятной программисту.

Основной сложностью применения способа считается полное отсутствие ИК для большинства ТКУ. Достаточно часто производители устройств являются транснациональными корпорациями (такие, как Cisco) или расположены в других странах (такие, как Huawei и Juniper). Вследствие этого производители не предоставляют какой-либо информации о коде реализации или деталях работы алгоритмов устройств, опираясь на множество причин: коммерческих, политических, географических или иных. По сути, это является использованием ТКУ в форме «черного ящика» на свой страх и риск. Также у способа есть недостатки, присущие и способу разработки кода без уязвимостей: последние могут быть внесены на последующих стадиях жизненного цикла кода.

Поиск уязвимостей в исполняемом коде

Другим альтернативным вариантом противодействия уязвимостям в ПрК ТКУ путем их поиска является проверка кода уже на этапе его непосредственного выполнения. Это, во-первых, позволяет «работать» с кодом уже в окончательном виде, после которого никакие изменения с ним не должны быть произведены. Во-вторых, даже если злоумышленник произведет модификацию кода с его обратной загрузкой в ТКУ (официальным образом через обновление или ручным при поставке и настройке устройства), то внеплановая проверка позволит обнаружить

внесенные уязвимости. И, в-третьих, плановые проверки ПрК ТКУ позволят иметь данные о состоянии его безопасности, что также не маловажно для аудита информационной системы в целом.

Хотя поиск уязвимостей без наличия ИК считается алгоритмически трудной задачей, уже существуют отдельные немногочисленные решения [109], целью которых является перевод исполняемого кода в исходный с последующим поиском уязвимостей в нем рассмотренным ранее способом.

Создание безопасной среды выполнения

Поскольку среда выполнения является связующим звеном между информацией – целью угроз, и ПрК – уязвимым местом для возникновения угроз, то возможно наличие способов защиты именно в ней. Средами, реализующими некоторые такие возможности, являются виртуальные машины (наиболее известные из них Java Virtual Machine [110] и Common Language Runtime [111]), назначение которых сводится к выполнению в собственном пространстве памяти программ, имеющих вид байт-кодов. Безопасность в данном случае обеспечивается полным контролем всех действий, исполняемых программой, и некоторыми особенностями языков программирования, препятствующих созданию типичных ошибок программирования (например, переполнения буфера). При этом, такой байт-код может быть восстановлен в исходный (такими утилитами, как JD-GUI [112] для Java и .NET Reflector [113] для C#).

Тем не менее, низкая производительность таких систем пока не допускает их применение в ТКУ. Также безопасность обеспечивается лишь для узкого круга уязвимостей, как правило, вносимых в код случайно. При этом, сама концепция распространения своих продуктов с открытым или восстановимым кодом будет крайне негативно оценена производителями ТКУ и его ПрК следуя целому ряду причин: авторским правам, экономическим и политические интересам и т. п. В подтверждение этого можно отметить практическое отсутствие байт-кода в ТКУ большинства мировых производителей.

1.1.2 Особенности уязвимостей в исполняемом коде

Согласно рассмотренным в пункте 1.1.1 способам нейтрализации уязвимостей в ТКУ невозможно однозначно сказать, какой из них мог бы быть наиболее перспективным разрешением проблемы небезопасности ПрК. Однако можно утверждать, что способы криптографической защиты и защиты информации с помощью среды выполнения хотя и эффективно решают собственную область задач, но не применимы, как общие решения. Таким образом, можно обоснованно предположить, что решения стоит искать именно в областях разработки и поиска уязвимостей в ПрК. При этом разработка ПрК без уязвимостей является все же достаточно утопическим решением, а проверка уязвимостей в процессе сборки имеет недостаточную практическую ценность. Особенность же современных ТКУ, такая, как отсутствие ИК, не позволяет искать уязвимости именно в нем. Суммируя вышесказанное, можно утверждать, что наиболее востребованным среди способов нейтрализации уязвимостей является их поиск в исполняемом коде, который в случае ТКУ является МК. Тем не менее, для более глубокого понимания особенностей такого поиска (как в принципе и любого подобного) необходимо изучить свойства самого целевого объекта поиска – уязвимости; при этом исследованию должны быть подвергнуты не только ее статические, но и динамические свойства – определяющие жизненный цикл уязвимостей. Во-первых, динамические свойства покажут причины возникновения уязвимостей в исполняемом коде. И, во-вторых, анализ свойств позволит более широко рассмотреть все возможные особенности поиска, поскольку последние напрямую зависят от того, какой вид имеет уязвимость. Статические же свойства дадут лишь удобную классификацию уязвимостей и частично сформируют дополнительные знания о них.

Свойства уязвимостей машинного кода

Рассмотрим основные статические свойства уязвимости с позиции таксономии. Известно значительное количество способов деления уязвимостей на различные типы и классы (в том числе и специализированных [114]). Они могут за-

висеть от рода ошибок в программе, их вызывающих, величины возможного ущерба, сложности поиска, быть строго заданы в CVE [115], NVD [116] и OSVDB базах и т. п. Тем не менее, в рамках выбранного проблемного вопроса интерес представляет деление уязвимостей по их структурному уровню в коде. Такая типизация позволит разделять уязвимости в соответствии с этапами разработки ПрК и алгоритмами их поиска. Произведем деление уязвимостей на 3 следующих типа.

Во-первых, это низкоуровневые уязвимости (далее – НУ), такие как ошибки в вычислениях, структурах данных, доступе к ним и т. п. Наличие их возможно на последних этапах жизненного цикла ПО – в процессе непосредственного кодирования, а поиск может быть формализованным по шаблонам. Типичным примером уязвимости может быть переполнение массива с перезаписыванием полезных данных, расположенных в памяти за ним [117].

Во-вторых, это среднеуровневые уязвимости (далее – СУ), такие, как неверная реализация алгоритма подпрограммы, передачи входных параметров, возврата из нее и т. п. Они, как правило, возникают на этапах проектирования и написания логики ПрК; поиск уязвимостей осуществляется ручным способом с применением автоматизирующих средств. Примером может быть ошибка в условии для вызова подпрограммы.

И, в-третьих, это высокоуровневые уязвимости (далее – ВУ), такие, как ошибки в архитектуре программной системы или концепции, нарушение общих принципов ее функционирования, безопасности и т. п. Такой тип уязвимостей характерен для начальных этапов разработки, когда еще какая-либо строгая формализация отсутствует, что соответствующим образом усложняет их поиск, осуществимый практически только ручным способом [66, 82]. Примером может быть использование слабых алгоритмов шифрования или возможность компрометации секретных ключей.

Динамические свойства уязвимости задают весь ее *жизненный путь* – от возникновения до обнаружения (после которого нейтрализация будет лишь технической задачей). Так как уязвимость существует только внутри ПрК и его прообразах, то и ее вид должен определяться соответствующим состоянием ПО (со-

стоящим помимо собственного ПрК так же и из описания его моделей, архитектур, документации и т. п.) в соответствующей точке жизненного цикла. Далее будем называть состояние ПО с заданными формой и содержанием – *Представлением ПО*. Так, например, состояние кода в момент его ассемблирования будет называться Представлением ассемблерного кода (далее – АК), имеющим текстовую форму и содержащим инструкции процессора выполнения. Соответственно, в процессе разработки ПО оно переходит из одних Представлений в другие. При этом, уязвимости могут возникать в одних Представлениях ПО, мутировать при переходе к другим и быть обнаружены в третьих.

Точка возникновения уязвимости может быть более чем в половине состояний ПО – в момент создания нового Представления; точка же ее обнаружения присутствует практически в каждом Представлении и зависит от наличия соответствующих способов поиска уязвимостей. Если способы отсутствуют, то она «живет» во всех Представлениях ПО после возникновения. Путь жизни уязвимости, как правило, является линейным однонаправленным (т. е. вектором), однако существуют следующие исключения. Во-первых, Представления могут переходить от более позднего к раннему в результате их восстановления (например, для получения АК применяется дизассемблирование машинного), и, следовательно, такой путь будет иметь обратное направление. Во-вторых, процесс восстановления может получить подобное, но все же отличное от предыдущего, Представление (например, декомпиляция на практике восстанавливает не ИК, а его подобие), и, следовательно, путь будет иметь разветвление.

Поскольку путь каждой отдельной уязвимости рассматривать не представляется возможным, то целесообразно их объединить и исследовать все доступные пути в комплексе. Множество таких объединенных путей будем называть – *областью жизни* уязвимостей. Типизация уязвимостей позволит построить области жизни каждого из типов, что даст более глубокие и детализированные знания об их свойствах.

Известные способы поиска уязвимостей

Применение поиска по МК со 100%-ным обнаружением всех уязвимостей в нем позволила бы решить проблему безопасного кода практически полностью. Однако такой поиск существует лишь в ограниченном числе случаев – например, для простых уязвимостей, поскольку поиск сложных плохо автоматизируем, требует неоправданно большого количества времени и высокой квалификации пользователя со специальными навыками. Последний является экспертом по уязвимостям и подходам к их поиску в МК (далее – Эксперт-ПУ).

Обзор возможных подходов к анализу МК, а именно – ручного и автоматического, статического и динамического, реверс-инжиниринга [118], и их комбинирование позволили выделить следующие способы поиска уязвимостей в машинном коде.

Способ 1. Статический ручной поиск уязвимостей в машинном коде применим для аналитических задач высокой сложности, поскольку он основан на использовании труда эксперта. Обратной стороной способа является то, что могут быть найдены лишь локализованные в коде уязвимости, анализ требует обработки огромного количества кода, необходимо детальное знание набора инструкций процессора.

Способ 2. Применение декомпиляции машинного кода позволяет проводить анализ получаемого представления, хорошо понятного эксперту, более успешно. Тем не менее, его не предназначенность для поиска высокоуровневых уязвимостей, как и наличие излишней информации для поиска среднеуровневых можно считать ограничениями в использовании.

Способ 3. Автоматизация статического способа тождественна применению программных средств, осуществляющих поиск уязвимостей по шаблонам или эвристическим правилам. Наиболее известными средствами является антивирусное ПО [119]. Способ хотя и позволяет обрабатывать весь код целиком за короткое время, тем не менее, он подходит исключительно для поиска низкоуровневых уязвимостей, достаточно формализуемых. Отдельной задачей является разработка самих шаблонов для поиска, осуществляемая в основном вручную.

Способ 4. Альтернативой статических способов считаются динамические, представляющие собой анализ не самого кода, а результатов его выполнения (на эмуляторе или реальном процессоре). В этом случае уязвимости обнаруживаются не напрямую, а по их влиянию на «нормальное» выполнение кода. Применение скриптов для автоматизации процесса частично повышает скорость выполнения заданных сценариев. От эксперта требуются особые навыки, заключающиеся по крайне мере в построении мысленной динамической модели поведения кода и работе с ней.

Способ 5. Полная автоматизация динамического способа может быть произведена путем сбора результатов выполнения кода для большого множества входных данных (как при запуске, так и в процессе). Ускорение процесса осуществимо путем оптимальной выборки набора данных. Способ носит название – «фаззинг тестирование» (от англ. fuzzing или fuzz testing). Результатом работы, как правило, является отчет о программных исключениях кода, вызванных определенными условиями. Задача эксперта заключается в создании набора входных данных и анализе полученных отчетов. Поиск средне- и высокоуровневых уязвимостей практически невозможен по причине сложности подбора условий их задействования.

Критерии эффективности способов поиска

Для выбора способа из существующих или постановки требований к новому необходимы критерии, определяющие его эффективность. Наибольший интерес представляют подходящие для поиска СУ и ВУ в условиях наличия только МК. Поскольку какие-либо количественные требования к эффективности трудноопределимы, введем следующие качественные критерии.

Критерии 1 и 2. Исходя из того, что поиск достаточно формализуемых НУ является технически решаемой задачей, существует определенное количество средств их поиска. Остальные же оставлены практически без внимания, что приводит к установлению первых двух критериев – применимость для поиска СУ и

ВУ. Особенностью уязвимостей является то, что их обнаружение возможно только при помощи субъективной оценки эксперта.

Критерий 3. Теоретическая возможность обнаружения любого типа уязвимостей ручным способом при наличии неограниченно количества времени означает, тем не менее, его малую эффективность, устанавливая третий критерий – автоматизация поиска для уменьшения затрачиваемого времени. Естественно, автоматизации должны быть подвержены лишь такие шаги способа, которые не зависят от экспертной оценки.

Критерий 4. Высокая субъективность понятия уязвимости, в особенности с повышением их структурного уровня, неизменно будет приводить к увеличению ложных обнаружений. Это потребует дополнительных проверок, уменьшающих эффективность поиска. Поэтому четвертым критерием введена достоверность получаемых результатов, очевидно низкая в случае полной автоматизации без применения труда эксперта.

Критерий 5. Достаточно большое количество типов процессоров выполнения, а также ограниченность средств их поддержки (анализаторов и эмуляторов машинного кода, штата специалистов и т. п.), означает очевидное преимущество поиска, основные алгоритмы (и их реализации) которого работают с инвариантным представлением данных – независимым от процессора; это считается пятым критерием.

Критерий 6. Интенсивные потоки разнородной информации приводят к большим объемам ПрК для реализации систем ее обработки. Возможность анализа МК большого размера на предмет наличия уязвимостей определяет шестой критерий; он не тождественен автоматизации и не противоположен достоверности. Так, при увеличении объема данных автоматизация способа может не давать желаемого эффекта (например, если поиск не охватывает весь объем МК), а применение труда экспертов может не снижать заметно скорость поиска уязвимостей (например, если анализируемое представление достаточно компактно).

Критерий 7. Седьмым критерием будем считать достаточно редко реализуемую, но крайне востребованную особенность поиска уязвимостей в виде адапта-

ции его процесса под требования эксперта, которая позволит соптимизировать работы последнего.

Критерий 8. Присутствие в данных, по которым осуществляется поиск, не используемой информации тем или иным образом будет снижать его эффективность. Поэтому критерий минимизации излишней для анализа информации является существенным.

Критерий 9. Предоставление дополнительной информация об уязвимостях потенциально поможет их обнаружению, что является девятым критерием.

Критерий 10. Немаловажным можно считать и сложность применения способа, влияющая на требования к квалификации эксперта, его использующего. Так, например, высокая сложность как приведет к появлению ошибок – из-за сильного влияния человеческого фактора, так и не позволит распараллелить поиск уязвимостей среди нескольких экспертов – их с таким уровнем квалификации попросту может не оказаться в наличие.

Введенные критерии можно считать независимыми друг от друга – т. е. удовлетворение способа по одним не приводит к недостижимости по другим – что позволяет их оценивать по-отдельности и без учета взаимного влияния. Таким образом, итоговая оценка способа может быть получена экспертно-балльным методом. В качестве баллов достаточно использовать минимально-необходимую шкалу из 3-х значений, отражающую удовлетворительность по критерию. Очевидность возможностей описанных способов поиска, четкость введенных критериев и тривиальность логики определения удовлетворительности позволяют свести инструментарий оценки до объективного начисления баллов и их простого суммирования.

Результаты сравнительной оценки способов поиска уязвимостей по введенным критериям приведены в таблице 1.1. В качестве оценки выполнения критериев используются следующие обозначения: «+» – удовлетворяет полностью, «–» – не удовлетворяет, «+/–» – удовлетворяет частично.

Таблица 1.1 – Критериальная оценка известных способов поиска уязвимостей в МК

Критерий	**Способы**				
	1. Статический ручной	**2. Статический ручной с декомпиляцией**	**3. Автоматический по шаблонам**	**4. Динамическое выполнение**	**5. Фаззинг тестирование**
1. Поиск СУ	+/–	+/–	–	+/–	–
2. Поиск ВУ	–	–	–	–	–
3. Автоматизация	–	+	+	+/–	+
4. Достоверность результатов	+	+	+/–	+/–	–
5. Инвариантность представления МК	–	+	+/–	+/–	+
6. Обработка больших объемов	–	+/–	+	–	–
7. Возможность адаптации процесса	+/–	–	–	+/–	–
8. Минимизация излишней информации	–	+/–	+	+/–	+
9. Дополнительная информация об уязвимостях	–	–	+/–	–	+/–
10. Не высокая сложность применения	–	+/–	+/–	–	+
Сумма баллов	2	5	5	3	4.5

Согласно результатам оценки, ни один из способов не является удовлетворительным. При этом наиболее близкий к оптимальному (имеющему сумму баллов 10) – статический ручной с декомпиляцией и автоматический по шаблонам – соответствуют всем критериям лишь наполовину (имеют по 5 баллов). Следовательно, создание нового способа, удовлетворяющего всем критериям, существенно – как минимум в 2 раза – повысит эффективность поиска СУ и ВУ в МК ТКУ, что будет считаться существенным повышением эффективности поиска. Разработка же методик оценки разработанного способа позволит производить его разностороннее сравнение, как с существующими, так и с будущими аналогами.

1.1.3 Области жизни уязвимостей в Представлениях программного обеспечения

Представления программного обеспечения

Рассмотрим более детально динамические свойства уязвимостей. Наложение областей жизни уязвимостей на схему Представлений ПО будем называть схемой областей жизни уязвимостей в Представлениях ПО. Анализ такой схемы позволит оценить текущее состояние вопросов поиска уязвимостей с точки зрения удовлетворительности средств их поиска и предложить перспективное направление исследования.

Для каждого Представления ПО, имеющего реальную практику применения, можно выделить следующие свойства. Во-первых, у каждого Представления есть свое назначение. Во-вторых, любое Представление обладает собственным уникальным содержанием, соответствующим этому назначению. В-третьих, вид содержания Представления имеет определенную форму, наиболее подходящую для соответствия назначению. В-четвертых, каждое Представление получается из предыдущего определенным способом. В-пятых, каждое Представление потенциально восстанавливаемо из последующего определенным способом. В-шестых, при получении нового Представления возможно появление одних уязвимостей, а в результате его анализа возможно обнаружение других. И, в-седьмых, для каждого Представления существуют только определенные способы поиска уязвимостей, зависящие от его формы и содержания. Возможные значения некоторых свойств приводятся далее.

Каждое из Представлений может принимать следующие формы (или их комбинацию): *словесную* – с помощью естественного языка; *графическую* – в виде блок-схем, диаграмм, изображений графов; *программно-языковую* – с помощью формально-знакового текста на языках программирования; *бинарную* – в виде набора байтов (реже битов).

Получение последующих Представлений из текущих осуществляется с помощью превращений, таких как: *синтез* – результат творческой деятельности человека с получением нового содержания (путем внесения в содержание предыду-

щего Представления внешнего, взятого из контекста текущего); *прямое преобразование* – результат работы специализированных программных средств (далее – ПС) без изменения содержания; *генерация* – результат работы специализированных ПС с изменением структуры содержания или метаинформации. Форма представления при этом меняется на соответствующее.

Также, возможно частичное или полное восстановление предыдущих Представлений из текущих с помощью превращений, таких как: *анализ* – выделение из Представления содержания восстанавливаемого (возможно, с элементами предсказания); *обратное преобразование* – результат работы специализированных ПС без изменения содержания; *парсинг* – результат работы специализированных ПС с изменением структуры содержания или метаинформации. Форма представления при этом меняется на соответствующее.

Единственное статическое свойство задает тип уязвимости, описанный ранее. Поиск уязвимостей может иметь как ручной тип – субъективно человеком, так и автоматический – специализированным ПС; естественно, возможно их объединение.

Приведем описания всех Представлений ПО, типичных для ТКУ. Вначале возникает мысленный прообраз программного продукта, отражающий его основную идею. Уязвимости в таком прообразе отсутствуют, поскольку именно он определяет требуемое состояние продукта; другими словами уязвимости являются сознательно задуманной часть прообраза. Затем разрабатывается концептуальная модель, вводящая основные понятия, их структуру и т. п. В этот момент возможно возникновение уязвимостей в виде ошибок в переложении идеи. По концептуальной модели разрабатывается архитектура кода, приближенная к будущей реализации. Ошибки в ее проектировании также приведут к наличию уязвимостей. Далее создаются алгоритмы работы функционала архитектуры. Процесс носит творческий характер, что ведет как к случайным, так и злонамеренным ошибкам. Полученные алгоритмы переписываются в виде операций на языке программирования, создавая ИК программы. При достаточной степени детализации алгоритмов и аккуратности выполнения кодирования случайных ошибок можно избе-

жать; однако человеческий фактор (в данном случае программиста) играет решающее значение. После этого ИК преобразуется в ассемблерный путем компиляции. Возникновение в результате ошибок носит лишь теоретический характер и связано с некорректной работой утилит сборки. Машинный код программы получается из ассемблерного тривиальным преобразованием текстового описания инструкций процессора в бинарную форму без каких-либо возможных ошибок. Для загрузки и развертывания на конечном ТКУ выполнения МК собирается в файл образа также без дополнительных уязвимостей.

Схема жизни уязвимостей в Представлениях программного обеспечения

Представления ПО и их свойства, включая относящиеся к уязвимостям, детально описаны в авторской статье [71]. В интересах разрешения проблемы в статье предлагается преобразование Представления АК в новое (а точнее восстановление путем анализа), хранящее необходимую и достаточную информацию для поиска уязвимостей [88]. Данное Представление будет в себе сочетать как архитектурные элементы ПО (содержащие ВУ), так и алгоритмические (содержащие в СУ), а также элементы их низкоуровневой реализации (содержащие НУ). Представления ПО, их свойства, уязвимости и способы их поиска представлены в сводной таблице 1.2.

В таблице 1.2 Представления 1-8 являются стандартными, а Представления А1-3 представляют собой новые, предлагаемые в интересах поиска уязвимостей в МК.

Согласование Представлений ПО, способов их получения и восстановления, возникновение и обнаружение различных типов уязвимостей, приведенных в таблице 1.2, позволяют получить следующую схему областей жизни уязвимостей в Представлениях ПО (рисунок 1.2).

Таблица 1.2 – Представления ПО, их свойства, уязвимости и способы их поиска

№ п\п	Название	Назначение	Содержание	Форма	Получение	Восстановление	Уязвимости		
							Возникновение	Обнаружение	Тип поиска
1	Основная идея	Мысленный прообраз ПО	Набор функционала, характеристик и свойств	Словесная		Анализ 2			
2	Концептуальная модель	Базовая смысловая структура и терминологический аппарат	Основные понятия, их структура, взаимосвязь, особенности	Словесно-графическая	Синтез из 1	Анализ 3 или А2	ВУ	ВУ	Ручной
3	Архитектура	Высокоуровневая модель ПО	Специфические особенности (технологии, языки), модули и элементы системы, взаимосвязи	Графическая (с комментариями)	Синтез из 2	Анализ 4	ВУ	ВУ	Ручной
4	Алгоритмы (ИК)	Алгоритмы подпрограмм (среднеуровневые)	Назначение подпрограмм и последовательности их шагов	Графическая (в виде блок-схем)	Синтез из 3	Анализ 5 или 5*	СУ	СУ	Ручной
4*	Алгоритмы (АК)	Алгоритмы подпрограмм (низкоуровневые)	Назначение подпрограмм и последовательности их шагов	Графическая (в виде блок-схем)		Анализ 6		НУ, СУ	Ручной
4**	Алгоритмы (АК) – модифицированные	Алгоритмы подпрограмм (низкоуровневые) с внесенными уязвимостями	Назначение подпрограмм и последовательности их шагов	Графическая (в виде блок-схем)	Синтез из 4*		НУ, СУ		
5	ИК (оригинальный)	Прямая реализация алгоритмов	Элементы языка программирования	Программно-языковая	Синтез из 4	Нет	НУ, СУ	НУ, СУ	Ручной, автоматический
5*	ИК (псевдо)	Обратная реализация алгоритмов	Элементы языка программирования (псевдо)	Программно-языковая		Анализ 6		НУ, СУ	Ручной

№ п\п	Название	Назначение	Содержание	Форма	Получение	Восстановление	Уязвимости		
							Возникновение	Обнаружение	Тип поиска
6	АК	Промежуточная низкоуровневая реализация ПО	Элементы языка ассемблера	Программно-языковая (язык ассемблер)	Генерация по 5, прямое преобразование 4**	Парсинг 7	НУ	НУ	Ручной, автоматический
7	МК	Конечный код ПО	Набор инструкция процессора	Бинарная	Генерация по 6	Обратное преобразование 8	НУ	НУ	Аналогично 6
8	Файл образа	Образ для загрузки в устройство	Контейнер с набором инструкций процессора	Бинарная	Прямое преобразование 7		НУ	НУ	Аналогично 7
A1	Расширенная алгоритмизация АК	Промежуточное внутреннее описание архитектуры, модулей, алгоритмов и элементов их реализации	Элементы и служебная информация внутреннего представления (таблицы, графы, хэши и т. п.)	Бинарная		Анализ 6			
A2	Алгоритмизированный псевдо-исходный код	Алгоритмы и частичная архитектура ПО	Элементы языка описания алгоритмов и архитектуры	Программно-языковая, графическая (в виде блок-схем)		Анализ A1		СУ, ВУ	Автоматизированный
A3	Список шаблонных уязвимостей	Результаты поиска уязвимостей по шаблонам	Информация о найденных уязвимостях в формализованном виде	Программно-языковая (в спец. формате)		Анализ A1		НУ, СУ	Автоматический

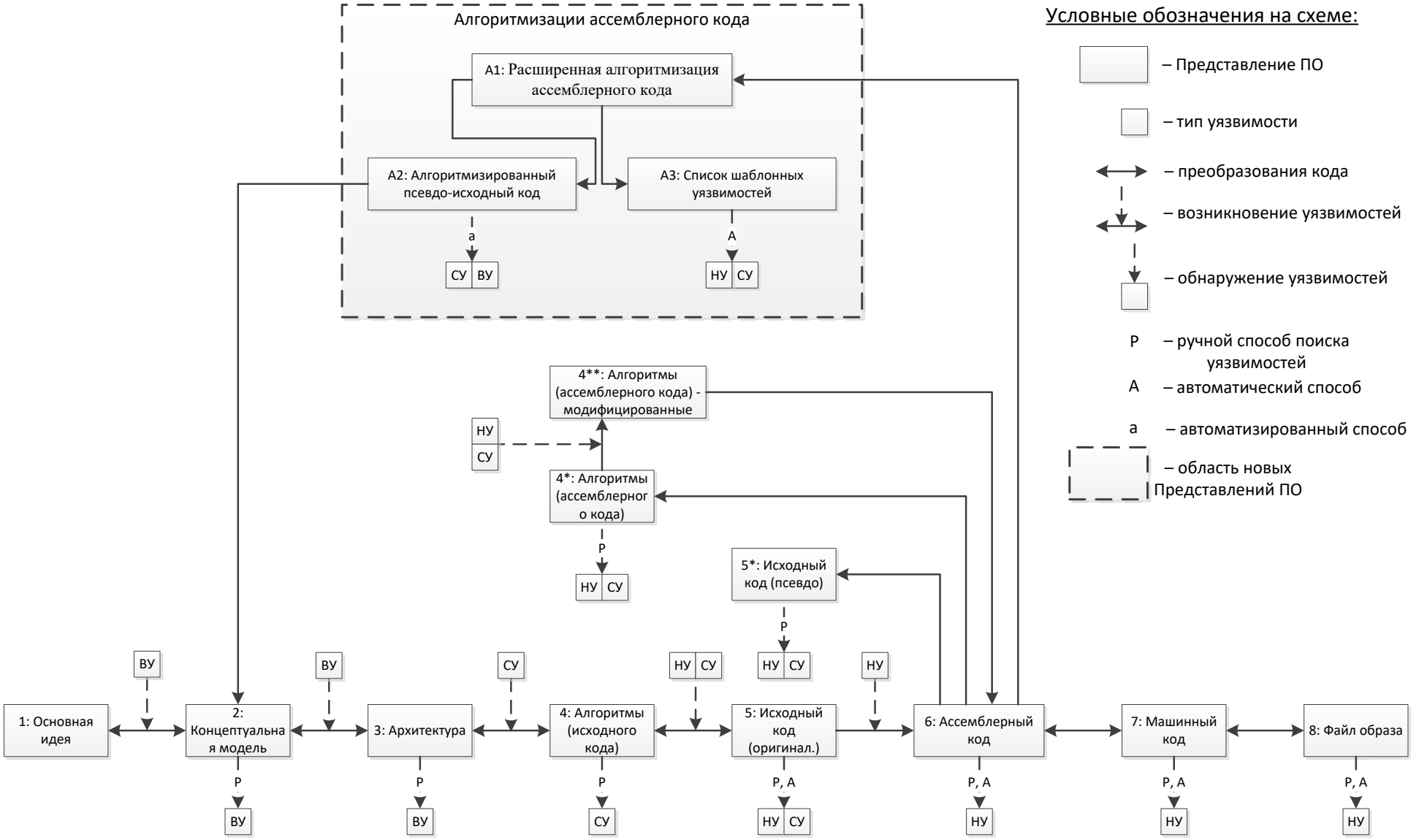

Рисунок 1.2 – Схема областей жизни уязвимостей в Представлениях ПО

Структурные метаданные машинного кода

Рассматривая существующие способы решения выбранного проблемного вопроса – поиска уязвимостей в МК, необходимо отметить, что для поиска НУ уже существует достаточное количество автоматических средств. Причина этого заключается в достаточной формализации уязвимостей данного типа и большой накопленной опытной базе. Так, например, использование неинициализированной переменной определимо автоматически по анализу графов потока данных и управления. Поиск же СУ и ВУ плохо сводим к автоматизации, поскольку большинство таких уязвимостей носят субъективный характер, формализация правил которого является не решенной задачей. Так, например, слабый алгоритм генератора псевдослучайных чисел может быть обусловлен требованиями к производительности кода и наличием дополнительных механизмом защиты, а не модификацией злоумышленника.

Используя схему областей жизни уязвимостей наиболее перспективным способом поиска СУ и ВУ можно считать следующий. Во-первых, производится однозначное преобразование МК в АК. Во-вторых, по АК создается бинарное Представление А1, которое можно считать промежуточным. В-третьих, Представление А1 преобразуется к А2 – алгоритмизированному Представлению, подходящему для анализа пользователем со специальными навыками. Последний является экспертом по безопасности ПрК на предмет наличия в нем уязвимостей (далее – Эксперт-БК). Его знания в МК (который, при этом, различен для каждого процессора выполнения) могут ограничиваться лишь общими представлениями. Таким образом, Эксперт-БК считается Экспертом-ПУ, использующим для поиска уязвимостей алгоритмизированное Представление. И, в-четвертых, Эксперт-БК производит поиск СУ и ВУ соответствующими методами и средствами.

Опишем более детально смысл введенного понятия *алгоритмизированности.* Будем под ним понимать представление организации и функционирования программной системы МК в виде согласованной структуры архитектуры и алгоритмов ПрК. Для определения такой структуры необходимо выделение из МК данных специального типа (этимологически приводящих к появлению приставки

мета-), при этом не хранящихся напрямую в нем. Такие данные, названные *структурными метаданными* (далее – СМД), состоят из следующих: модули и их взаимодействие, задающие архитектуру ПрК; взаимное использование подпрограмм и их данных, задающее строение модулей; алгоритмы и сигнатуры (имя, входные и выходные параметры), задающие функционал подпрограмм, управляющие структуры и логика их потока управления, задающие выполнение алгоритма. Иерархия элементов СМД может быть представлена в следующем табличном виде (таблица 1.3).

Таблица 1.3 – Иерархия элементов структурных метаданных

Родительский элемент	**Дочерние элементы**	**Связь дочерних элементов**
Архитектура	Модули	Взаимодействие модулей
Модуль	Подпрограммы, данные	Вызовы подпрограмм и использование ими разделяемых данных
Подпрограмма	Сигнатура (имя, входные/выходные параметры), алгоритм	Использование входных параметров алгоритмом для вычисления выходных
Алгоритм	Управляющие структуры	Логика потока управления

Согласно введенному понятию, *алгоритмизацией* будем считать процесс, задача которого заключается в восстановлении СМД, содержащихся в МК; в результате будет получено алгоритмизированного представления МК. Последнее позволит Эксперту-БК понять архитектуру и логику работы ПрК в интересах последующего поиска уязвимостей в нем.

1.2 Идентификация структурных метаданных типичных парадигм разработки программного кода телекоммуникационных устройств

Поскольку СМД отражают архитектуру и алгоритмы работы ПО (сквозную или проходящую через все Представления) и частично определяют его *содержание*, а их вид (словесный, графический, программно-языковый, бинарный) определяется лишь *формой*, то очевидно, что присутствие метаданных в той или иной

степени может быть найдено в любом Представлении ПО. Таким образом, весь процесс алгоритмизации сводится к преобразованию ПрК в такую форму, в которой вид СМД будет подходящим для обработки и поиска уязвимостей (ручным или автоматическим способом) [83]. Так, вид МК абсолютно не применим для поиска СУ и ВУ, а восстановление идеи или концептуальной модели практически невозможно.

В качестве предпосылок к определению свойств метаданных и их последующей идентификации рассмотрим следующие основные парадигмы программирования, типичные при разработке ПО ТКУ: модульную, процедурную, структурную и императивную [120-121]. Отметим, что наиболее распространенным языком для ПрК ТКУ является С, при разработке программ на котором эти парадигмы применяются совместно.

1.2.1 Модульная парадигма программирования

Суть модульной парадигмы программирования заключается в разделении ПрК на отдельные сущности – модули или логически взаимосвязанные совокупности функциональных элементов. На физическом уровне ПрК это может выглядеть, как его деление на отдельные файлы или их директории, реализующие строго выделенный функционал. По сути, модуль является более абстрагированной и сложной подпрограммой кода, поскольку предоставляет возможность, как многократного использования, так и своей замены без изменения всей системы. Использование парадигмы позволяет упорядочивать элементы ПрК и скрывать в них данные, закрытых для внешнего доступа.

Исходя из описанных особенностей парадигмы, части МК, собранные из ИК отдельных файлов или их групп в директориях, в той или иной степени должны быть обособлены и в едином образе МК. Это является основным следствием назначения модулей – способа физического и логического деления всей совокупности ПрК. При этом, исходя из назначения каждого модуля в качестве реализации обособленного функционала, все внешнее взаимодействие с ним должно про-

исходить по выделенному каналу вызовов – через его интерфейс. Таким образом, возможно выделение модулей в МК и их взаимосвязи, являющихся элементами СМД. Пример СМД, создаваемых модульной парадигмой программирования в МК, приведен на рисунке 1.3.

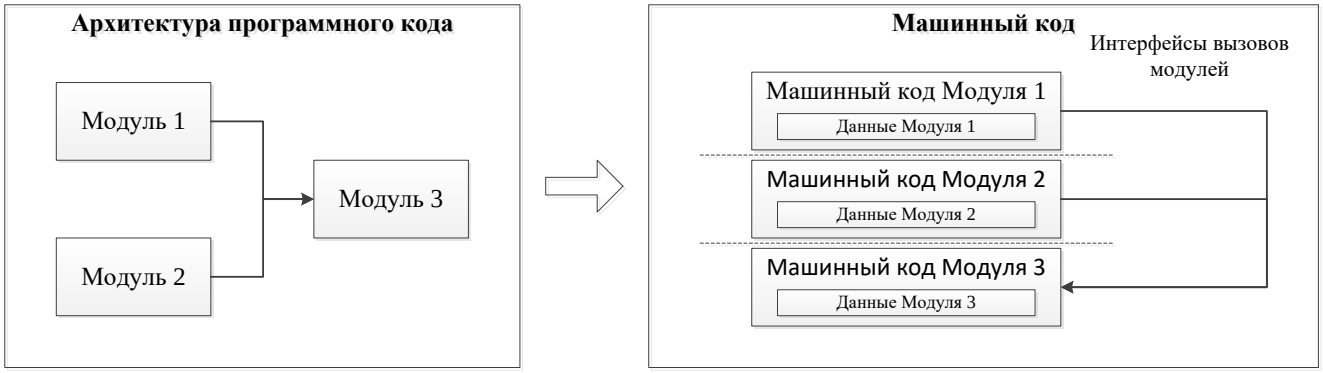

Рисунок 1.3 – Пример структурных метаданных модульной парадигмы программирования

Согласно рисунку, иерархическое деление архитектуры на отдельные модули найдет отражение и в линейной структуре МК. При этом возможно выделение, как самих модулей, так и схемы их взаимосвязей через интерфейсы.

1.2.2 Процедурная парадигма программирования

Суть процедурной парадигмы заключается в сборе многократно выполняемых последовательностей операций в подпрограммы, поименованные или иным образом идентифицированные. В языках высокого уровня используется два типа подпрограмм: процедура и функция (хотя зачастую второе название употребляется вместо первого). Принципиальным отличием второго типа от первого является обязательное наличие возвращаемого значения. Использование парадигмы программирования позволяет как произвести оптимизацию кода по занимаемому им объему (за счет однократной записи многократно выполняемых операций), так и повысить его понимание путем структуризации (за счёт использования более абстрактных и поименованных элементов). Также, разбиение на подпрограммы

предоставляет возможность программисту делить решаемую задачу на шаги, реализуя каждый из них по отдельности и практически независимо. Алгоритм в данном случае является телом подпрограммы и не имеет параметров.

Исходя из описанных особенностей парадигмы, облик подпрограммы ИК должен оставаться также и в МК, поскольку подпрограмма относится к определяющему элементу сути (формально, топологии) алгоритма. С этой точки зрения в процессе получения МК из ИК суть алгоритмов остается неизменной, а меняется лишь их форма – с программно-языковой на бинарную, а также детали реализации – с понятных человеку на воспринимаемые выполняющим устройством. При этом, исходя из особенностей назначения подпрограммы, как многократно-используемого кода, она должен иметь в МК одну точку входа (с существующими переходами на нее извне) и, как правило, одну точку выхода в конце собственного кода. Также очевидно, что подпрограммы могут быть вызваны из других таких же подпрограмм, анализ графа вызовов которых позволит восстановить логику их взаимодействия. Таким образом, возможно определение подпрограмм в МК и их взаимодействия, являющихся элементами СМД.

Пример СМД, создаваемых процедурной парадигмой программирования в МК, приведен на рисунке 1.4.

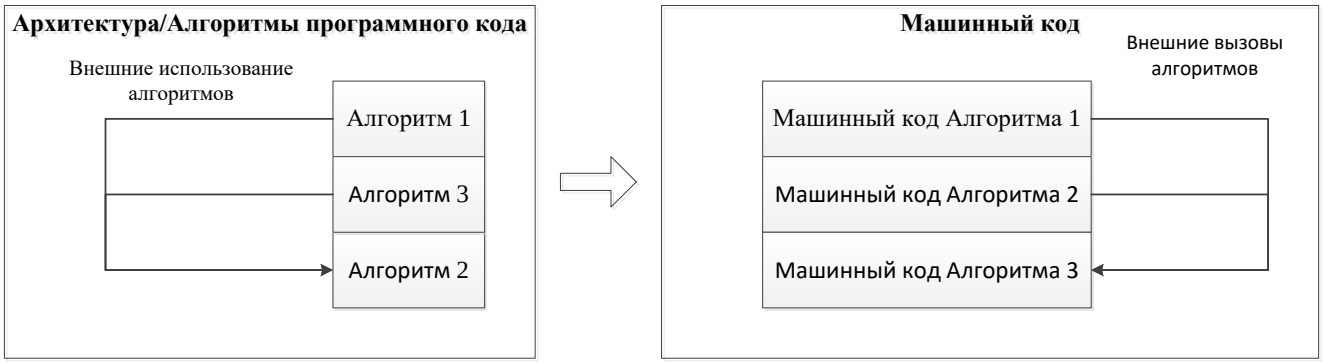

Рисунок 1.4 – Пример структурных метаданных процедурной парадигмы программирования

Согласно рисунку, деление ИК на отдельные подпрограммы/алгоритмы найдет отражение и в линейном МК. При этом возможно выделение, как самих подпрограмм/алгоритмов, так и их взаимодействия.

1.2.3 Структурная парадигма программирования

Сутью структурной парадигмы программирования является представление ПрК в виде иерархической структуры блоков. Последние соответствуют логически сгруппированным наборам идущих подряд операций. Использование парадигмы приводит к разработке кода при отсутствии оператора безусловного перехода GOTO; впрочем, многие языки допускают использование этого оператора в крайних случаях. В соответствии с данной методологией, любой ПрК строится с использованием трех базовых управляющих структур (далее – УС): последовательности, ветвления (IF) и цикла (FOR). Естественно, отдельные части кода могут объединяться в подпрограммы в соответствии с процедурной парадигмой программирования.

Одним из существующих развитий парадигмы является так называемое двумерное графическое программирование (ярчайшим представителем которого можно считать язык ДРАКОН [122], в котором ключевые слова заменяются на управляющую графику.

Использование парадигмы позволяет упростить разработку кода (а, следовательно, его понимание, оптимизацию, отладку) для решения сложных задач. Код в такой парадигме достаточно хорошо представляем в графическом виде, например, с помощью диаграмм Насси-Шнейдермана [59].

Исходя из описанных особенностей парадигмы, все УС должны найти свое отражение и в МК, поскольку они, как и подпрограммы, определяют суть или содержание алгоритма, которая остается неизменна при переходе от программно-языковой формы к бинарной. Так, в преобладающем количестве наборов инструкций процессоров существуют аналогичные им сущности: последовательно-выполняемые операции, оператор условного перехода (единственная доступная форма ветвления в языках ассемблер, например JNE/JNZ для Intel и BC/BCA для PowerPC [123]) и операторы цикла (например, LOOP для Intel). Также, в МК оператор безусловного перехода GOTO имеет свое точное отражение (например, JMP для Intel и B/BA для PowerPC), равно как и оператор вызова подпрограмм (напри-

мер, CALL/RET для Intel и B/BLR для PowerPC). Необходимо отметить, что не во всех наборах инструкций процессоров присутствует оператор цикла (например, для PowerPC его нет); в таких случаях, эта УС организуется с помощью оператора условного перехода, образуя цикл на графе потока управления кода, который определяем аналитически. Таким образом, возможно определение УС в МК и построение по ним логики алгоритма, являющихся элементами СМД.

Пример СМД, создаваемых структурной парадигмой программирования в МК, приведен на рисунке 1.5.

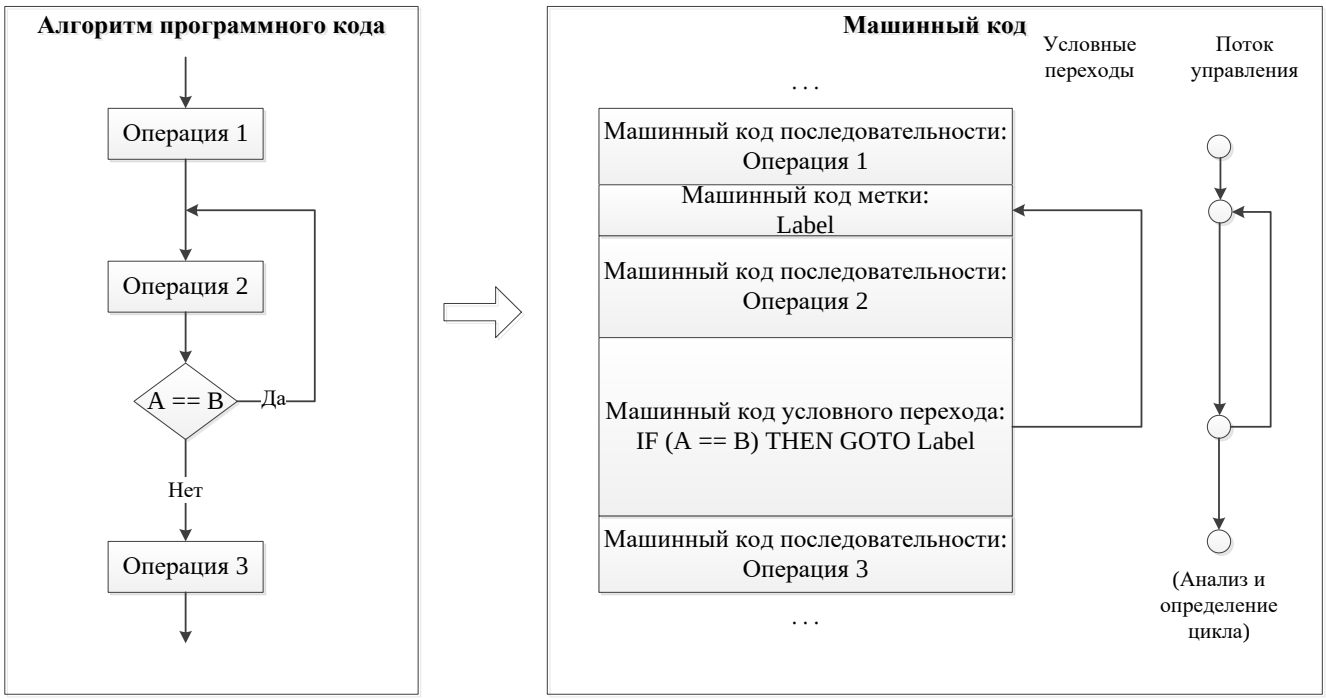

Рисунок 1.5 – Пример структурных метаданных структурной парадигмы программирования

Согласно рисунку 1.5, после преобразования алгоритмов ПрК в форму МК, все УС сохраняются. Так, структуре ветвления «A==B» сопоставляется в МК соответствующий оператор условного перехода «IF (A == B) THEN GOTO» на метку «Label». Построив по МК граф потока управления, можно аналитически определить оператор цикла, присутствующий и в исходном алгоритме.

1.2.4 Императивная парадигма программирования

Сутью императивной парадигмы программирования является описание процесса вычислений в виде инструкций, изменяющих состояние памяти (в отличие от декларативной). При использовании парадигмы, очевидно, интенсивно используется операция присваивания (регистру, ячейке памяти). Также очевидно, что данная парадигма тесно переплетается со структурной и процедурными парадигмами; противопоставлением является функциональная парадигма, которая описывает процесс вычисления значений функций.

Данная парадигма лежит в основе развития программирования в целом, поскольку первым ее языком был МК и, соответственно, язык ассемблер. Это и определяет основное достоинство парадигмы – возможность написания низкоуровневых реализаций алгоритмов, что позволяет делать их максимально оптимизированными под конечный процессор выполнения.

Исходя из описанных особенностей парадигмы, никаких СМД напрямую в МК она не создает. Тем не менее, отдельные вычисления могут быть использованы как для уточнения метаданных структурной парадигмы – значений условий для переходов, так и процедурной – значений возвращаемых значений и их связей с параметрами функций. Таким образом, возможно определение входных и выходных параметров подпрограмм – т. е. частичной сигнатуры подпрограммы, являющейся элементом СМД.

Пример СМД, создаваемых императивной парадигмой программирования в МК, приведен на рисунке 1.6.

Согласно рисунку 1.6, хотя в МК никакие СМД не переходят, тем не менее, результат сравнения вычисленных значений переменных A и B используется для выбора ветки условного перехода – на метку Label при равенстве A и B. Такая информация может считаться дополнительной и также использоваться при алгоритмизации МК.

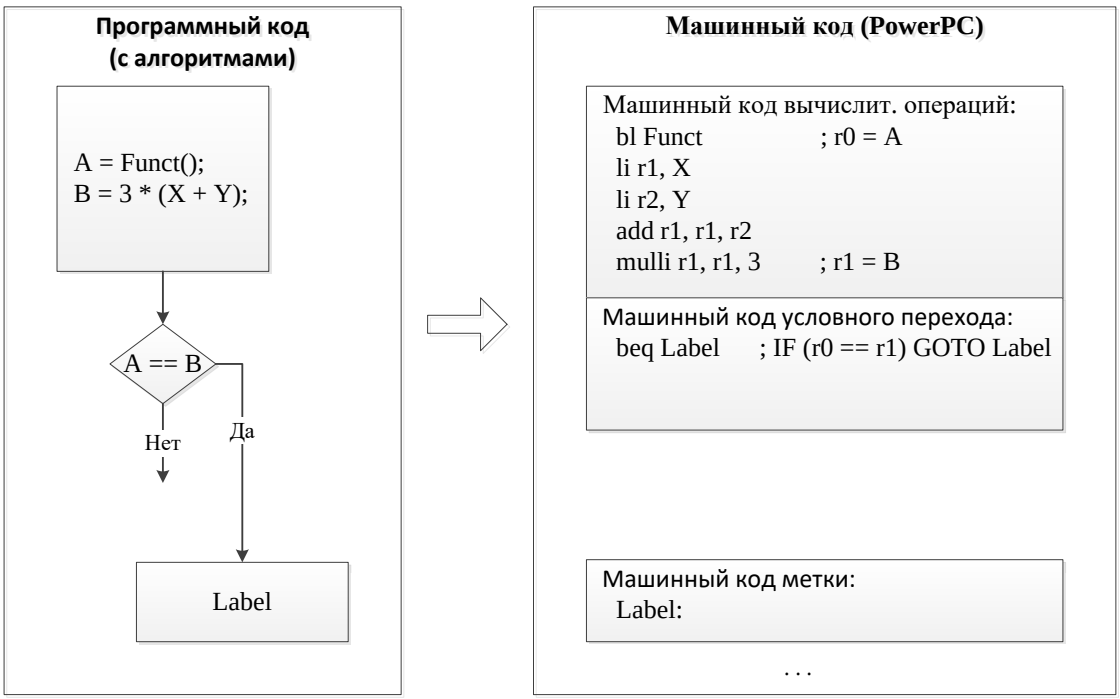

Рисунок 1.6 – Пример структурных метаданных, уточняемых императивной парадигмой программирования

1.3 Исследование моделей машинного кода с позиции структурных метаданных

Проведем анализ машинного кода с позиции его СМД, что позволит получить модель очерченной предметной области (далее – Модель), отражающая основные базисные элементы и их свойства, применимые для алгоритмизации.

Вначале выделим информацию, содержащуюся в МК, и определим возможности идентификации по ней СМД [124-125]. Согласно этому определим требования к Модели. Произведем анализ существующих моделей МК и в случае их отсутствия предложим собственную. Полученная таким образом модель может быть использования для осуществления алгоритмизации.

1.3.1 Низкоуровневая информация в машинном коде

МК имеет низшую степень абстрактности описания реализации ПО, поэтому восстановление ручным способом даже общего вида алгоритмов представляет

серьезную задачу. Тривиальное преобразование его в ассемблерное Представление частично упрощает задачу восстановления. Также – следуя из того, что вид МК есть лишь одна из форм ПрК, с сохраненным содержанием, и с учетом особенностей приведенных парадигм программирования – некоторая «полезная» низкоуровневая информация в нем все же присутствует. Определим такую информацию в АК, выделение которой может быть использована в интересах алгоритмизации [84].

Код выполнения

В любом МК существует область, содержащая код выполнения – те инструкции процессора, которые непосредственно реализуют действия ПрК и используются конечным устройством для выполнения алгоритмов [126]. В интересах алгоритмизации в АК, полученном из МК, может быть найдена следующая информация.

1) Деление на подпрограммы. АК содержит подпрограммы, которые собственно и содержат реализацию всех алгоритмов ПрК. Пример АК для процессора PowerPC с комментариями к инструкциям приведен ниже:

```
; пример подпрограммы-функции Funct
       .globl Funct              ; декларация переменной-подпрограммы Funct
       .type  Funct, @function   ; задание типа переменной Funct, как подпрограмма
Funct:                           ; метка начала (входа) подпрограммы
       mr r3, 0x1                ; занесение в регистр r3 значения 0x1
       blr                       ; возврат из подпрограммы значения в регистре r3
(по умолчанию)
```

Анализ в АК переходов на некий адрес блока инструкций (имеющий в тексте вид именованной метки), завершающихся инструкцией возврата из подпрограммы BLR, позволит выделить код подпрограмм (которые могут быть как процедурами, так и функциями). Точка такого перехода считается входом в подпрограмму и адресом ее первой инструкции. А все точки с инструкциями BLR – выходами из подпрограммы и ее последними, для каждой ветки выполнения, инструкциями. Отметим, что любые переходы между подпрограммами помимо этих точек считаются признаком кода, не соответствующего процедурной парадигме

программирования, и далее не рассматриваются, как редчайшие исключения (они могут быть результатом специальных межподпрограммных оптимизаций или ВУ).

2) Инструкции вычислений. АК содержит инструкции вычислений, которые производят изменение переменных и памяти согласно заданным выражениям (формулам). Пример АК для процессора PowerPC с комментариями к инструкциям приведен ниже:

```
; пример вычисления формулы r3 * (r1 + r2) + r3
      add r1, r1, r2    ; сложение регистр r1 с регистром r2 и занесение результата в r1
      mulli r1, r1, r3  ; умножение регистр r1 на регистр r3 и занесение результата в r1
      add r1, r1, r3    ; сложение регистра r1 с регистром r3 и занесение результата в r1
```

Подстановка констант, «сворачивание» и оптимизация уравнений, а также другие вычислительные операции могут быть использованы для получения дополнительной информации, такой как значения выбора для операторов условного перехода.

3) Инструкции безусловного перехода. АК содержит инструкции безусловного перехода, которые изменяют адрес следующей выполняемой инструкции на заданный независимо от каких-либо условий. Пример АК для процессора PowerPC с комментариями к инструкциям приведен ниже:

```
; пример перехода на метку Label
      b .Label                  ; безусловный переход на метку Label
      …                   ; пропущенные при переходе инструкции
.Label:                         ; метка Label
```

Безусловный переход не может присутствовать в графе потока управления, поскольку последний состоит из *базовых блоков* – прямолинейных участков без каких-либо входящих или выходящих путей, связанных друг с другом с помощью ветвления. Безусловные переходы, как правило, в АК появляются вследствие «разворачивания» УС ИК компилятором и, следовательно, при построении потока управления могут быть в них обратно «свернуты». Непосредственное использование операторов GOTO в ИК применяется редко (см. пункт 1.2.3), поскольку они всегда могут быть переписаны с помощью УС.

4) Инструкции условного перехода. АК содержит инструкции условного перехода, которые изменяют адрес следующей выполняемой инструкции на заданный в зависимости от условий (как правило, по результатам сравнения регистров, значений памяти и констант). Пример АК для процессора PowerPC с комментариями к инструкциям приведен ниже:

```
; пример основной части подпрограммы-функции сравнения двух чисел в регистрах r1 и r2
      cmpw r0, r1, r2     ; сравнение регистров r1 и r2 с занесением результата в r0
      bne r0, .Label_1    ; переход на метку Label_1, если результат сравнения «неравенство»
      li r3, 0x1          ; занесение в регистр r3 значения 0x1 (аналог true – правда)
      b .Label_2          ; безусловный переход на метку Label_2
.Label_1:                 ; метка Label_1
      li r3, 0x0          ; занесение в регистр r3 значения 0x0 (аналог false – неправда)
.Label_2:                 ; метка Label_2
      blr                 ; возврат из подпрограммы значения в регистре r3 (по умолчанию)
```

Анализ графа потока управления (условный переход для которого является способом соединения его базовых блоков) может восстановить вид алгоритмов подпрограмм. При этом, в отличие от безусловного перехода, такое восстановление даст больше метаинформации, поскольку инструкции напрямую соответствуют УС в их «естественном» виде. В частности, в вышеприведенном примере можно заметить появление оператора безусловного перехода (b .Label_2), который, как было упомянуто, как раз и завершает «сворачивание» УС ветвления (IF).

5) Инструкции цикла. АК содержит инструкции цикла, которые задают многократное повторение участка кода. И если в наборе инструкций процессора отсутствует непосредственная инструкция цикла, таковым можно считать частный случай инструкции условного перехода с изменением одного из адресов следующей выполняемой инструкции на точку входящей ветки потока управления. Пример МК для процессора PowerPC с комментариями к инструкциям приведен ниже:

```
; пример цикла с инкрементацией регистра r1 от 0 до 10
      li r1, 0x0          ; занесение в регистр r1 значения 0
.Label:                   ; метка Label
      addi r1, r1, 0x1    ; инкрементирование регистра r1 на 1
      cmpwi r0, r1, 0xA   ; сравнение регистра r1 и числа 10 с занесением результата в r0
      ble r0, .Label      ; переход на метку Label, если результат сравнения «меньше или
равно»
```

Анализ графа потока управления может определить наличие в нем УС – циклов. Ценность получаемой информации аналогична той, которая получается для инструкций условного перехода.

6) Инструкции вызова подпрограммы. Для вызова внешних подпрограмм АК содержит инструкции перехода на начальные адреса первых; по завершению подпрограммы, управление передается на следующий за точкой вызова адрес. Пример АК для процессора PowerPC с комментариями к инструкциям приведен ниже:

```
; пример вызова подпрограммы Funct(x, y) с передачей значений параметров 1 и 2
      li r1, 0x1   ; занесение в регистр r1 значения 1
      li r2, 0x2   ; занесение в регистр r2 значения 1
      bl Funct     ; вызов подпрограммы-функции с двумя параметрами (r1 и r2)
```

Анализ графа вызовов подпрограмм может восстановить частичную архитектуру ПрК, поскольку определит взаимосвязь между ее подпрограммами и алгоритмами. Анализ же графа потока данных и его деталей поможет восстановить частичную сигнатуру подпрограммы – используемые без явной инициализации регистры, скорее всего, являются входными параметрами подпрограммы, а последние инициализированные без явного использования регистры – ее возвращаемым значением (единичным или целым их набором для структур и массивов).

Код данных

Поскольку любой нетривиальный МК в работе своих алгоритмов использует глобальные данные простых типов (константы, глобальные переменные) и составных типов (массивы и структуры; объединения являются вырожденным случаем), то часто для их хранения в коде отводится отдельная область – код данных. Пример АК для процессора PowerPC с комментариями приведен ниже:

```
Global_Var:         ; декларация глобальной переменной Global_Var
      .long  0xA    ; начальное значение 10 для переменной Global_Var
                    ; (аналогично декларации глобальной константы)
Global_Arr:         ; декларация глобального массива Global_Arr
                    ; (аналогично декларации глобальной структуры)
      .long  0x1    ; начальное значение 1 для 1-го элемента массива Global_Arr
      .long  0x2    ; начальное значение 2 для 2-го элемента массива Global_Arr
      .long  0x3    ; начальное значение 3 для 3-го элемента массива Global_Arr
      .long  0x4    ; начальное значение 4 для 4-го элемента массива Global_Arr
      .long  0x5    ; начальное значение 5 для 5-го элемента массива Global_Arr
```

Анализ размещения глобальных переменных в коде данных и их использования в подпрограммах позволяет, во-первых, определить их семантический тип (1, 2-х, 4-х байтовый для простых типов и б*о*льший для составных), а, во-вторых – связь между алгоритмами подпрограмм посредством «разделяемых» (shared) данных.

Отладочная информация

Для отладки исполняемого кода он зачастую собирается в МК вместе с дополнительной информацией (например, в формате DWARF [127]), не используемой при работе программы. Информация состоит из таких записей, как привязка к строкам ИК, имена переменных, их типы и проч. Такая отладочная информация позволяет полноценно отлаживать ИК, отображать значения переменных, стек вызова функций и т. п.

Пример DWARF-описания переменных для программы:

```
1: int a;
2: void foo();
3: {
4:      register int b;
5:      int c;
6: }
```

будет следующим:

```
<1>: DW_TAG_subprogram
        DW_AT_name = foo
<2>: DW_TAG_variable
        DW_AT_name = b
        DW_AT_type = <4>
        DW_AT_location = (DW_OP_reg0)
<3>: DW_TAG_variable
        DW_AT_name = c
        DW_AT_type = <4>
        DW_AT_location = (DW_OP_fbreg: -12)
<4>: DW_TAG_base_type
        DW_AT_name = int
        DW_AT_byte_size = 4
        DW_AT_encoding = signed
<5>: DW_TAG_variable
        DW_AT_name = a
        DW_AT_type = <4>
        DW_AT_external = 1
        DW_AT_location = (DW_OP_addr: 0)
```

В преобладающем большинстве случаев отладочная информация отсутствует в конечных продуктах, как из соображения оптимизации (по размеру и скорости), так и для сокрытия информации о работе кода.

1.3.2 Требования к модели машинного кода и ее классическая реализация

Используя низкоуровневую информацию в МК и возможность ее применения для алгоритмизации, выделим свойства Модели, которые смогут наиболее полно отразить предметную область в интересах поиска уязвимостей.

Элементы и связи Модели

Определим вначале элементы Модели, которые будут отражать особенности алгоритмизации. Во-первых, объектом для применения последней является МК, который должен быть связан с большинством других элементов предметной области. Во-вторых, суть алгоритмизации заключается в восстановлении СМД – значит, некое их представление должно быть четко отражено в Модели. В-третьих, конечной целью служит поиск уязвимостей, а, следовательно, введение в саму Модель таких элементов или их свойств сильно упростило и расширило бы его применение. Также, целесообразно для уязвимостей использовать сделанную ранее их типизацию по структурному уровню. Следует отметить, что в Модели достаточной будет поддержка не столько окончательной и истинной информации об уязвимостях (что практически невозможно), сколько информации об их предполагаемом наличии, конечное решение о которых остается за Экспертом-БК.

Модель должна отражать следующие элементные связи. Во-первых, соответствующие блоки МК должны быть связаны со всеми другими элементами Модели, поскольку первые являются исходным материалом для восстановления вторых; также такая привязка даст необходимую информацию при проведении моделирования, например, точные адреса подпрограмм и возможных уязвимостей в МК. Во-вторых, элементы СМД должны содержаться в Модели согласно их введенной логической иерархии; так, архитектура должна состоять из модулей, мо-

дули из подпрограмм, подпрограммы из алгоритмов, а последние – из УС и вычислительных операций. И, в-третьих, потенциальные уязвимости должны быть также связаны со СМД, поскольку тип первых однозначно определяет место нахождения их во вторых.

Общенаучные требования к Модели

Помимо определения элементов Модели, к ней предъявляются «общенаучные» требования, которые можно определить как следующие.

1) Адекватность. Модель должна соответствовать представляемому ею МК, в котором возможно наличие разно-типовых уязвимостей и который должен быть описан в алгоритмизированном виде.

2) Анализируемость. Получаемые на Модели результаты, а именно алгоритмы и архитектура, должны подходить для последующего ручного анализа.

3) Универсальность. Модель, по возможности, должна мало зависеть от языка ИК, процессора МК и различных модификаций языка конечного описания его алгоритмов.

4) Целесообразность. Ресурсы, затрачиваемые на использование Модели, должны подтверждать преимущество алгоритмизации, как низко-трудоемкой и высокоэффективной по сравнению с аналогами.

Поскольку МК является основой работы компьютерных программ, то и его исследование целесообразно проводить в такой же среде, т. е. при помощи компьютерного моделирования. А из-за масштабности и сложности МК, как и полной формализации его основных элементов (инструкций), необходимо применение математического моделирования. Также, по причине отождествления МК с реальным миром (по сути МК является лишь отражением конечной реализации идеи ПрК), как сама Модель, так и способы ее моделирования и проведения экспериментов полноценно могу быть построены только с помощью внешних методов и алгоритмов. Это приводит к необходимости наличия у Модели средств обмена информацией и управления ею – т. е. наличию внешних интерфейсов.

L-модель

Анализ сферы моделирования МК показал отсутствие каких-либо моделей в принципе . Если быть более точным, то существует лишь «классическая» линейная модель (далее – L-модель), явно нигде не упоминаемая и подразумеваемая «de-facto». Схема L-модели, применимой для использования при алгоритмизации (путем внесения в нее всех требуемых элементов – МК, уязвимостей, СМД), представлена на рисунке 1.7.

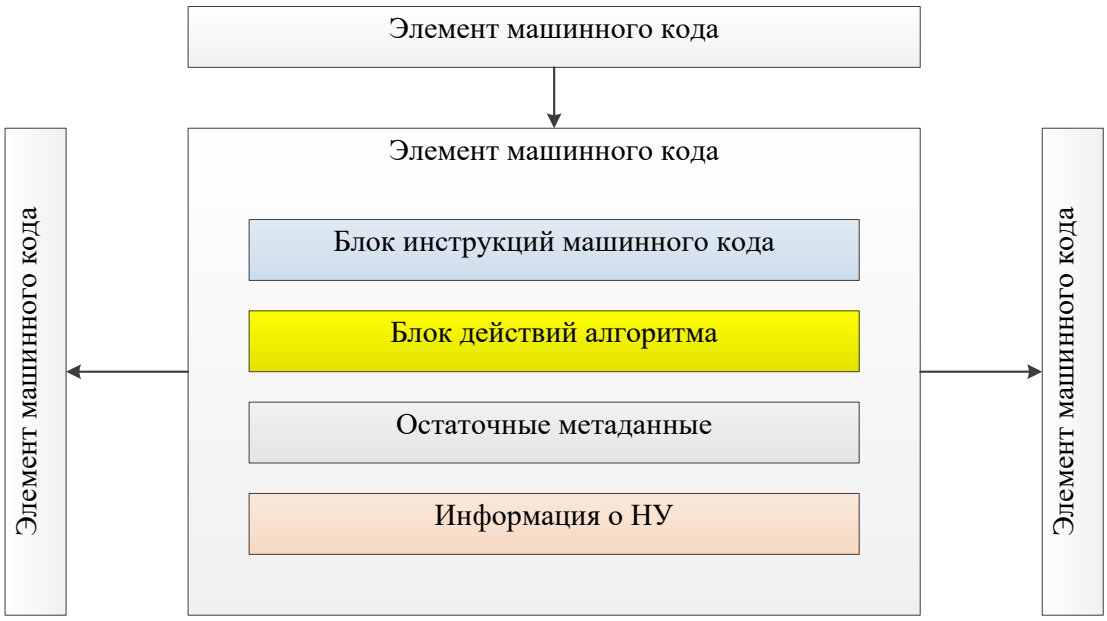

Рисунок 1.7 – Схема L-модели машинного кода

Основными элементами L-модели являются инструкции МК, поделенные на блоки (в лучшем случае – на подпрограммы), что неизбежно приводит к затруднительному представлению более высокоуровневых абстрактных элементов (архитектуры, модулей, подпрограмм, алгоритмов). Так, например, если НУ и представляются моделью, то СУ «размыты» по всем элементам кода, что не дает эффективно применять алгоритмы их поиска. ВУ так и вовсе не представляется возможным обнаруживать с использованием такой модели. То же самое относится и к алгоритмам подпрограмм, нераздельно связанным в модели с МК, который (как неоднократно отмечалось в авторских статьях [67, 75, 83]) не подходит для ручного анализа. Остаточные же метаданные в МК, также присутствующие в L-модели, не предназначены для осуществления алгоритмизации; ко всему прочему они, как

и алгоритм, являются неотделяемой частью МК. С учетом вышеизложенного, использование алгоритмизации на базе данной (возможно и модифицированной, но все же «классической») модели нецелесообразно.

Завершая анализ применимости данной модели, кратко укажем на выполнение ею выдвинутых общенаучных требований. Модель имеет недостаточную адекватность для алгоритмизации кода; также анализируемость получаемых на ней результатов прогнозируется как низкая. И хотя L-модель и имеет достаточную универсальность для представления кода, однако крайне высокая сложность ее использования сводит на нет это единственное преимущество. Таким образом, логичным является создание оригинальной Модели, построенной на заданных элементах и удовлетворяющей всем заданным требованиям.

1.3.3 Вектор структурных моделей машинного кода

Прежде чем непосредственно строить новую Модель, определим ее в вопросе алгоритмизации МК для последующего поиска уязвимостей. Очевидно, что с одной стороны, Модель должна быть более абстрактной, чем реальный объект – МК, поскольку она лишь отражает его существенные черты – СМД; в частности, уровень абстракции должен быть соизмерим с уровнем архитектуры ПрК. С другой стороны, Модель должна достаточно точно отражать детали кода, поскольку из инструкций МК (являющихся практически минимальными единицами деления) состоят алгоритмы подпрограмм. И, с третьей стороны, конечной целью решения является применимость для поиска (ручного или автоматического) уязвимостей, и, следовательно, Модель должна подходить и для этого, явно вовсе не отражаемого в МК. Также очевидно, что каждый МК является абсолютно уникальным, поскольку изменение даже бита инструкции может коренным образом изменить суть алгоритма. Следовательно, для любого экземпляра МК должна строиться своя – конкретная Модель. Согласно предназначению новой Модели и приведенным особенностям дадим общее название моделям такого рода, как структурным или S-моделям [64]. Как будет показано далее, существует целый вектор S-моделей, уточняющих каждую из предыдущих.

Общая S-модель

Идея предлагаемой S-модели заключается в создании системы вложенных структур абстракций ПрК и линейного МК со следующими свойствами. Во-первых, из-за псевдоортогональности все абстракции связаны со своими представлениями в МК, что позволяет избежать потери какой-либо существенной низкоуровневой информации (эта часть схожа с классической L-моделью). А, во-вторых, алгоритмизация может решать собственные подзадачи на элементах более высоких уровней абстракции (например, производить анализ алгоритмов функций и построение связей между ними), поскольку последние имеют выделенную плоскость представления. Таким образом, без потери детальной информации о МК, Модель заключает в себе основные иерархические элементы соответствующего ему ПрК.

Модель состоит из следующих элементов и связей. Во-первых, в ней присутствуют все элементы СМД: архитектура ПрК, модули, подпрограммы, алгоритмы с УС; притом, эти элементы являются иерархически связанными. Во-вторых, УС используют результаты вычислений, что также отражаются в Модели. В-третьих, УС в рамках каждого алгоритма взаимосвязаны и определяют логику его работы – т. е. связаны графом потока управления. В-четвертых, модули Модели содержат данные (простых и составных типов), используемые процедурами в рамках модуля – т. е. связаны графом разделенных данных. В-пятых, подпрограммы могут вызывать друг друга: напрямую в рамках модуля и через интерфейсы из других модулей – т. е. связаны графом внутренних и внешних (относительно модуля) вызовов. И в-шестых, все приведенные элементы имеют ссылки на инструкции МК.

Описанную Модель можно считать общей для МК и не привязанной к конкретному (в том числе и разрабатываемому) способу; хотя основным ее предназначением конечно можно считать алгоритмизацию МК. Будем называть такую модель общей S-моделью.

Адаптированная S-модель

Поскольку алгоритмизация на S-модели осуществляется для достижения непосредственной цели – поиска уязвимостей, то целесообразна адаптация общей S-модели для поиска уязвимостей, т. е. создание адаптированной S-модели. Ее построение производится статически с помощью анализа множества шаблонов уязвимостей и выделения их структурных свойств.

Отличие адаптированной S-модели от общей заключается в наличии дополнительных элементов – информации для поиска уязвимостей и их связей с остальными. Определение и детализация такой информации выходит за рамки настоящей научной работы, поскольку представляет собой отдельное направление исследования. Для обоснованности ее существования приведем примеры следующих уязвимостей каждого структурного типа, отражаемых в МК, а также способов их поиска. Анализ и выделение всех возможных значений переменных в результате работы алгоритмов позволит обнаружить НУ работы с данными – например, выход индекса массива за пределы диапазона значений. Анализ вызовов алгоритмом внутренних подпрограмм этого же модуля при отсутствии проверок необходимых условий позволит обнаружить возможную СУ в логике работы алгоритма – например, установку без необходимых прав успешности результата проверки пароля (потребует от человека дополнительной настройки поиска). Анализ вызовов алгоритмом внешних подпрограмм другого модуля в обход общего интерфейса позволит обнаружить возможную ВУ – использование нестандартных резервных или скрытых каналов передачи управления и данных. Также для использования внешних алгоритмов работы с Моделью необходимо наличие у последней соответствующих интерфейсов (математических, программных).

Схема адаптированной S-модели с использованием недетализированных шаблонов уязвимостей представлена на рисунке 1.8 (здесь *инф.* = информация).

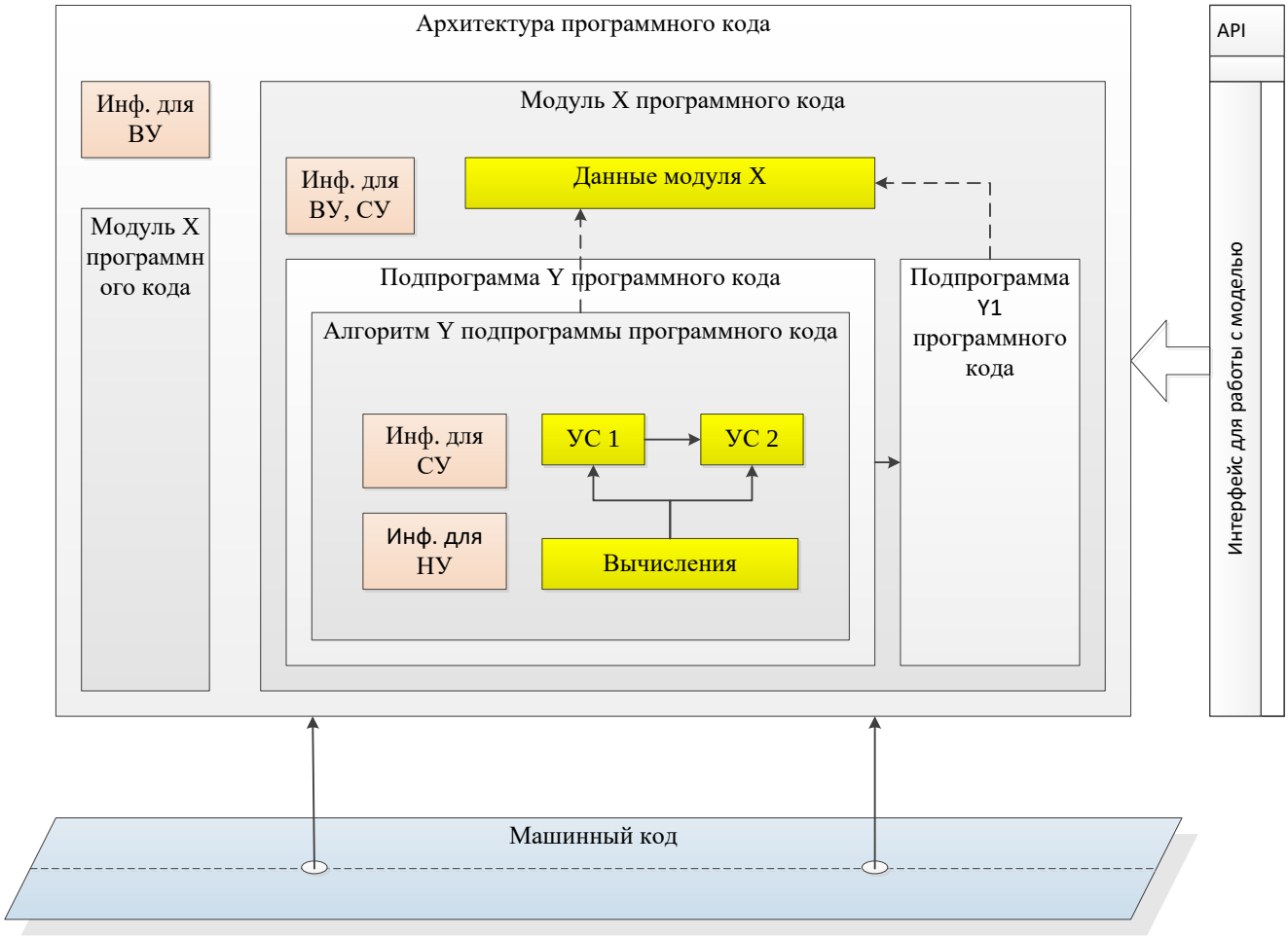

Рисунок 1.8 – Схема адаптированной S-модели машинного кода

Адаптированную S-модель целесообразно рассматривать исключительно в интересах алгоритмизации МК для последующего поиска уязвимостей.

Как было отмечено, ни общая, ни адаптированная S-модели не предназначаются для непосредственного применения при алгоритмизации – они являются лишь подготовительными и обобщенно отражают структуру любого МК. Тем не менее, они и определяют общую форму конечной S-модели.

Частная S-модель

Поскольку в интересах алгоритмизации необходимо представление конкретного экземпляра МК, что не отражают рассмотренные S-модели, введем так называемую частную S-модель, являющуюся отражением исследуемого образа

МК. Модель является конечной в схеме моделирования и исходной для проведения эксперимента. Построение производится динамически путем наложения адаптированной S-модели на конкретный экземпляр МК. Таким образом получается уникальный формализованный образ последнего, подходящий для автоматической обработки. Схему модели описывать нецелесообразно, поскольку она будет подобна адаптированной S-модели, детализированной для выбранного МК.

Формализацию частной S-модели целесообразно делать полностью независимой от инструкций конечного процессора выполнения (за исключением, естественно, ссылок на оригинальные инструкций МК) – это сделает модель необходимо-абстрактной, а, значит, и пригодной для работы. Так, МК не нуждается в специальной формализации и может храниться как есть – в виде последовательности байтов или текстовых ассемблерных строк. Вычисления в алгоритмах же целесообразно перевести в абстрактную форму деревьев, чтобы операции в них являлись независимыми от исходного МК – это соответствует некому аналогу байт-кода. УС также могут иметь абстрактную форму, но предназначенную для описания логики выполнения алгоритмов, например – граф управления. Совокупность этого графа и деревьев и будут составлять внутреннее представление каждого алгоритма (содержащееся в соответствующем контейнере). Последнее может быть дополнено информацией о возможных значениях используемых переменных и их времени жизни, собранных путем анализа графа и деревьев. Каждый контейнер-алгоритм упаковывается в подпрограмму, имеющую собственную сигнатуру – входные и выходные параметры, полученные путем анализа алгоритмов. Контейнер-модуль содержит свои глобальные данные и подпрограммы: по использованию первых внутри вторых строится граф разделенных данных. Также по вызовам одних подпрограмм другими строится граф внутренних вызовов. Модули содержатся в контейнере-архитектуре, который также имеет граф внешних вызовов, построенный по использованию подпрограмм одних модулей другими. Большинство элементов содержат связанную с ними информацию для поиска уязвимостей: контейнер-алгоритмы соответственно – информацию для поиска НУ и СУ, кон-

тейнер-модули – для поиска СУ и ВУ, контейнер-архитектура – для поиска ВУ. Все элементы должны содержать привязку к соответствующим инструкциям МК.

1.3.4 Концептуальный подход к алгоритмизации машинного кода

Исходя из многосторонности требований к Модели, предлагается следующий концептуальный подход к решению задачи алгоритмизации МК. Во-первых, перевод множества вариантов МК в обобщенный вид возможен с помощью общей S-модели. Во-вторых, для систематизации информации об уязвимостях предназначена адаптированная S-модель. Окончательное же построение частной S-модели уже будет содержать в себе все необходимые СМД и информацию об уязвимостях. Следовательно, сам процесс такого построения по сути можно считать основой алгоритмизации. Дополнением должен лишь стать перевод Модели в описание, подходящее для анализа Экспертами-БК. Таким образом, алгоритмизацию МК для поиска уязвимостей можно рассматривать, как процесс моделирования частной S-модели экземпляра МК с проведением итерационных экспериментов на ней для получения требуемого алгоритмизированного описания кода. Так, целесообразно трехэтапное построение частной S-модели МК: на каждом этапе будет построена соответствующая модель, детализирующая и дополняющая предыдущую некоторым образом (или в некоторой плоскости), решающая собственные задачи и имеющие принципиальные особенности.

Подытожим вектор моделей, используемых в подходе к алгоритмизации. Все создаваемые модели призваны отражать структурные свойства МК, поэтому имеют общее название – структурные. Первая S-модель названа *общей*, поскольку является альтернативой классической и описывает МК с новой позиции. Вторая S-модель названа *адаптированной,* поскольку является эволюцией первой и дополнительно отражает информацию для обнаружения уязвимостей в МК на основании их шаблонов и особенностей. Третья S-модель названа *частной* и является конечной эволюцией моделей с конкретным отображением экземпляра МК, восстановленными СМД и привязанной к ним информацией об уязвимостях и исходных инструкциях МК.

Общая схема моделирования и проведения эксперимента

Согласно предложенному концептуальному подходу решения задачи алгоритмизации, схема моделирования МК и проведения эксперимента на нем имеет следующий вид. Изначально создается общая S-модель, отражающая структурную информацию любого МК. Модель является абстрактной, поскольку, очевидно, что никакие алгоритмы на ней не дадут сколько-нибудь существенной информации о конкретном экземпляре МК. Для построения используется собранная обобщенная низкоуровневая информация, которая присутствует в любом МК. Затем происходит ее преобразование к адаптированной S-модели путем выделения информации для поиска уязвимостей. Эта модель, как и первая, имеет полезность в виде получения новых знаний о МК и является основой для построения следующей модели. Последняя частная S-модель уже является точным отображением МК, но в плоскости, подходящей для ручного и автоматического анализа. Именно на ней возможно построение конкретной реализации решения задачи в виде способа алгоритмизации; для этого должна существовать возможность ее построения. Поскольку ручной анализ на любой из S-моделей очевидно носит субъективный характер, то необходима ее «подстройка» в интересах повышения эффективности конечного результата моделирования – информации для поиска уязвимостей. Это осуществимо типичной для моделирования операцией – путем оптимизации модели. Схема такого моделирования и проведения модельного эксперимента показана на рисунке 1.9 [85].

Согласно схеме, изначально последовательно строятся общая и адаптированная абстрактные модели обобщенного Представления МК (для множества всех возможных вариантов МК с отражением типовых особенностей уязвимостей), затем конкретная модель экземпляра МК (частная S-модель, детализирующая предыдущие модели), содержащая все выделенные СМД. По последней проводится модельный эксперимент с восстановлением алгоритмизированного Представления МК. Результатом эксперимента является описание архитектуры и алго-

ритмов МК, «удачность» которого определяется пригодностью Представления для анализа Экспертом-БК.

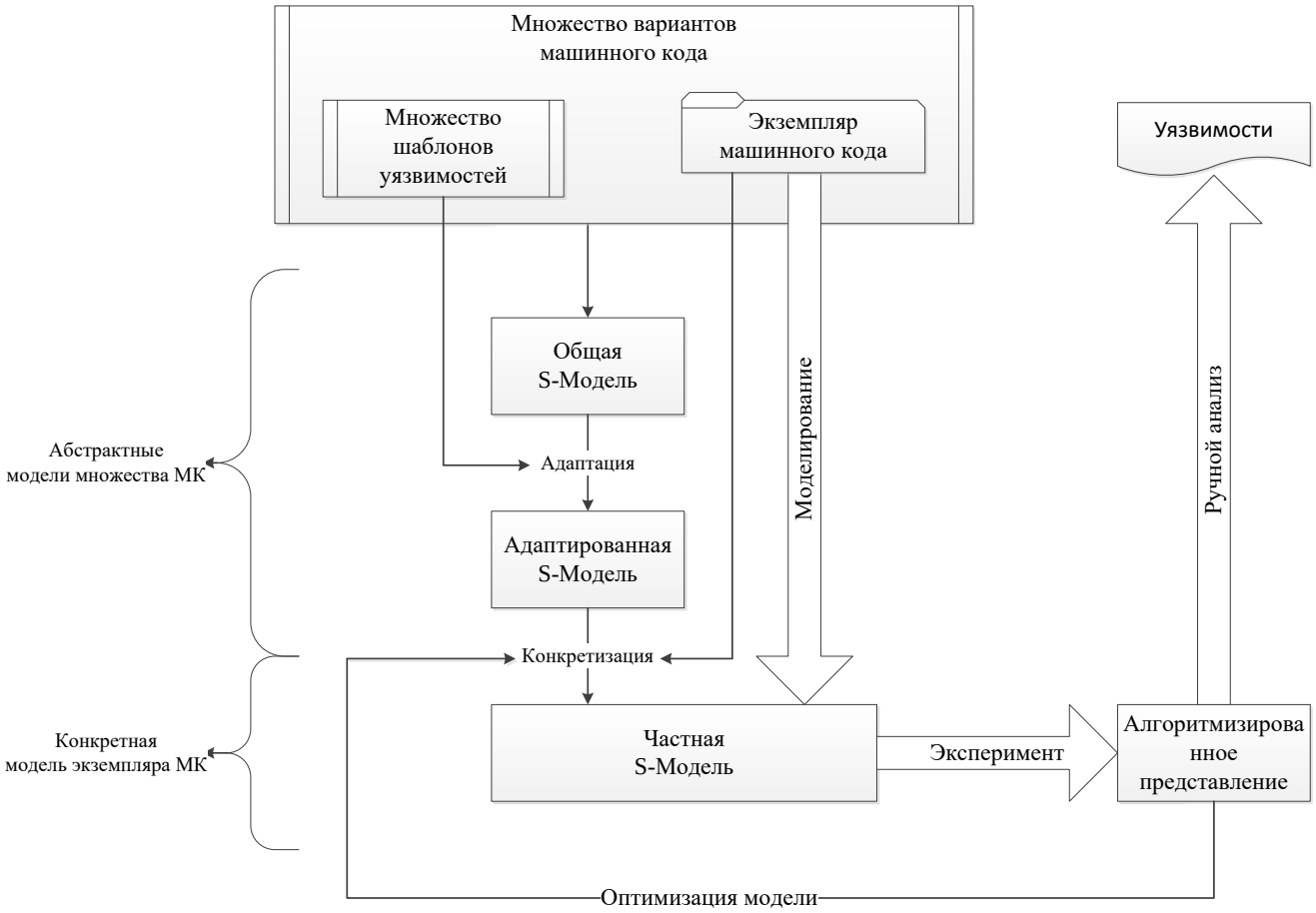

Рисунок 1.9 – Схема моделирования машинного кода

По полученным результатам, в случае необходимости, производится оптимизация частной S-модели путем «ручного» задания поправок, повторение моделирования и эксперимента. Удовлетворительные результаты же могут быть использованы непосредственно для поиска уязвимостей.

Обобщенные требования к схеме алгоритмизации

Исходя из предложенной схемы моделирования и проведения эксперимента, сформулируем следующие требования к алгоритмизации МК. Во-первых, основой всей схемы должна быть частная S-модель экземпляра МК, поскольку она является центральной – как при моделировании, так и при проведении эксперимента.

Во-вторых, основным назначением алгоритмизации будет построение этой модели, максимально точно отражающей конкретный экземпляр МК, поскольку в ней уже будет отражена все необходимая информация. В-третьих, алгоритмизация должна иметь возможность настройки, позволяя тем самым производить оптимизацию модели; это осуществимо путем добавления дополнительных корректировок к входным данным процесса моделирования. В-четвертых, на последних шагах алгоритмизации должно осуществляться преобразование Модели к алгоритмизированному Представлению (например, путем генерации по Модели текстового описания последнего), подходящему для анализа Экспертом-БК. И, в-пятых, реализация S-модели должна быть компьютерной и математической, как наиболее подходящей для поддержки сложными алгоритмами процесса моделирования.

Выводы по разделу 1

В качестве объекта исследовалось представление машинного кода с уязвимостями на предмет содержащихся в нем структурных метаданных, под которыми понимаются архитектура, модули, подпрограммы и алгоритмы (с управляющими структурами – последовательностями, ветвлениями и циклами) программного кода.

В ходе исследования:

1) Рассмотрены проблемные вопросы обеспечения безопасности программного кода телекоммуникационных устройств и выбран наиболее перспективный из них для решения – поиск уязвимостей в исполняемом коде.

2) Произведена типизация уязвимостей программного кода по их структурному уровню. Рассмотрены известные способы поиска уязвимостей в машинном коде и произведено критериальное сравнение их эффективности.

3) Синтезирована схема областей жизни уязвимостей в представлениях программного обеспечения, на основании которой выделено направление для решения выбранного проблемного вопроса.

4) Идентифицированы структурные метаданные в машинном коде согласно используемым парадигмам программирования. Установлено, что применение модульной парадигмы приводит к наличию в машинном коде информации о модулях программного кода и их взаимосвязи (т. е. архитектуре), процедурной – подпрограммах и их взаимодействия, структурной – алгоритмах и управляющих структурах, императивной – взаимосвязи метаданных через переменные.

5) Выявлена низкоуровневая информация в машинном коде, которая может быть использована для идентификации структурных метаданных, а именно: в коде выполнения – переходы на адреса для подпрограмм, вычисления для параметров управляющих структур, безусловные/условные переходы и циклы для управляющих структур, вызовы подпрограмм для сигнатуры подпрограмм и архитектуры программного кода; в коде данных – глобальные переменные для связи алгоритмов подпрограмм через «разделяемые» данные.

6) Сформулированы требования к модели машинного кода, применимой для его алгоритмизации и последующего поиска уязвимостей. Так, во-первых, заданы отражение и взаимосвязь основных элементов модели: блоков машинного кода, структурных метаданных, дополнительной информации об уязвимостях. Во-вторых, определены общенаучные требования к модели: адекватность, анализируемость, универсальность и целесообразность. И, в-третьих, обоснована необходимость наличия у модели внешних интерфейсов для управления и обмена информацией.

7) Описана классическая линейная модель машинного кода, подразумеваемая «de-facto» (L-модель) и обоснована ее неприменимость в интересах алгоритмизации. Задан вектор структурных моделей (S-моделей), позволяющих использовать на их базе алгоритмизации и последующий поиск уязвимостей. Вектор состоит из 3-х моделей, уточняющих каждую из предыдущих: общая, адаптированная и частная.

8) Предложен концептуальный подход к алгоритмизации машинного кода, основанный на его S-моделировании с последующим проведением эксперимента

для получения алгоритмизированного представления. Сформулированы требования к реализации предложенного подхода.

Основным научным результатом, изложенным в первом разделе, является структурная модель машинного кода с уязвимостями (S-модель), суть которой заключается в его представлении в виде иерархии структурных метаданных и разноуровневых уязвимостей. Также, модель содержит специальную информацию, используемую для поиска уязвимостей различных типов: низко- средне- и высокоуровневых.

Частными научными результатами, изложенными в первом разделе, являются: критерии эффективности известных способов поиска уязвимостей, структурная типизация уязвимостей программного кода, схема областей жизни уязвимостей в представлениях программного обеспечения, линейная L-модель и промежуточные S-модели машинного кода. Первый результат непосредственно был использован для установления условия достижения цели исследования, а последние – для получения основного научного результата.

Основное содержание раздела и полученных научных результатов изложено в работах автора [61, 64, 66-67, 71, 75, 80, 82-86, 88-90].

2 СИНТЕЗ МЕТОДА АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ

2.1 Построение схемы алгоритмизации машинного кода для поиска уязвимостей

В интересах алгоритмизации создадим соответствующий ей метод (далее – Метод), для чего синтезируем общую схему алгоритмизации, достаточно абстрактную и инвариантную к конкретной реализации МК. Ее целесообразно пошагово строить по предложенной схеме моделирования МК и проведения модельного эксперимента (и согласно сформулированным требованиям), поскольку они определяют один и тот же процесс, но с разных позиций. Также она призвана сгенерировать требования к реализации собственно Метода алгоритмизации.

2.1.1 Гипотетическая схема алгоритмизации машинного кода

Гипотетическая схема (далее – Схема) состоит из 13 шагов, основанных на поддержке S-модели, и имеет вид, представленный на рисунке 2.1.

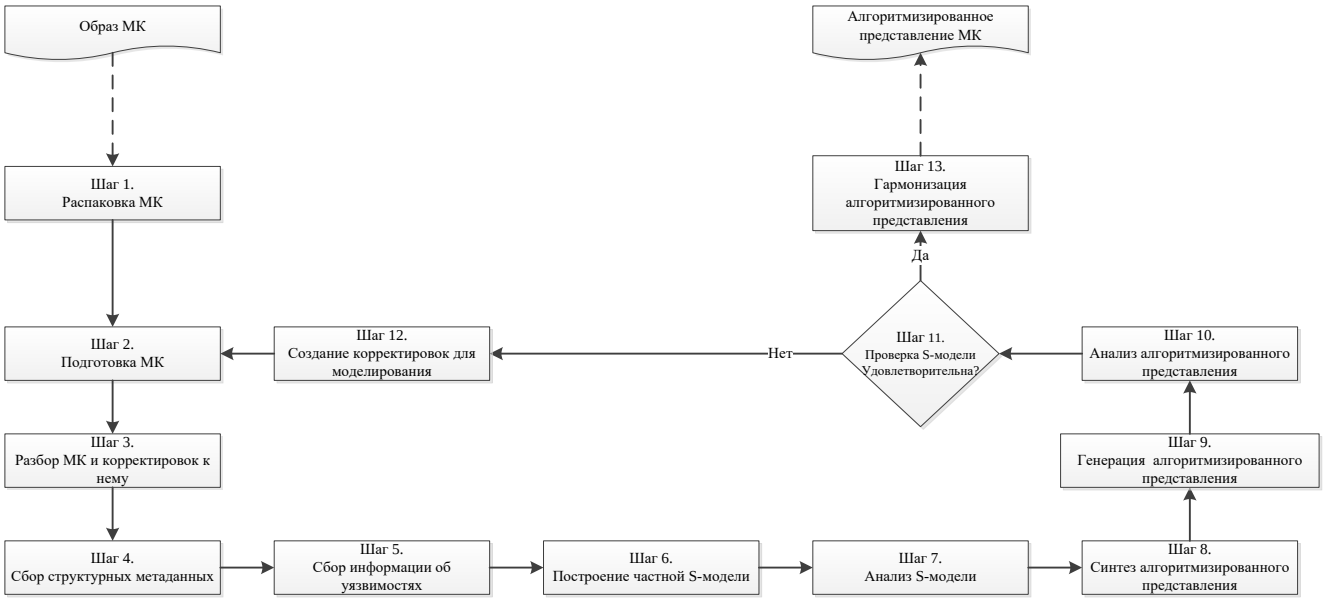

Рисунок 2.1 – Гипотетическая пошаговая схема алгоритмизации машинного кода

Согласно рисунку 2.1 Схема содержит следующие шаги.

Шаг 1 – «Распаковка МК». Производится распаковка экземпляра МК в бинарной форме.

Шаг 2 – «Подготовка МК». Производится подготовка распакованного содержания экземпляра МК к обработке с заданием (в случае итерации) базовых корректировок построения модели.

Шаг 3 – «Разбор МК и корректировок к нему». Производится разбор подготовленного содержания МК и его преобразование к формализованному виду с выделением элементов будущей S-модели. Принимаются и частично учитываются корректировки к моделированию.

Шаг 4 – «Сбор СМД». Выделяются и систематизируются СМД по МК. В случае необходимости, осуществляются оптимизационные действия.

Шаг 5 – «Сбор информации об уязвимостях». Выделяется и систематизируется информация об уязвимостях. Последние могут определяться на основании шаблонов.

Шаг 6 – «Построение частной S-модели». Производится анализ собранных СМД и уязвимостей, по которым строится частная S-модель экземпляра МК. Учитываются корректировки к построению, заданные при обработке МК.

Шаг 7 – «Анализ S-модели». Обрабатывается S-модель с целью сбора и подготовки информации к генерации алгоритмизированного Представления.

Шаг 8 – «Синтез алгоритмизированного Представления». Генерируется алгоритмизированное Представление МК на основании собранной по S-модели информации. Вид Представления является подходящим для анализа Экспертом-БК.

Шаг 9 – «Генерация алгоритмизированного представления». Генерируется описание архитектуры и алгоритмов МК для ручного анализа, соответствующее алгоритмизированному Представлению.

Шаг 10 – «Анализ алгоритмизированного Представления». Полученное синтезированное Представление МК анализируется Экспертом-БК, субъективно оценивается на предмет возможности поиска по нему уязвимостей.

Шаг 11 – «Проверка S-модели. Удовлетворительна?». Субъективно анализируется удовлетворительность (избыточность, недостаточность, точность) алгоритмизации МК.

Шаг 12 – «Создание корректировок для моделирования». На основании анализа причин недостаточности алгоритмизации МК создается совокупность корректировок к построению S-модели для повторного моделирования

Шаг 13 – «Гармонизация алгоритмизированного Представления». В случае удовлетворительности S-модели восстановленные алгоритмы МК собираются и гармонизируются в отдельное Представление; целью гармонизации в данном случае является приспособленность для последующего анализ на предмет наличия уязвимостей.

Необходимо отметить, что приведенные шаги являются логическими и в реальности могут быть объединены в группы, реализованные с помощью единых ПС или процессов. Такие объединения будем называть этапами Метода. Также, как хорошо видно, описание шагов является абстрагированным от конкретных реализаций Метода, выбор и обоснование которых будет произведено далее.

2.1.2 Поддержка структурной модели

Основополагающим концептуальным элементом Схемы является S-модель, от *удачности* построения и *успешности* обработки которой зависит применимость получаемого результата. Так, если она будет недостаточно детализированной, то некоторые уязвимости могут не отразиться в ней, если же слишком детализированной – низкий уровень абстракции и структуризации элементов не позволит работать с восстановленными алгоритмами. Если форма ее представления будет слишком простой, то произведенная алгоритмизация получится «размытой», если будет слишком сложной – то трудоемкость разработки алгоритмов работы с Моделью окажется чрезмерной. При этом, сильная зависимость Модели, как от входного процессора, так и от выходного языка описания плохо отразятся,

как на абстрактности реализации Метода, так и на его переносимости – что, в конечном счете, приведет к ошибкам в его работе.

Как указывалось ранее, ручное построение и обработка Модели имеют запредельную и неоправданную трудоемкость, поэтому Модель целесообразно сделать математической, а построение и проведение эксперимента осуществить с помощью специальных (существующих или разработанных) ПС.

Выделение СМД и восстановление алгоритмизированного Представления возможно проводить напрямую на создаваемой Модели путем последовательного уточнения ее информации: добавления новых алгоритмов, уточнения сигнатур подпрограмм, образования модулей и т. п. Для описания восстановленных алгоритмов в конечной форме, предназначенной для анализа вручную, возможно использование отдельного графа или дерева, хранящего ссылки на элементы Модели. Гипотетический пример преобразований различных внутренних представления экземпляра МК в рамках Метода для простейшего цикла с декрементацией переменной показан на рисунке 2.2

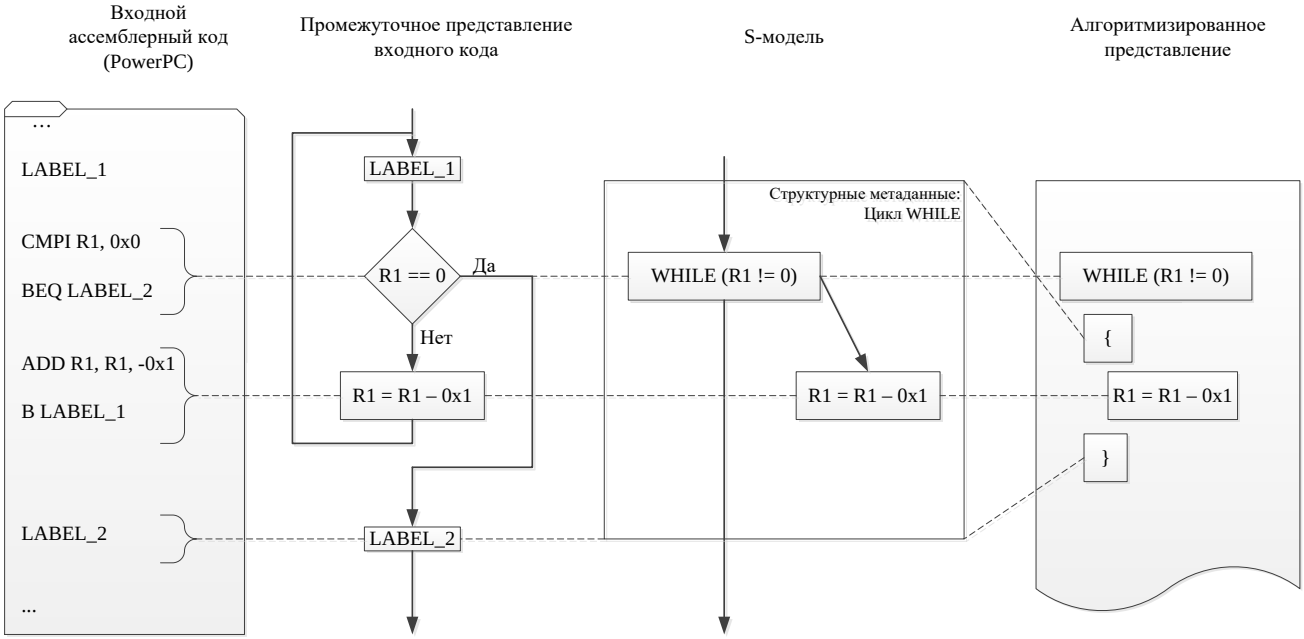

Рисунок 2.2 – Гипотетический пример преобразований внутренних представлений МК для цикла

Согласно рисунку, для экземпляра МК в виде ассемблерного текста через некое его промежуточное представление строится S-модель, из элементов которой синтезируется конечное алгоритмизированное Представления. Так, в S-модели содержится программа в структурированном виде с выделенными СМД, а Представление алгоритма является отдельно-стоящим деревом, элементы которого имеют ссылки на эту Модель.

2.2 Систематизация существенных аспектов алгоритмизации машинного кода

Согласно синтезированной Схеме выделим ее аспекты, рассмотрение и систематизация которых с позиции существующих практик позволит реализовать шаги Схемы в виде этапов Метода. Существенными аспектами могут считаться нижеперечисленные.

1) Близкие подходы. Для использования имеющегося опыта в области решения задачи алгоритмизации целесообразно рассмотрение и частичное применение близких к ней подходов. Некоторые из них помогут решить лишь отдельные этапы Метода, некоторые же позволят сформировать общее направление решения.

2) Преобразование машинного кода. Поскольку применение Метода требует первоначальной формализации входного МК, необходимо рассмотрение имеющихся способов его разбора. И хотя код изначально является формализованным (состоит из набора байтов, строго задающих инструкции процессора), однако такая его форма без дополнительных преобразований не позволит напрямую построить S-модель.

3) Описание алгоритмов. Основополагающим результатом Метода является описание алгоритмов в виде, хорошо понятном Эксперту-БК и применимым для ручного поиска уязвимостей. Спецификация формата такого описания является отдельной задачей (если не проблемой) другой предметной области. Тем не менее, общие характеристики и удовлетворительный вариант Представления ал-

горитмов могут быть выбраны на основании возможностей Модели и базовых требований к Представлению.

4) Поиск уязвимостей. Хотя конкретная методика поиска уязвимостей в МК, как и описание алгоритмов, выходят за рамки предметной области, тем не менее, сам Метод должен обладать возможностью быть используемым в составе более общего – метода поиска уязвимостей. Для проверки и обоснования этого целесообразно рассмотрение существующих подходов к поиску уязвимостей в МК, притом не только по восстановленным алгоритмам.

5) Моделирование и модельный эксперимент. Хотя как моделирование, так и модельный эксперимент, можно считать уникальными процессами, тем не менее, они имеют собственные законы построения и стандартные действия. Так, связующее звено между ними – оптимизация Модели – должно оперировать понятиями характеристик и эффективности.

Рассмотрим перечисленные подходы, способы и средства, частично или полностью применимые в интересах реализации Метода, более подробно.

2.2.1 Близкие к алгоритмизации подходы

Декомпиляция ассемблерного кода

Наиболее близким подходом к Методу является превращение, обратное компиляции, а именно – *декомпиляция* [47, 48]. В строгом смысле, цель его заключается в получении ИК по скомпилированному, т. е. по АК. В более общем смысле, под декомпиляцией понимается частичное восстановление ИК или информации о нем, не столько обязательное компилируемое, сколько воспринимаемое человеком [128]. Задача декомпиляции является нерешаемой полностью и позволяет получать адекватный результат лишь в определенных условиях; попытки ее решения в виде утилит декомпиляции были предприняты различными научными и коммерческими организациями. Наиболее известной из них является плагин Hex-Rays [129] для продукта IDA Pro [130], который позволяет по МК (для процессоров X86, x64, ARM32, ARM64, и PowerPC) получать его C-подобное

Представление. Также, существуют менее известные утилиты декомпиляции: Boomerang [131], C4Decompiler, DDC, ExeToC Decompiler [132], REC Studio 4 [133], Reko [134], RelipmoC [135], Retargetable Decompile [136], SmartDec [137], Snowman [138] и др. – часть из которых уже практически не поддерживаются, а часть обладает недостаточным функционалом.

Принципы работы декомпиляторов различны и зависят от соответствующих реализаций, но основные этапы следующие. Во-первых, производится разбор МК (или АК после дизассемблирования) и построение его внутреннего представления (в контексте ПС). Во-вторых, в процессе обработки внутреннего представления собирается дополнительная информация об АК. В-третьих, делаются предсказания относительно ИК. В-четвертых, внутреннее представление АК преобразуется к такому же, но для ИК (с использованием сделанных предсказаний). В-пятых, внутреннее представление оптимизируется с целью получения в дальнейшем ИК, подходящего для ручного анализа или компиляции. И, в-шестых, генерируется ИК. Зачастую, если есть такая возможность, используется отладочная информация в МК; тем не менее, для ТКУ она практически всегда отсутствует. Обобщенная схема работы такого декомпилятора представлена на рисунке 2.3.

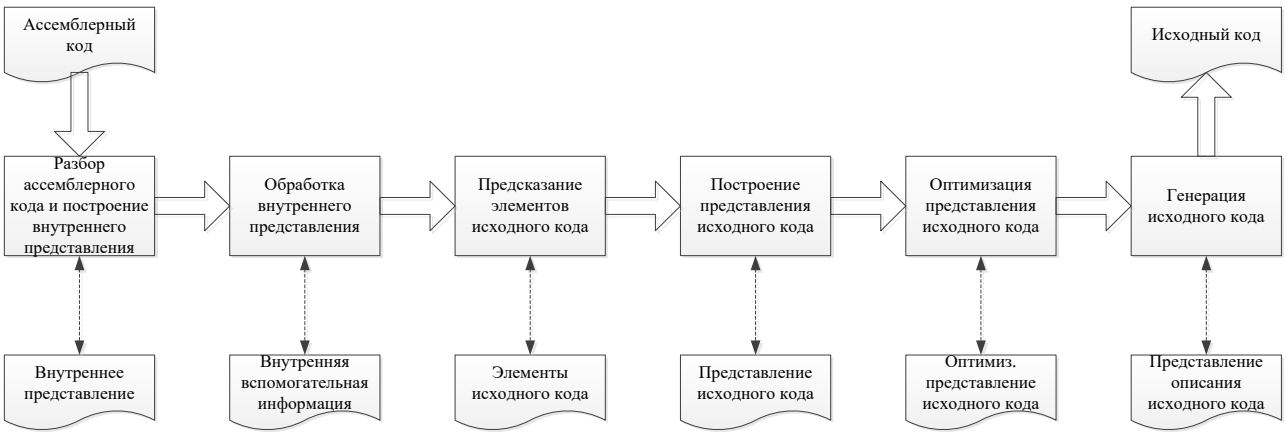

Рисунок 2.3 – Обобщенная схема работы декомпилятора

Декомпиляция повышает уровень абстракции кода, поскольку преобразует его с низкоуровневого языка в высокоуровневый.

Компиляции исходного кода

Поскольку задача Метода схожа с процессом декомпиляции, целесообразно рассмотреть и его антипод – компиляцию. Несмотря на то, что цель компиляции прямо противоположна, тем не менее, общая структура соответствующих утилит и применяемые подходы частично совпадают [139]. Так, компиляторы, как и декомпиляторы, производят разбор входного языка, преобразование во внутреннее представление, обработку, оптимизацию и генерацию выходного языка. Совпадают даже фазы синтаксического анализа и распределения переменных, хотя внутренняя реализация их, конечно, отличается. Утилиты компиляции являются стандартными и входят в состав всех средств разработки ПрК.

Классический принцип работы компиляторов является следующим. Во-первых, производится разбор ИК с представлением его в виде дерева абстрактного синтаксиса, отображающим исходную программу в формализованном виде, составленном из языково-зависимых элементов. Во-вторых, по дереву абстрактного синтаксиса строится внутреннее представление ИК. Все дальнейшие операции производятся на нем, поскольку оно является не зависимым ни от входного языка ПрК, ни от выходного ассемблерного языка, что позволяет использовать языково-независимые алгоритмы. В-третьих, внутреннее представление обрабатывается различными алгоритмами и собирается дополнительная информация об ИК. В-четвертых, производится оптимизация внутреннего представления с целью последующей генерации МК по определенным критериям (как правило, для увеличения скорости работы кода или уменьшения его размера). В-пятых, выбираются области памяти данных и регистры процессора для размещения значений переменных, участвующих в операциях – так называемое *распределение переменных*. И, в-шестых, по конечному внутреннему представлению производится генерация АК (а зачастую напрямую и МК, совмещая в себе, таким образам, фазу ассемблирования). Распространенной и достаточно удобной практикой компиляции является передача в утилиту различных параметров, влияющих на сам процесс и результирующий код. Обобщенная схема работы компилятора представлена на рисунке 2.4.

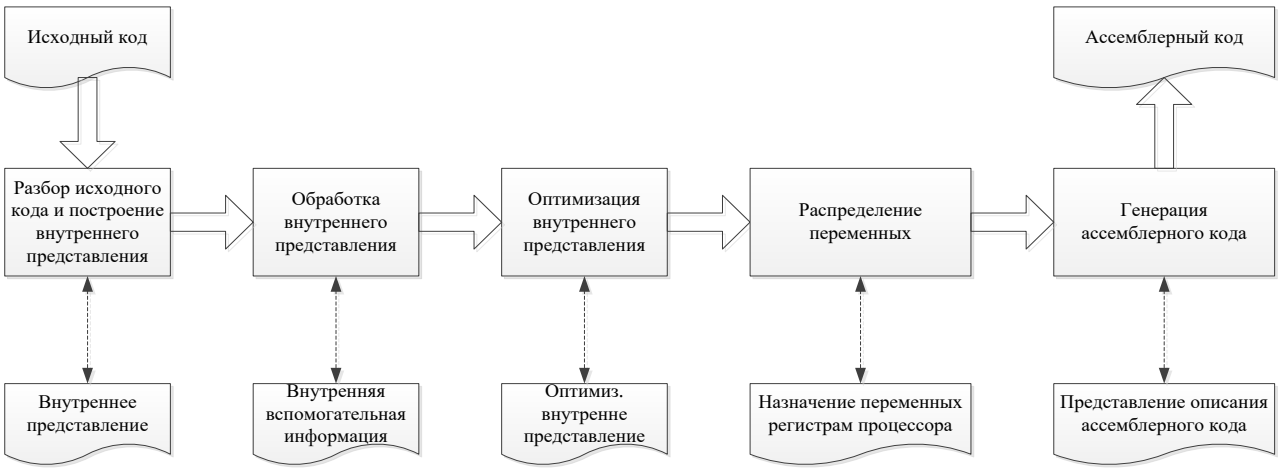

Рисунок 2.4 – Обобщенная схема работы компилятора

Компиляция понижает уровень абстракции кода, поскольку преобразует его с высокоуровневого языка в низкоуровневый.

Трансляции кода

Трансляция является процессом, близким как к компиляции, так и к декомпиляции, с точки зрения изменения абстрактности кода преобразуемых языков. Суть трансляции заключается в преобразовании программы на одном языке, в некотором смысле, в равносильную программу на другом языке. Назначением трансляции является изменение формы представления программы с сохранением ее содержания, например, для обеспечения ее понятности (как ручной, так и машинной) в новых условиях.

Классический принцип работы трансляторов является следующим. Во-первых, производится разбор кода на Языке 1 и построение его внутреннего представления. Во-вторых, внутреннее представление кода на Языке 1 преобразуется к внутреннему представлению кода на Языке 2 согласно заданным правилам. И, в-третьих, по конечному внутреннему представлению производится генерация кода на Языке 2. Обобщенная схема работы транслятора представлена на рисунке 2.5.

Трансляция кардинально не изменяет уровень абстракции кода, поскольку его содержание остается практически неизменным.

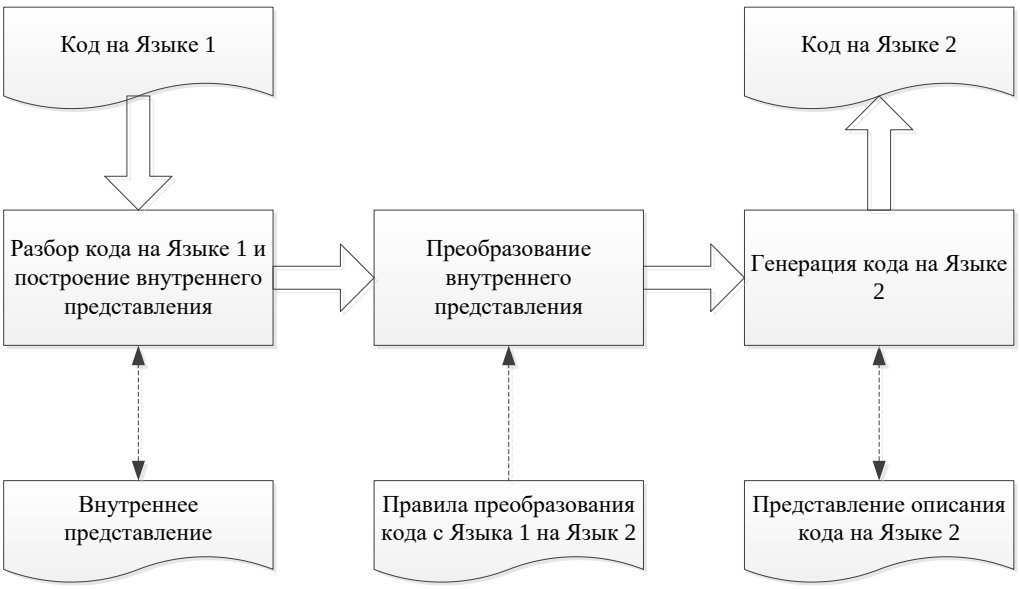

Рисунок 2.5 – Обобщенная схема работы транслятора

Границы применимости близких подходов

Согласно обзору подходов, декомпиляция, хотя и является близкой к Методу, тем не менее, не решает его задачи. Отсутствие полноценной теории (как и решений) не дает возможность использовать ее как теоретическую базу. Для этой цели возможно использование подходов, нашедших широкое практическое применение в компиляторах и трансляторах, а именно разбор входного кода, построение внутреннего представления, работа с деревьями и графами, оптимизация, генерация выходного кода.

2.2.2 Средства преобразования машинного кода

Утилиты дизассемблирования

Поскольку конечное Представление кода имеет вид МК, то для работы утилит, подобных декомпиляторам, необходимо его превращение в ассемблерный – дизассемблирование. Зачастую это осуществляется самим декомпилятором, т. е. последний принимает на вход МК; тем не менее, имеет смысл рассмотреть данное преобразование отдельно. Как было указано ранее, дизассемблирование является достаточно тривиальным прямым преобразованием, реализуемым целой группой

стандартных утилит, имеющих вид консольного приложения или отдельного модуля (и, как правило, входит в состав графической среды дизассемблирования или отладчика кода). Дизассемблеры обычно не изменяют уровень абстракции кода, поскольку лишь преобразуют форму представления инструкций. Минимальное повышение абстракции может быть связано лишь с выделением подпрограмм, автоматически осуществляемое некоторыми из них.

Интерактивные среды дизассемблирования

Среды дизассемблирования являются графическим расширением утилит дизассемблирования, имеющим дополнительный функционал по анализу полученного кода и, иногда, его модификации. Наиболее известными средами являются такие, как IDA Pro и Immunity Debugger [140]. Интерактивные среды дизассемблирования могут дополнительно повышать уровень абстракции кода лишь путем построения графов вызовов и управления. Пример этого для стандартной программы Блокнот (от англ. Notepad) из ОС Windows, дизассемблированной в IDA Pro, приведен на рисунке 2.6.

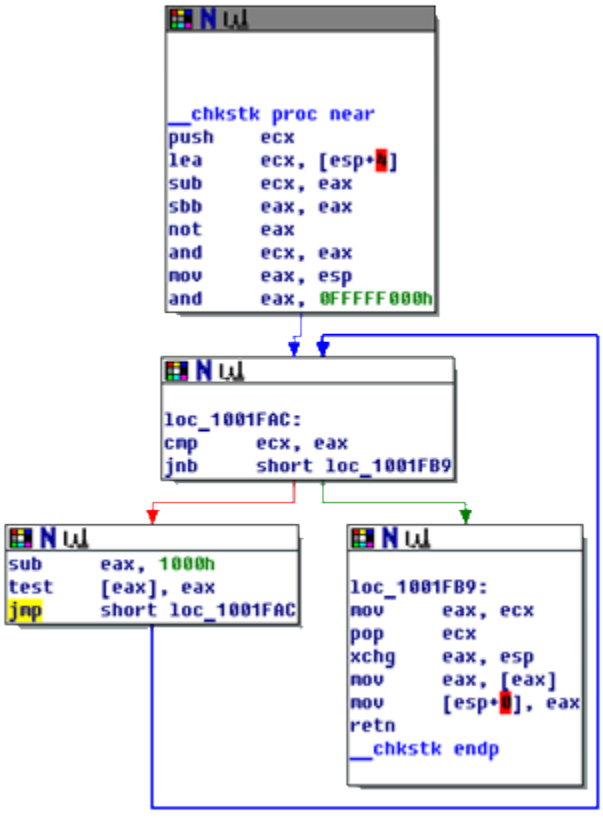

Рисунок 2.6 – Пример графа ассемблера программы Notepad в продукте IDA Pro

Некоторые среды являются достаточно продвинутыми в плане автоматизации обработки кода. В частности, IDA Pro поддерживает скрипты на C-подобном языке (IDA скрипты), позволяющие управлять анализом МК, генерировать информацию о нем и т. д.

Утилиты ассемблирования

Аналогично компиляции, ассемблирование является обратным преобразованием к дизассемблированию. Утилиты ассемблирования считаются стандартными и входят в состав, как всех средств разработки кода, так и множества интерактивных дизассемблеров, позволяющих модифицировать МК. Ассемблеры не изменяют уровень абстракции кода, поскольку лишь преобразуют форму его представления. Минимальное понижение абстракции может быть связано лишь с потерей деления кода на подпрограммы, имен переменных и другой информации, не влияющей на процесс выполнение.

Программные библиотеки разбора машинного кода

Помимо утилит дизассемблирования, для разбора МК возможно применение библиотек с аналогичной функциональностью в составе собственных разрабатываемых ПС. Такие библиотеки позволяют декодировать МК (реже АК) в набор внутренних конструкций, соответствующих каждой процессорной операции и ее операндам, проводить базовый анализ инструкций, иногда выполнять их. По сути, такая библиотека, расширенная функционалом по декодированию и выводу инструкций, представляет собой консольный дизассемблер. Примером для x86/x64 могут быть библиотеки libdisasm [141], Udis86 [142] и др.

Границы применимости средств преобразования машинного кода

Согласно обзору средств, для разбора МК целесообразно применение дизассемблера в виде готовых утилит, поскольку процесс является однозначным, тривиальным и не требующим специализированной разработки. Однако уже в рамках этого процесса возможно восстановление отдельных СМД, таких, как деление ко-

да на подпрограммы и глобальные переменные. Для этих целей удовлетворительным решением может быть дизассемблирование в продукте IDA Pro, использование его автоматических средств анализа и генерация кода выбранных или всех подпрограмм с распознанными глобальными переменными. Восстановленные таким образом метаданные можно располагать в АК путем расширения его синтаксического формата.

2.2.3 Способы описания алгоритмов кода

Графическое описание алгоритмов кода

Графический способ представления алгоритмов использует различные блок-схемы и диаграммы для описания логики работы [56, 58]. Данный способ является «де-факто» и «де-юре» распространенным (например, ГОСТ 19.701-90 «Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения») и даже имеет развитие в виде отдельного стандартизированного языка моделирования – UML [55]. Основным преимуществом способа является наглядность и использование ограниченного набора стандартизованных элементов описания и их связей. Тем не менее, к недостаткам можно отнести большое «размазывание» описания логики алгоритма в пространстве его представления, притом двухмерном, что приводит как к трудности понимания больших алгоритмов, так и невозможности охватить даже средний алгоритм в целом. Также, такое описание хорошо понятно инженерам и недостаточно для разработки кода современными программистам.

Текстовое описание алгоритмов кода

Текстовый способ описание алгоритмов, как правило, с помощью псевдокода, не имеет особенного распространения, в частности, из-за плохого визуального восприятия. Тем не менее, для детального и вдумчивого анализа алгоритма (в особенности понимания его сути) способ представления можно считать подходящим. В качестве аналогии можно представить ИК, в котором убраны все несу-

щественные для анализа алгоритма конструкции (типы, не ключевые вычисления, иногда сгруппированные вызовы подпрограмм и т. п.). Также, такое описание подходит для разработки кода программистами, но плохо воспринимаемо инженерами.

Двумерное описание алгоритмов кода

Крайне интересным (естественно, в рамках рассматриваемых аспектов) является так называемое двумерное описание алгоритмов. Основная идея его заключается в совмещении графического и текстового описания. Целью создания описания является одновременная понятность алгоритмистам (инженерами) и программистам (кодерами). Первые применяют его для работы с алгоритмами, а вторые – для их реализации на языке программирования. Также часто алгоритм описывается в форме, близкой к автомату состояний. Достаточно известным способом, ставшим отдельным языком, можно считать ДРАКОН (Дружелюбный Русский Алгоритмический язык, который Обеспечивает Наглядность); впрочем, некоторые базовые графические элементы языка подобны классическим блок-схемам. Пример программного цикла DO на языке ДРАКОН приведен на рисунке 2.7.

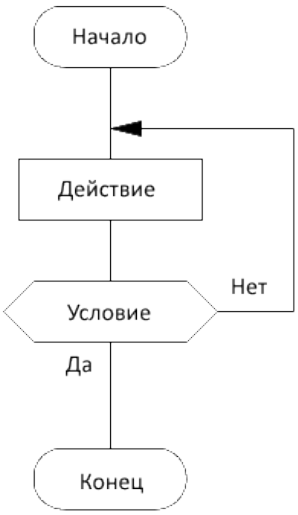

Рисунок 2.7 – Пример цикла DO на языке ДРАКОН

В качестве более сложного примера на языке ДРАКОН можно привести упрощенный алгоритм разработки компилятора, состоящий из двух последовательно-выполняемых этапов: языково-зависимой и независимой (рисунок 2.8).

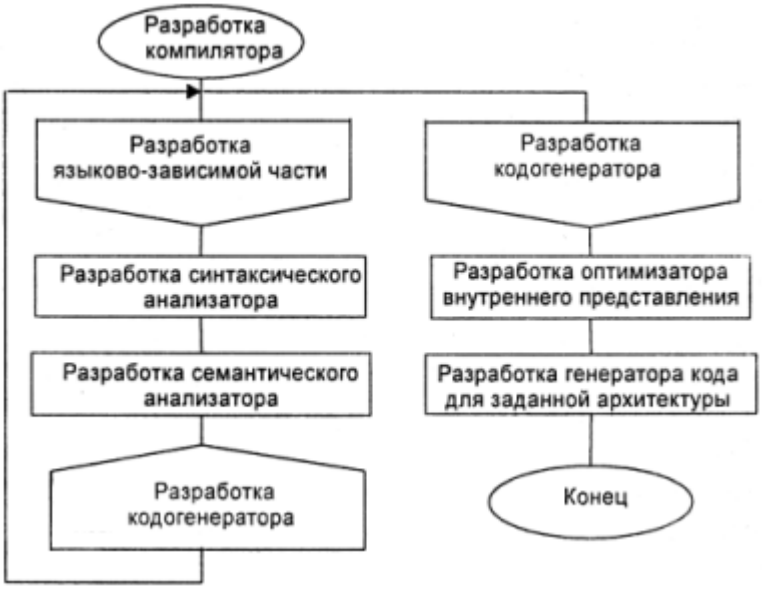

Рисунок 2.8 – Пример алгоритма разработки компилятора на языке ДРАКОН

Основными принципами построения ДРАКОНа является формализация, эргономизация и специальная структуризации алгоритмических блок-схем. Его описание приведено в ГОСТ 19.701-90 и ISO 5807-85.

Границы применимости способов описания алгоритмов

Выбор способа описания алгоритмов произведем исходя из сделанного обзора различных его вариантов и следующих предпосылок. Во-первых, алгоритмизация МК должна применяться, в первую очередь, для поиска уязвимостей, а не получения общего представления о коде для быстрого ознакомления, и, следовательно, допускается описание алгоритмов для длительного изучения. Во-вторых, реальные алгоритмы подпрограмм могут быть достаточно большими, и, следовательно, их графический вид не всегда удобен для понимания. В-третьих, дальнейший поиск уязвимостей осуществляется только для СУ и ВУ, и, следовательно, нет необходимости в чрезмерно-детализированном восстановлении алгоритмов, а именно всех переменных и точных операций над ними. И, в-четвертых, как

уже было подчеркнуто, разработка полноценного и строго удовлетворяющего всем требованиям (субъективным и объективным) формата Представления алгоритмов требует отдельного научного исследования; в рамках же текущей работы предполагается разработка лишь удовлетворительного варианта такого Представления с целью обоснования, как самого его существования, так и проведения оценки эффективности Метода.

2.2.4 Подходы к поиску уязвимостей

Критерии сравнительной оценки

Рассмотрим существующие способы и подходы к поиску уязвимостей с точки зрения их применимости на всех шагах алгоритмизации [76, 143-144]; для оценки и сравнения используем следующие критерии:

1) Время поиска – среднее время, затрачиваемое на поиск основной части уязвимостей данным способом;

2) Количество обнаруживаемых уязвимостей – количество уязвимостей относительно общего их числа, которое может быть найдено данным способом;

3) Ложность результатов – доля уязвимостей, найденных данным способом, которые являются особенностями кода или просто выявлены ошибочно;

4) Покрытие кода – объем обрабатываемого данным способом кода относительно общего объема;

5) Требуемая квалификация использующего способ – уровень подготовки, необходимый для применения данного способа; выделяются 3 следующих уровня:

– пользователь, способный выполнять лишь операции по запуску способа, базовому управлению, сбору результатов и не обладающего знаниями о языках программирования;

– инженер, дополнительно к способностям пользователя умеющий производить настройку способа, базовый анализ результатов с поверхностным знанием языков программирования;

– эксперт, дополнительно к способностям инженера обладающий возможностью *точечного* применения способа, глубокого анализа результатов, хорошо знакомого с языками программирования и особенностями разработки безопасного ПрК.

6) Возможность обойти защиту от обнаружения – наличие у данного способа возможностей по поиску уязвимостей даже в случае защиты ПрК от этого (например, кодирование тела вирусов и их выполнение на внутренней виртуальной машине);

7) Эффективность без ИК – возможность применение данного способа в условиях отсутствия ИК без существенного снижения показателей других связанных критериев;

8) Формализация конечных результатов – возможность использования полученных результатов для автоматической обработки (например, с целью определения дубликатов или обнаружения конфликтов с результатами других способов);

9) Информативность конечных результатов – качественная оценка информации в результатах данного способа, используемая для локализации, проверки и обезвреживания уязвимостей;

10) Основные типы уязвимостей для поиска – список типов уязвимостей, для которых применим данный способ;

11) Особые требования к уязвимостям для поиска – наличие дополнительных особенностей уязвимостей, без которых для их поиска данный способ не подходит.

В таблицах сравнения по критериям для каждого способа используем систему оценки со следующими обозначениями: «+» отражает преимущество по критерию, «–» – отражает недостаток по критерию, а «0» – нейтральность; комментарии в скобках имеют детализирующий и обосновывающий смысл.

«Белый», «черный» и «серый ящики»

В зависимости от доступа к тестируемому объекту выделяют 3 подхода: «*белого*», «*черного*» и «*серого ящика*». В первом случае, вся информация о внут-

реннем строении объекта доступна, во втором – имеется возможность исследования только реакции объекта на внешние воздействия, а в третьем – такая информация об объекте имеется, однако для исследования также используется тестирование воздействием. Применительно к поиску уязвимостей подход «белого ящика» означает, что поиск осуществляется по ПрК, «черного» – исследуется функционирование ПрК при различных внешних условиях и данных, «серого» – также исследуется функционирование ПрК, но для проведения самого поиска и анализа его результатов используется информация о ПрК. Также, первый подход можно поделить на поиск по ИК и МК, однако в рамках задачи алгоритмизации в наличии имеется только последний. Результат сравнительного анализа рассмотренных подходов приведен в таблице 2.1.

Таблица 2.1 – Сравнение подходов «белого», «черного» и «серого» ящика относительно поиска уязвимостей

Критерий сравнения подходов	**Подход «белого ящика» (МК)**	**Подход «черного ящика»**	**Подход «серого ящика»**
Время поиска	0	0	0
Количество обнаруживаемых уязвимостей	0	0	+
Ложность результатов	0	0	0
Покрытие кода	+	0	+
Требуемая квалификация использующего подход	– (эксперт)	+ (пользователь)	0 (инженер)
Возможность обойти защиту от обнаружения	+	–	–
Эффективность без исходного кода	0 (наличие желательно)	+	+
Формализация конечных результатов	0	0	0
Информативность конечных результатов	+	–	+
Основные типы уязвимостей для поиска	0 (НУ, СУ)	0 (НУ, СУ)	0 (НУ, СУ)
Особые требования к уязвимостям для поиска	+ (нет)	– (воспроизводимость при выполнении)	0 (получение информации для воспроизводимости)

Таким образом, с позиции заданных критериев оценки, все подходы можно считать равноценными; хотя и есть незначительный перевес в сторону поиска именно по ПрК («белый» и «серый ящики»).

Фаззинг-подход и тестовые случаи

Суть фаззинг-подхода (fuzzing или fuzz testing) заключается в генерации полностью случайного и частично-специализированного набора данных, подаваемых на вход тестируемого ПС или его модуля с целью выявления программных исключений или другого явного нарушение корректного функционирования [145]. Проверка функционирования осуществляется выполнением отдельного прохода запуска ПС на сгенерированном наборе данных. Подход можно считать удовлетворительными лишь для отдельных типов уязвимостей или условий применения (например, поиск программных исключений путем фаззинга данных в памяти тестируемой программы). Наиболее распространенным применением является тестирование всевозможных разборщиков файлов (браузеров, ассемблеров, компиляторов, других преобразователей данных); впрочем, из-за высоких требований к ресурсам (огромному количеству генерируемых данных и запусков тестируемого ПС) и низкой ценности получаемых результатов (только факт падения с указанием состояния памяти и регистров процессора) подход малоприменим в интересах Метода.

Альтернативой подхода может служить использование так называемых тестовых случаев (test case) – заданного набора условий и действий, согласно которым будет проверяться корректность функционирования ПС [146]. С точки зрения Метода, подход будет заключаться в применении предопределённых шаблонов выявления уязвимостей. Результат сравнительного анализа рассмотренных подходов приведен в таблице 2.2. Результаты сравнительного анализа показывают, что применение тестовых случаев имеет небольшое преимущество по сравнению с фаззингом. Очевидно, что их объединение не позволит получить новый подход, поскольку будет соответствовать применению одного подхода в другом – либо фаззингу со сценариями, либо тестовым случаям с использованием фаззинга.

Таблица 2.2 – Сравнение фаззинг-подхода и тестовых случаев относительно поиска уязвимостей

Критерий сравнения подходов	**Фаззинг-подход**	**Тестовые случаи**
Время поиска	0 (большое количество проходов при незначительном времени выполнения каждого)	0 (при небольшом количестве проходов значительные временные затраты на создание каждого)
Количество обнаруживаемых уязвимостей	– (малое и несложных)	0 (среднее, включая сложные, и зависит от способностей создателя тестовых случаев)
Ложность результатов	+ (все найденные результаты однозначно соответствуют ошибке функционирования)	+ (ложность отсутствует при условии адекватности тестов)
Покрытие кода	- (обрабатывается лишь код, выполнение которого может быть спровоцировано проходом)	0 (обрабатывается лишь код, для выполнения которого создан тестовый случай)
Требуемая квалификация использующего подход	+ (пользователь)	0 (инженер или эксперт)
Возможность обойти защиту от обнаружения	– (подход принципиально не имеет такой возможности)	0 (в ряде случаев возможно)
Эффективность без исходного кода	+ (наличие ИК не влияет)	0 (наличие ИК приводит к более точному созданию тестов)
Формализация конечных результатов	0 (в виде отчетов о запусках и ошибках функционирования, что требует дополнительной обработки)	0 (зависит от конкретной политики тестовых случаев, но в общем случае возможна)
Информативность конечных результатов	0 (только факт программного исключения для входных данных)	0 (факт отличия реального выполнения ПС от требуемого)
Основные типы уязвимостей для поиска	– (малая часть НУ)	+ (НУ, ВУ и СУ)
Особые требования к уязвимостям для поиска	0 (возможность вызвать «грубое» некорректное функционирование – как правило, лишь программное исключение)	0 (наличие четкой спецификации поведения ПС, используемой для сравнения с поведением в процессе тестирования)

Ручной и автоматический способы

Одним из вариантов типизации способов поиска уязвимостей в коде (не только МК) с точки зрения субъективности оценки является их деление на ручные

и автоматические. Суть ручного способа поиска заключается в применении труда экспертов, проводящих анализ кода (ИК, МК, любого другого Представления), а также его субъективной оценки на наличие и местоположение уязвимостей. Суть автоматического способа прямо противоположна ручному и основана на применении автоматических средств поиска без участия человека. Первый способ широко применяется для поиска СУ, а также когда невозможно применение автоматизации. Второй же считается *де-факто* для персональных компьютеров – классическим примером является антивирусное ПО; оно эффективно для НУ, осуществляет поиск по шаблонам распространенных уязвимостей и зачастую содержит эвристические алгоритмы обнаружения. Результат сравнительного анализа рассмотренных способов приведен в таблице 2.3.

Таблица 2.3 – Сравнение ручного и автоматического способов относительно поиска уязвимостей

Критерий сравнения способов	**Ручной способ**	**Автоматический способ**
Время поиска	–	+
Количество обнаруживаемых уязвимостей	+	0 (среднее)
Ложность результатов	+	0 (средняя)
Покрытие кода	0	0
Требуемая квалификация использующего подход	– (эксперт)	+ (инженер)
Возможность обойти защиту от обнаружения	+	0 (иногда и частично)
Эффективность без исходного кода	0 (наличие желательно)	+
Формализация конечных результатов	0 (возможна)	+
Информативность конечных результатов	+ (высокая)	+ (высокая)
Основные типы уязвимостей для поиска	+ (НУ, СУ)	0 (НУ)
Особые требования к уязвимостям для поиска	+ (нет)	– (шаблонное совпадение)

Как видно из таблицы, у обоих подходов есть сильные и слабые стороны; объединение же их достоинств тождественно частичной автоматизации поиска.

Статический и динамический способы

Другим вариантом типизации способов поиска уязвимостей в коде (не только МК) с точки зрения оценки эффектов тестируемого кода является их деление на статические [147-149] и динамические [150]. Суть первого заключается в анализе ПрК в статическом виде – т. е. без его непосредственного выполнения. Суть второго заключается в выполнении кода в различных Представлениях и различными способами: ИК, МК на реальном оборудовании, в эмуляторах и т. п. Как и в случае типизации по субъективности оценки, каждый из способов имеет собственные сильные и слабые стороны. Результат сравнительного анализа рассмотренных способов приведен в таблице 2.4.

Таблица 2.4 – Сравнение статического и динамического способов относительно поиска уязвимостей

Критерий сравнения способов	**Статический способ**	**Динамический способ**
Время поиска	–	0
Количество обнаруживаемых уязвимостей	0 (среднее)	+
Ложность результатов	– (средняя)	0 (средняя)
Покрытие кода	+	0 (только тот, вызов которого можно спровоцировать)
Требуемая квалификация использующего подход	0 (эксперт)	0 (эксперт)
Возможность обойти защиту от обнаружения	+	0 (иногда и частично)
Эффективность без исходного кода	0 (наличие желательно)	0 (наличие желательно)
Формализация конечных результатов	+	0 (возможна)
Информативность конечных результатов	+ (высокая)	+ (высокая)
Основные типы уязвимостей для поиска	0 (НУ, СУ)	0 (НУ, СУ)
Особые требования к уязвимостям для поиска	+ (нет)	– (воспроизводимость при выполнении)

Объединение достоинств подходов возможно в так называемом подходе Symbolic execution [151] – наиболее подходящим названием его могло бы стать «статическое выполнение кода», основанном на совокупном анализе не только графов вызовов функций и потока управления, но и всех возможных значений переменных и состояния памяти для каждой ветки выполнения УС из всего их множества. Такой подход далее не рассматривается, как выходящий за рамки предметной области.

Границы применимости существующих способов

Скомпонуем наиболее подходящий для алгоритмизации способ, исходя из рейтинга всех рассмотренных. С учетом результатов сравнительного анализа, наиболее подходящим представляется сочетание подходов поиска уязвимостей по МК автоматизированным способом (как по единичным, так и целым наборам шаблонов), преимущественно статическим анализом белого и/или серого ящика, по возможности без прямого выполнения кода.

2.2.5 Возможности для построения, обработки и оптимизации моделей машинного кода

Создание модели машинного кода

Модель, являясь абстрактным представлением реального объекта, исследование которой позволяет получать необходимую информацию, хорошо сопоставима с внутренним представлением ПС, производящего разбор входных данных и их обработку. ПС может использовать это представление для выполнения сложных преобразований или генерации выходных данных. С этой позиции МК соответствует входным данным, его Модель – внутреннему представлению этих данных в ПС, а алгоритмизированное Представление аналогично результатам работы ПС. При этом нет принципиальной разницы между бинарными данными МК или текстом АК, поскольку они взаимно-преобразуемы и в конечном итоге будет получена их тождественная Модель.

Типичное построение внутреннего представления в упомянутых ПС реализуется с помощью модулей лексического, синтаксического и семантического анализатора. Опишем модули и возможности их реализации в интересах Метода более подробно.

Лексический анализатор предназначен для разбора последовательности входных символов на *лексемы* – минимальные единицы лексического анализа. Такой разбор можно производить как с помощью кода, разработанного полностью вручную, так и с применением специальных генераторов лексических анализаторов на основании заданных правил. Одним из распространенных генераторов является утилита Flex [152], создающая код на языке C, который преобразует входной поток символов в последовательность лексем согласно шаблонам, заданным регулярными выражениями. В интересах Модели, такой анализатор может производить разбиение бинарных данных МК или текста АК на соответствующие лексемы.

Синтаксический анализатор предназначен для сопоставления последовательности входных лексем с формальной грамматикой и выполнением при этом соответствующих правил, конкретный ПрК которых может быть написан вручную. Такой разбор, по причине сложности, как правило, осуществляется кодом, созданным генератором синтаксических анализаторов по грамматике на специальном языке. Одним из распространенных генераторов является утилита Bison [153], создающая код на языке C, который разбирает последовательность входных лексем и выполняет заданный в правилах код на том же языке. В интересах Метода, такой анализатор может собирать последовательность лексем из разобранных бинарных данных МК или текста АК в соответствующие правила, сопоставляемыми с действиями процессора выполнения.

По мере работы синтаксического анализатора, соответствие правил входным лексемам – их свертка – приводит к выполнению заданного кода, который может соответствующим образом строить дерево его внутреннего представления, тем самым структурируя нужным образом входной поток лексем и правил. Например, при свертке правила всего АК будет построен верхний узел дерева,

при свертке правила операнда инструкции (например, регистра) – элемент дерева, соответствующий регистру, а при свертке правила всей инструкции – поддерево операции инструкции с уже созданными узлами операндов. В интересах Метода, код правил может строить соответствующие ветки дерева – поддеревья, определяющие действия для выполнения (арифметические операции, вызов подпрограмм, доступ к глобальным переменным и т. п.). Полное дерево определит первоначальный вид Модели. При этом, в связи с тривиальностью как МК, так и АК, внутреннее дерево можно сделать полностью инвариантным как от входного синтаксиса данных, так и от процессора выполнения.

Семантический анализатор предназначен для установления смысла собранных синтаксических правил по построенному дереву. В результате могут быть добавлены дополнительные деревья и таблицы (например, информация о сложных типах переменных в случае разбора ИК). Как правило, такие анализаторы разрабатываются вручную по причине сложности формализации семантического понятия *смысл*. Семантический анализ при построении Модели является простейшим и может быть упразднен.

Проведение эксперимента на модели машинного кода

Проведение модельного эксперимента в рамках Метода означает построение алгоритмизированного представления по частной S-модели в виде описания, подходящего для ручного анализа. Таким образом, необходима как обработка внутреннего представления модели, так и создание по ней нового. С точки зрения подходов программирования, такое превращение можно считать классическим преобразованием дерева внутреннего представления входного кода и генерацию выходного, типичного для компиляторов. Для этой задачи возможно как применение отдельных библиотек, так и разработка собственного языка преобразования графов.

Если код в L-модели представляется в линейном виде, то в Представлении S-модели он будет структурированным. Код будет состоять из архитектуры, набора модулей, подпрограмм и алгоритмов, по возможности не имеющих без-

условных переходов между собой. Такое Представление подобно графической диаграмме Насси-Шнейдермана, получившей широкое распространение в ряде стран (и имеющий официальный стандарт DIN 66261 [154], разработанный Немецким институтом по стандартизации). Получаемый вид позволяет применять к ней «математическую теорию графов» и «теоретическую информатику».

По S-модели возможна аппроксимация метаданных ИК, потерянных в процессе компиляции и ассемблирования, например, следующим образом. Первое использование значения любого регистра в подпрограмме без явной инициализации, как правило, означает, что регистр используется в качестве ее аргумента. Последняя инициализация регистра значением без явного последующего использования, как правило, означает, что регистр используется в качестве возвращаемого значения функции. Переход на точку кода, находящуюся ранее выполняемой, как правило, означает наличие цикла в алгоритме подпрограммы.

Оптимизация модели машинного кода

Суть любой оптимизации заключается в поиске наилучшего варианта среди множества в заданных условиях. Применительно к S-модели, это означает построение такого образа конкретного экземпляра МК, который позволил бы восстановить архитектуру и алгоритмы работы в виде, наиболее подходящем для ручного поиска уязвимостей. Поскольку объективные оценки «подходимости» восстановленных алгоритмов практически отсутствуют, то имеет смысл оценивать удовлетворительность результата восстановления вручную с необходимой корректировкой модели – т. е. проведение *оптимизации модели*.

Пусть состояние модели определяется совокупностью *показателей* $X = \{x_1, x_2, \ldots, x_n\}$. Тогда в рамках математического смысла оптимизации можно ввести функцию $F(X)$, называемую критерием эффективности. Будем считать, что чем б*о*льше значение $F(X)$, тем *лучше* состояние модели. Тогда, задача оптимизации заключается в нахождении такого значения X_0, при котором $F_0 = F(X_0)$ будет максимально. Вид функции $F(X)$, как и значение критерия эффективности, определяется субъективно производящим алгоритмизацию, однако показатели эффектив-

ности могут быть получены достаточно объективными рассуждениями. Показатели состоят из следующих: уровень детализации алгоритмов; уровень абстракции алгоритмов; количество выделенных СМД, в частности, сложных управляющих структур (циклов, вложенных условий); качество выделенных СМД, в частности, корректность определения сигнатур подпрограмм; качество структуризации, в частности количество оставшихся операторов безусловного перехода GOTO; количество и качество обнаруженных в процессе моделирования потенциальных уязвимостей; удобство восприятия кода.

Отметим, что недостаточно оптимальное построение S-модели будет связано не столько с ошибками в моделировании, сколько с невозможностью ее однозначного построения (в особенности автоматического) без субъективной оценки результатов моделирования. Так, например, точное определение сигнатуры подпрограммы, состоящей из ее входных параметров, может быть произведено только в результате ручного анализа логики работы алгоритмов других подпрограмм, используемых ее.

Реализацию процесса оптимизации возможно осуществить путем субъективной оценки полученных результатов моделирования по каждому показателю и расчета критерия эффективности Модели, внесения корректировочных метаданных в ассемблерное представление МК и повторения процесса моделирования. Итеративность такого подхода позволит осуществить постепенное приближение Модели к требуемой эффективности.

2.3 Этапизация метода алгоритмизации машинного кода

Осуществим поэтапный синтез метода алгоритмизации машинного кода для поиска уязвимостей путем группировки шагов его гипотетической Схемы. Выбор реализаций этапов осуществим на основании рассмотренных подходов и способов, частично или полностью применимых для решения задачи, и средств – существующих или необходимых для разработки; в том числе, будем использовать выявленное сочетание подходов поиска уязвимостей. Также следуя аналогии с кри-

терием сравнительной оценки способов и подходов к поиску уязвимостей (см. пункт 2.2.5) введем следующие уровни квалификации:

– инженер Метода, умеющий производить выполнение Метода и базовый анализ его результатов с поверхностным знанием языка С (далее – Инженер-М);

– эксперт Метода, обладающий возможностью глубокого анализа результатов Метода, имеющий опыт программирования на языке С (и, таким образом, понимаемый алгоритмизированное представление), а так же разбирающегося в особенностях разработки безопасного ПрК (далее – Эксперт-М).

На первом этапе производится дизассемблирование МК и получение его АК средствами продукта IDA Pro с применением скриптов. Синтаксис АК расширен для получения возможности внесения будущих корректировок в процесс алгоритмизации. На втором этапе полученный АК алгоритмизируется неким ПС, последовательно выполняющим построение внутреннего представления АК, обработку Представления со сбором и систематизацией информации об СМД и уязвимостях, а также создание внутреннего представления алгоритмов и его генерацию в текстовом виде. На третьем этапе текстовое описание алгоритмов анализируется Экспертом-М на предмет адекватности и удовлетворительности для последующего поиска уязвимостей. На четвертом этапе в случае недостаточно подходящего описания алгоритмов, Эксперт-М производит корректировки процесса алгоритмизации, добавляя их в расширенный синтаксис АК, и повторяет запуск разработанного ПС алгоритмизации. В случае удовлетворительности описания алгоритмов производится их ручное взаимное согласование Инженером-М, т. е. доведение до вида, подходящего Эксперту-М для будущего способа поиска уязвимостей по алгоритмизированному Представлению МК. Результат деления шагов Схемы Метода на этапы и выбора их реализаций приведен в таблице 2.5.

Опишем этапы Метода более детально [60, 69, 70, 72, 87].

Таблица 2.5 – Соответствие шагов гипотетической схемы реализациям этапов метода алгоритмизации машинного кода

Шаги	**Этап**	**Реализация**		
		Подходы, способы	**Средства**	
1. Распаковка МК	1. Получение ассемблерного представления	Автоматический анализ Статический анализ «Белый ящик» Дизассемблирование в интерактивной среде	IDA Pro + IDA скрипты	
2. Подготовка МК				
3. Разбор МК и корректировок к нему	2. Алгоритмизация ассемблерного представления	Автоматический анализ Статический анализ Создание/Обработка модели «Серый ящик» (условно) (Де)Компиляция Библиотеки разбора МК Шаблоны единичных уязвимостей и целых групп	Новое ПС	Построение внутреннего представления
4. Сбор структурных метаданных				
5. Сбор информации об уязвимостях				Обработка внутреннего представления
6. Построение частной S-модели				
7. Анализ S-модели				
8. Синтез алгоритмизированного Представления				Создание текстового описания архитектуры и алгоритмов
9. Генерация алгоритмизированного Представления				
10. Анализ алгоритмизированного Представления	3. Анализ алгоритмизированного Представления АК	Ручной анализ Статический анализ «Белый ящик» Текстовое описание алгоритмов	Ручной анализ алгоритмов Экспертом-М	
11. Проверка S-модели. Удовлетворительна?				
12. Создание корректировок для моделирования	4. Корректировка алгоритмизации АК	Ручной анализ Статический анализ Оптимизация модели «Белый ящик»	Ручной синтез корректировок алгоритмов Экспертом-М	
13. Гармонизация алгоритмизированного Представления	5. Гармонизация алгоритмизированного Представления АК	Ручной анализ	Ручное согласование восстановленной архитектуры и алгоритмов Инженером-М	

2.3.1 Этап 1. Получение ассемблерного Представления

Этап является подготовительным, так как его назначение заключается в преобразовании начального вида МК к пригодному для обработки автоматизированным средством на следующем этапе.

Изначально МК представляет собой линейный набор бинарной информации (впрочем, иногда поделенной на секции кода и данных). Такой вид, в отличие от текстового, абсолютно не подходит для ручной обработки, а создание средств его алгоритмизации из-за этого будет усложнена. И хотя первоначальный этап любого средства, обрабатывающего на входе формализованную бинарную информацию, не представляет особой сложности (разборщик бинарного кода проще аналогичного текстового), но дальнейшая разработка и отладка такого приложения будет несоизмеримо сложнее. Это следует хотя бы из того, что для написания и проверки тестовых примеров, скорее всего, придется применять отдельный этап – ассемблирование текстового АК в бинарный МК. Таким образом, целесообразно на первом этапе Метода получить текстовое ассемблерное Представление МК, используемое как входное для последующих этапов – визуально-понятное и подходящее для создания тестов.

Выполнение этапа может быть осуществлено с помощью существующих утилит – дизассемблеров, поскольку назначение последних полностью совпадает с предназначением этапа. Для задания соответствия формата АК с входным форматом последующего этапа был выбран хорошо распространенный и отлаженный продукт – дизассемблер IDA Pro. Продукт является безусловным лидером в своей области и, помимо полноценного решения задачи этапа, обладает следующими преимуществами: поддержка большого числа процессоров; автоматическое выделение сегментов кода данных, функций (в том числе библиотечных, заданных сигнатурами), меток и переходов внутри функций; автоматическое комментирование инструкций; интерактивная работа с кодом (позволяющая корректировать процесс дизассемблирования вручную).

Таким образом, для преобразования бинарного кода на первом этапе Метода рациональным решением является не написание своего дизассемблера, а использования достаточно мощного продукта IDA Pro, расширяющего эффективность выполнения этапа дополнительными возможностями.

Для автоматизации процесса преобразования Представления можно использовать скрипт, исполняемый в указанном продукте и генерирующий АК в необходимом формате. Также, скрипт может дополнять АК метаинформацией (т. е. расширять его синтаксис), созданной IDA Pro и которая будет использована для корректировки алгоритмизации на последующем этапе Метода. Реализация скрипта не зависит от типа процессора МК, поскольку продукт поддерживает дизассемблирование и обработку кода для их большого количества. ИК IDC-скрипта приведен в Приложении А. Таким образом, этап можно считать практически независимым от исследуемого МК.

2.3.2 Этап 2. Алгоритмизация ассемблерного Представления

Этап является основным в Методе, поскольку остальные можно считать лишь подготовительными и заключительными. Этап преобразует текстовый АК, имеющий линейный и плохо-анализируемый вид, в алгоритмизированное Представление, подходящее для ручного анализа Экспертом-М.

Для конечного вида алгоритмизированного Представления были выдвинуты следующие требования. Во-первых, он должен быть близок к синтаксису современных высокоуровневых языков программирования (таких как языки C и C++), поскольку они являются популярными, хорошо воспринимаемыми программистами и предназначенными для эффективной разработки ПрК. Во-вторых, вид должен быть адаптирован для описания алгоритмов ПрК. И, в-третьих, в нем должна иметься возможность указания информации о потенциальных уязвимостях. Дополнительным требованием, логично следующим из первых трех, является «заточенность» конечного вида именно для передачи сути алгоритмов (а не последующей компиляции, или, например, анализа сущностей ИК [155]), а значит, в

нем могут отсутствовать несущественные особенности реальных языков программирования, детальное соблюдение их семантик, точное следование парадигмам программирования или особенностям реализации компиляторов.

Этап может быть реализован с помощью специально разработанного ПС, имеющего 3 последовательно-выполняемые фазы, и поэтому считается автоматическим.

2.3.3 Этап 3. Анализ алгоритмизированного представления ассемблерного кода

Этап выполняется Экспертом-М ручным способом; он, используя алгоритмизированное Представление из предыдущего этапа, проверяет код на корректность восстановления алгоритмов, их воспринимаемость и применимость для будущего поиска уязвимостей. В случае неудовлетворительности алгоритмизации (например, алгоритмы получились слишком детализированными или наоборот – расплывчатыми) Эксперт-М может добавить корректировки для повторной алгоритмизации, уточняющие или изменяющие процесс. Так, он может уже на этом этапе понять суть работы отдельных алгоритмов и дать им собственные понятные названия или указать аргументы функций, которые будут учтены при следующей итерации моделирования.

В качестве параметров оценки удовлетворительности кода, Эксперт-М может использовать субъективные параметры оценки его алгоритмизированности. Использование объективных параметров оценки в данном случае неоправданно, поскольку назначением алгоритмизированного Представления является дальнейший ручной анализ (как самим Экспертом-М, так и отдельной группой специалистов, близкой с ним по уровню подготовленности и применяемым подходам); следовательно, субъективность оценки выходит на первое место.

2.3.4 Этап 4. Корректировка алгоритмизации ассемблерного кода

На данном этапе Эксперт-М изменяет изначальный АК, внося в его расширенные синтаксис корректировки к алгоритмизации в формализованном виде и повторно запускает Этап 2. Решение о новой итерации принимается Экспертом-М субъективно, поскольку объективных условий для этого практически не существуют. Теоретически, итерации могут принять вид бесконечного цикла, выход из которого путем их прекращения осуществляется также по решению Эксперта-М. Тем не менее, каждая новая итерация носит уточняющий характер и позволяет улучшить описание алгоритмизированного Представления или же оставляет его без изменений, а, следовательно, определение зацикленности Эксперту-М не составит труда и зависит лишь от его квалификации.

В качестве примеров корректировок можно привести следующие, логично вытекающие из требований к «улучшению» алгоритмизации и возможностей реализации:

– присвоение распространенных или интуитивно-понятных названий именам подпрограмм и переменных;

– уточнение сигнатур подпрограмм, таких, как входные и выходные регистры;

– замена тела подпрограммы на разъясняющий комментарий;

– изменение стиля генерации описания отдельных подпрограмм;

– явное задание глобальных переменных (с указанием адреса, имени и размера), сложных типов данных (таких, как структуры) и некоторых деталей типов (например, границ массивов или размеров его элементов);

– задание сложных циклов (не достаточно точно распознанных на Этапе 2), конструкций множественного выбора по условию (SWITCH), специфических подпрограмм (например, функций выделения памяти) и специфических данных (например, массивов, хранящих пароли, с доступом только на чтение), целевых уязвимостей, и др.

Таким образом, корректировка позволит не только повысить читаемость алгоритмов кода Экспертом-М, но и дать дополнительную информацию для более эффективного обнаружения уязвимостей.

2.3.5 Этап 5. Гармонизация алгоритмизированного представления ассемблерного кода

Описания алгоритмов, полученные на Этапе 2 и проанализированные на Этапе 3 уже подходят для поиска уязвимостей; тем не менее, для увеличения эффективности поиска целесообразна их гармонизация, заключающаяся в выполнении следующих действий Инженером-М (т. е. человеком, не обладающим глубоким уровнем аналитики и знания предметной области):

– группировка переменных и алгоритмов, которая улучшит навигацию по тексту и его восприятие;

– сортировка переменных, которая улучшит восприятие алгоритмов;

– замена содержания отдельных алгоритмов (как целиком, так и частей) текстом, который упростит их описание и улучшит понимание;

– задание названий подпрограмм для тривиальных алгоритмов, которое упростит понимание других подпрограмм, их вызывающих;

– выборочная замена значений переменных и бинарных массивов на более подходящие (например, на шестнадцатеричный вид, булевские значения или текстовые строки), которая позволит оперировать видом значений, наилучшим образом соответствующим содержанию;

– массовое отсечение объективно неверной информации о потенциальных уязвимостях и др.

Таким образом, данный этап Метода является заключительным, предоставляя исследуемый МК в алгоритмизированном виде, готовом для дальнейшего поиска уязвимостей.

Выводы по разделу 2

В качестве объекта исследовалась структурная модель машинного кода на предмет создания его алгоритмизированного представления в форме, подходящей для анализа специалистом.

В ходе исследования:

1) С использованием концептуального подхода из пункта 1.3.4 построена гипотетическая схема алгоритмизации машинного кода, которая состоит из 13-ти шагов и имеет итеративное выполнение. Особенностью схемы является ее инвариантность к последующей реализации метода.

2) Описаны общие принципы поддержки структурной модели на основании особенностей и условий ее применения в гипотетической схеме алгоритмизации. Показана работоспособность модели на гипотетическом примере ее построения и преобразования.

3) Выделены существенные аспекты алгоритмизации машинного кода в интересах реализации ее схемы, а именно: близкие подходы, преобразование машинного кода, описание алгоритмов, поиск уязвимостей, моделирование и модельный эксперимент.

4) Произведена этапизация метода алгоритмизации путем группирования шагов ее схемы в следующие пять этапов: первый – получение ассемблерного представления, второй – алгоритмизация ассемблерного представления, третий – анализ алгоритмизированного представления ассемблерного кода, четвертый – корректировка алгоритмизации ассемблерного кода и пятый – гармонизация алгоритмизированного представления ассемблерного кода.

5) Определены средства, необходимые для реализации этапов метода, а именно: автоматическая обработка машинного кода интерактивным дизассемблером IDA Pro и ассемблерного кода специальным программным средством алгоритмизации, а так же ручной анализ алгоритмизированного представления экспертом.

6) Заданы требования к новому программному средству алгоритмизации и следующий 3-х фазный процесс его работы: построение внутреннего представления, обработка внутреннего представления и создание текстового описания архитектуры и алгоритмов.

Основным научным результатом, изложенным во втором разделе, является метод алгоритмизации машинного кода, суть которого заключается в моделировании экземпляра машинного кода и построении его алгоритмизированного представления.

Частными научными результатами, изложенными во втором разделе, являются критерии и результаты сравнительной оценки существующих способов и подходов к поиску уязвимостей с точки зрения их применимости для метода алгоритмизации. Эти результаты непосредственно были использованы для составления требований к реализации программного средства алгоритмизации.

Основное содержание раздела и полученных научных результатов изложено в работах автора [60, 69, 70, 72, 76, 87].

3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА

3.1 Разработка функциональной архитектуры Утилиты

Из раздела 2 следует, что ключевым этапом Метода, влияющим на успешность получения необходимых результатов, является «Этап 2. Алгоритмизация ассемблерного кода», реализуемый разработкой нового ПС. Такая утилита алгоритмизации (далее – Утилиты) и была разработана в рамках исследовательской работы в интересах проверки работоспособности Метода [62, 69, 72].

Согласно подразделу 2.3 работа ПС алгоритмизации состоит из следующих фаз (последовательно выполняющих основной функционал средства), имеющих в области разработки компиляторов следующие англоязычные названия: Front-End, Middle-End и Back-End. В первой фазе осуществляется разбор АК – используя грамматику и выстраивая его внутреннее представление. Для функционирования Front-End потребуется реализация собственного лексического анализатора, разработка формальной грамматики расширенного ассемблера, реализация правил грамматики для построения дерева абстрактного синтаксиса и алгоритмов преобразования дерева в платформенно-независимое внутреннее Представление. Во второй фазе производится обработка построенных деревьев, выделение СМД, сбор информации об уязвимостях и построение S-модели (с помощью набора графов, таблиц, хэшей, дополнительных деревьев и т. п.). Также возможно восстановление потерянных СМД на основании распространенных практик программирования; основная масса заданных корректировок моделирования учитывается именно здесь. Для функционирования Middle-End потребуется реализация собственных алгоритмов обхода графов и поиск соответствия их элементов шаблонам уязвимостей. В третьей фазе создается алгоритмизированное Представление АК (подобно [156-157]), преобразованное в текстовый вид согласно выходному синтаксису. При наличии информации, собранной в Middle-End, построение такого Представления и его генерация являются технологически тривиальными.

Поскольку очевидна близость цели и функционала Утилиты к подходам, используемым при компиляции и декомпиляции, то и ее архитектуру целесообразно сделать аналогичной. При этом основополагающим элементом в архитектуре считается S-модель, вокруг которой будет строиться остальной функционал. Утилита может иметь вид консольного приложения, поскольку все взаимодействия в ней пользователя сведены до предоставления входных данных и анализа выходных. Для отладки Утилиты достаточно использование отладочного вывода ее внутренних представлений. Принцип работы Утилиты может быть сведен к последовательному (пофазному) выполнению действий согласно этапу Метода (в отличие, например, от приложений с графическим интерфейсом, имеющим принцип работы, связанный с обменом сообщениями или переходам между внутренними состояниями). В данной Утилите отсутствует прямая необходимость обмена данными с внешними объектами (другими ПС и базами данных). S-модель в такой архитектуре задается совокупностью различных структур – внутренних (промежуточных) данных Утилиты, на которых определены алгоритмы модулей Утилиты и которые задают Представление входного АК. Предлагаемая функциональная архитектуры Утилиты в схематичном виде показана на рисунке 3.1 [78].

Архитектурные элементы

На рисунке 3.1. присутствуют следующие архитектурные элементы. Во-первых, это вышерассмотренные фазы Утилиты. Во-вторых, это модули архитектуры и их взаимодействия. В-третьих, это входные и выходные данные Утилиты. И, в-четвертых, это внутренние данные Утилиты, определяющие S-модель. Как хорошо видно, архитектура представляет собой пофазное многостадийное выполнение преобразования АК в его алгоритмизированное Представление. Каждая стадия состоит из модулей, реализующих ее функционал. Модули производят преобразования внутренних данных, используемых также и в качестве способа обмена между ними. Совокупность таких данные образует внутреннее представление S-модели, которая строится на первой стадии по МК и обрабатывается на всех последующих; притом последняя стадия производит генерацию алгоритмизированного Представления в виде алгоритмов входного АК с информацией о потенциальных уязвимостях.

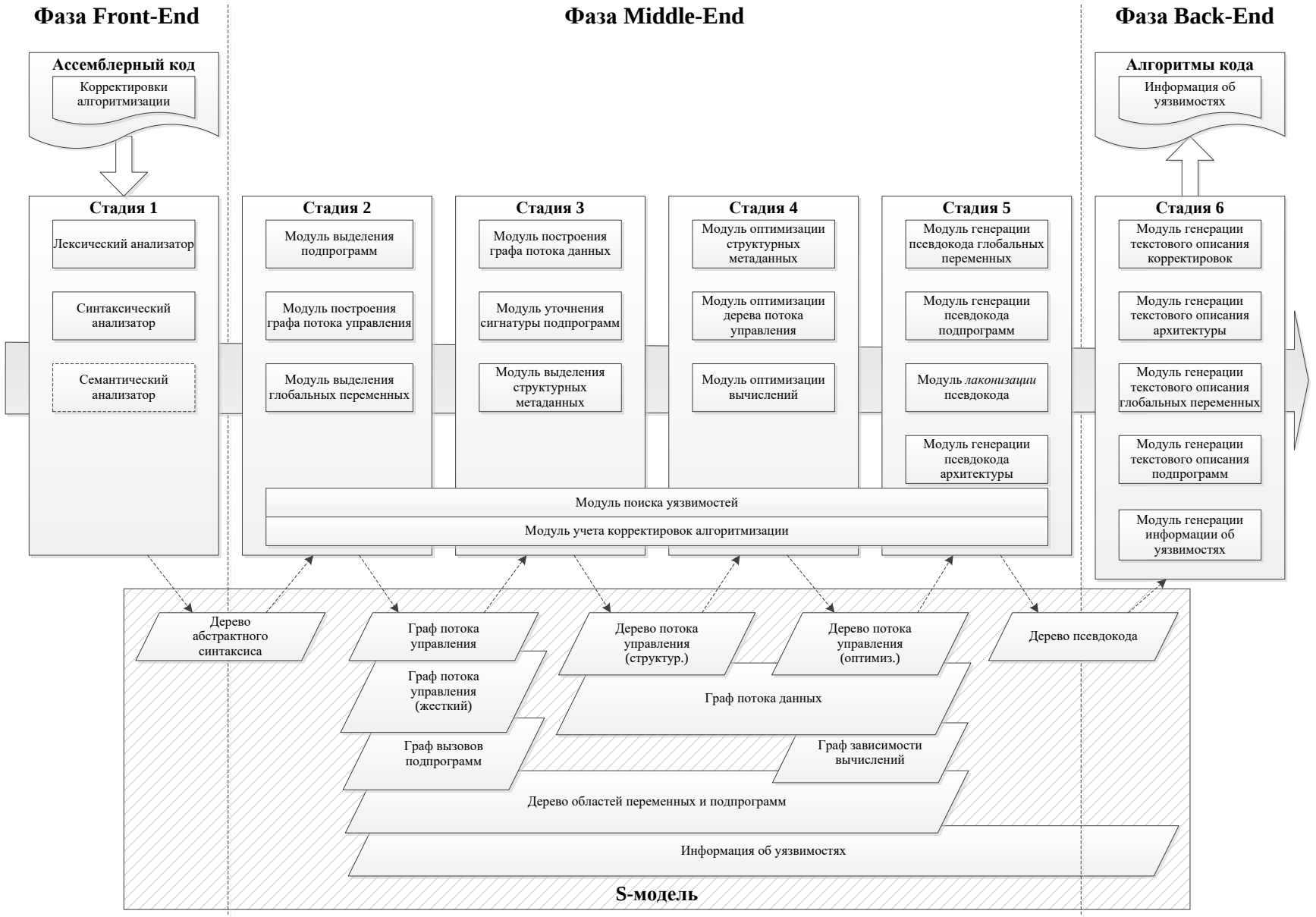

Рисунок 3.1 – Функциональная архитектура Утилиты (программного средства алгоритмизации)

Все модули архитектуры представляют собой независимые единицы выполнения. Обмен между модулями на различных стадиях работы Утилиты осуществляется посредством внутренних представлений Утилиты (подобно [158-159]); таким образом, каждая совокупность модулей одной стадии выполняет четко заданный набор действий по созданию нового или преобразования старого представления.

Взаимодействие модулей по стадиям выглядит следующим образом: модули Стадии 1 принимают на вход АК и создают дерево абстрактного синтаксиса; модули Стадии 2 принимают на вход дерево абстрактного синтаксиса (от англ. Abstract Syntax Tree или AST) [160] и создают графы потока управления (от англ. Control Flow Graph или CFG) и граф вызовов подпрограмм (от англ. Call Subroutine Graph или CSG), заполняя дерево областей переменных и подпрограмм (от англ. Scope Tree или SCT); модули Стадии 3 принимают на вход CFG и, обновляя SCT, создают граф потока данных (от англ. Data Flow Graph или DFG) и дерево потока управления (от англ. Control Flow Tree или CFT); модули Стадии 4 оптимизируют это дерево, используя SCT; модули Стадии 5 строят по нему дерево псевдокода (от англ. Pseudo Code Tree или PCT), которое используется модулями Стадии 6 для генерации текстового описания алгоритмов АК. Также, на Стадии 4 строится граф зависимости вычислений (на основании используемых в них общих переменных), используемый на этой же стадии для оптимизации.

Прямое взаимодействие модулей между стадиями практически отсутствует, поскольку каждый из таких модулей реализует собственный ограниченный функционал над входным в стадию или выходным из нее представлением данных Утилиты (графом или деревом). Такая организация модулей позволяет, в частности, без существенных трудозатрат добавлять новый функционал для оптимизации и лаконизации деревьев. Последняя, в отличие от оптимизации, призвана улучшить субъективные особенности кода, такие, как: компактность и воспринимаемость человеком за счет использования в конечном Представлении алгоритмов более компактных выражений, специальных языковых конструкций, именованных адресов и т. п. Исключением являются два модуля – учет корректировок алгоритми-

зации и поиск уязвимостей, поскольку они предоставляют данные и собственный функционал всем остальным модулям, не зависимо от стадии выполнения последних.

Стадии работы Утилиты, реализующие их модули, назначение, выполняемые функции и гипотетические примеры входных и выходных данных последних, а также специальные требования к реализации приведены в Приложении Б.

Опишем возможности настройки алгоритмизации МК Утилитой и обобщенный пример ее использования. Эта совокупность особенностей работы Утилиты определит базовый сценарий ее функционирования.

Настройка алгоритмизации

Принцип выполнения Утилиты, аналогично компиляторам в начале своего исторического появления, можно считать условно-однопроходным, поскольку все стадии выполняются последовательно (и, следовательно, единожды), работая с постепенно преобразуемым Представлением ПрК – от ассемблерного (текстового) через графовидное (бинарное) к алгоритмизированному (текстовому). Каждая стадия Утилиты производит собственные изменения во внутреннем представлении данных и, опционально, выводит отладочный результат своей работы.

Глобальные настройки, влияющие как на процесс выполнения Утилиты, так и на стиль генерируемого алгоритмизированного Представления, хранятся в отдельном объекте и состоят из следующих Флагов:

Флаг 1. Генерация отладочного вывода для каждого из промежуточных вариантов внутренних представлений между стадиями;

Флаг 2. Добавление в алгоритмизированное Представление комментариев, описывающих возникшие предупреждения, результаты работы оптимизации и лаконизации;

Флаг 3. Добавление в алгоритмизированное Представление комментариев, содержащих информацию о потенциальных уязвимостях;

Флаг 4. Удаление из алгоритмизированного Представления пустых веток для условных ветвлений («if(…)then{…}else{}»->« if(…)then{…}»);

Флаг 5. Уплотнение алгоритмизированного Представления путем удаления пробелов вокруг бинарных операций («X + Y» -> «X+Y»);

Флаг 6. Уплотнение алгоритмизированного Представления путем удаления пробелов вокруг управляющих конструкций условных ветвлений («if (…) then {…} else {…}» -> «if(…)then{…}else{…}»);

Флаг 7. Замена разыменования указателя на доступ к нулевому элементу массива(«*X» -> «X[0]»);

Флаг 8. Указание для входных и выходных переменных подпрограммы используемых процессорных регистров («FUNCT(X) { return X; }» -> «FUNCT(X:r3) { return X:r3; }»);

Флаг 9. Указание для подпрограммы регистра, используемого для хранения точки возврата («FUNCT()» -> «FUNCT(): lr»; регистр LR для процессоров PowerPC хранит точку возврата из подпрограммы);

Флаг 10. Указание возвращаемых регистров подпрограммы-функции («FUNCT()» -> «(X, Y, Z) FUNCT()»).

Обобщенный пример использования

Упрощенная схема основной ветки выполнения Утилиты с отладочным выводом показана на рисунке 3.2. Таким образом, общий процесс выполнения Утилиты является автономным, настраиваемым и отлаживаемым.

Обобщенный пример алгоритмизации с помощью Утилиты может быть описан следующими шагами, предпринимаемыми пользователем.

Шаг 1 – Начать выполнение. Пользователь выбирает МК, для которого необходимо произвести алгоритмизацию. Также подготавливаются для использования продукт IDA Pro и сама Утилита. Для последней задаются глобальные настройки с помощью приведенных ранее флагов.

Шаг 2 – Получить АК. С помощью продукта IDA Pro производится дизассемблирование МК, а с помощью специального разработанного скрипта генерируется его ассемблерное представление, соответствующее входному синтаксису Утилиты.

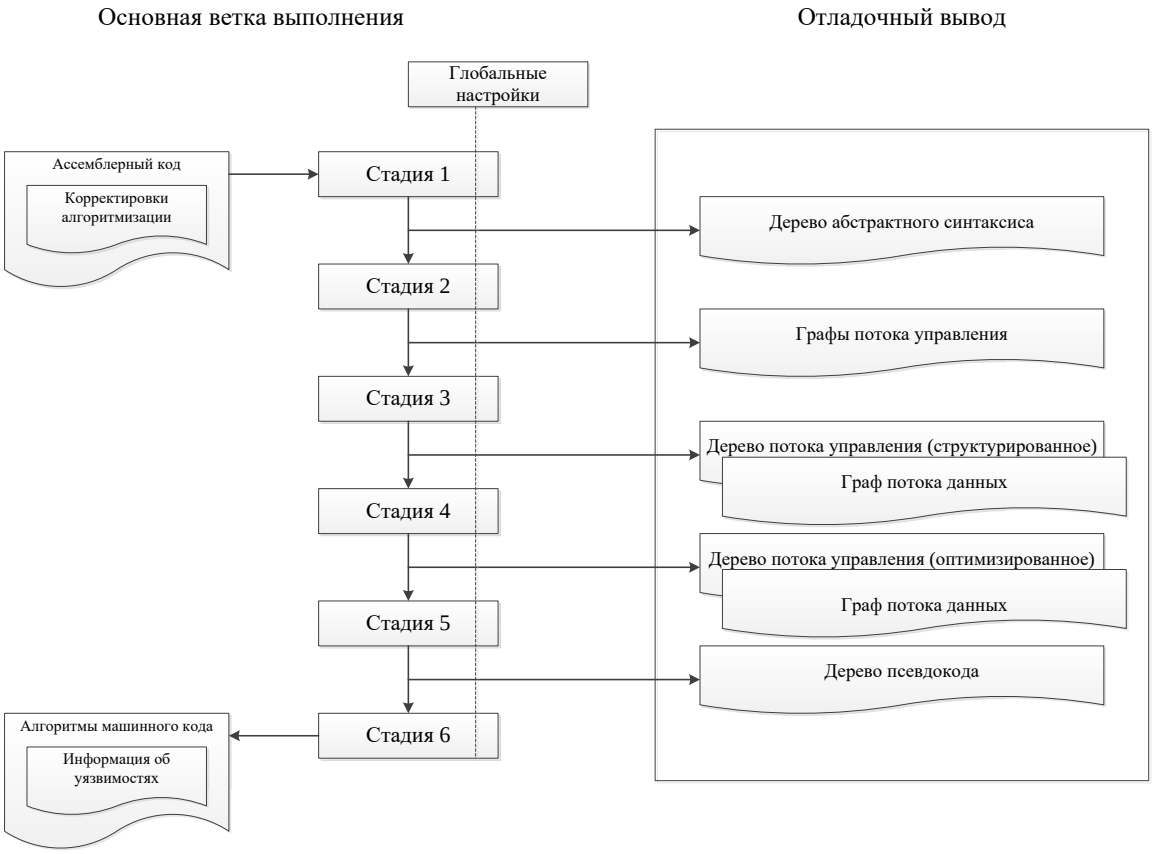

Рисунок 3.2 – Основная ветка выполнения Утилиты и ее отладочный вывод

Шаг 3 – Применить Утилиту. Для полученного АК запускается Утилита, в результате чего будет получено алгоритмизированное Представление.

Шаг 4 – Проанализировать результаты алгоритмизации. Анализируется полученное алгоритмизированное Представление на предмет его корректности и применимости для поиска уязвимостей.

Шаг 5 – Сделать корректировки к алгоритмизации. В случае необходимости на основании анализа алгоритмизированного Представления вносятся корректировки в АК (полученный на Шаге 2).

Шаг 6 – Применить Утилиту (вторая итерация). Аналогично Шагу 3, повторно применяется Утилита к АК с внесенными корректировками (полученными на Шаге 5).

Шаг 7 – Проанализировать результаты алгоритмизации (вторая итерация). Аналогично Шагу 4 анализируется алгоритмизированное Представление, полученное после внесения корректировок. Подтверждается, что результаты удовлетворительны и корректировки к алгоритмизации не требуются.

Шаг 8 – Завершить выполнение. Завершается использование продукта IDA Pro и Утилиты. Алгоритмизированное представление, полученное на Шаге 6, считается конечным.

Пошаговый алгоритм описанного базового сценария приведен в виде блок-схемы на рисунке 3.3.

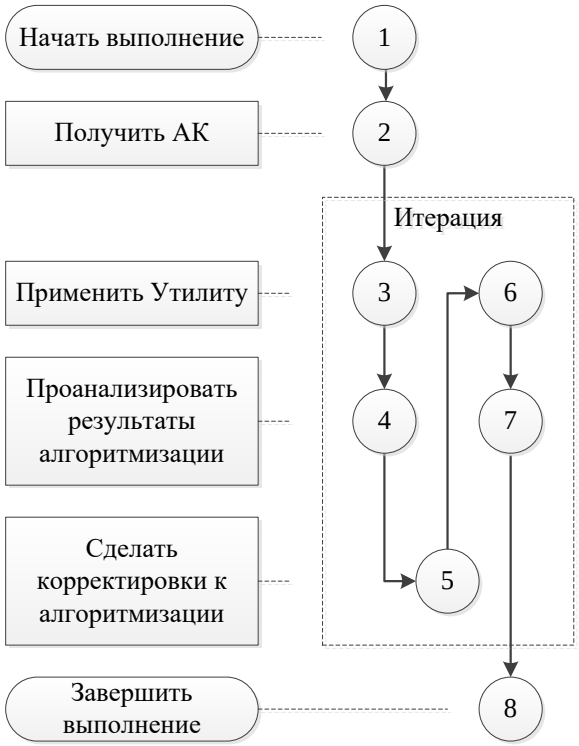

Рисунок 3.3 – Алгоритм базового сценария Утилиты

Примечание. Числа на рисунке 3.3 соответствуют номерам шагов сценария, а прямоугольник с пунктирными линиями очерчивает итерационную часть процесса алгоритмизации.

3.2 Разработка информационной архитектуры Утилиты

Организацию и формализацию данных, необходимых для работы и взаимодействия всех функциональных модулей, определяет так называемая информационная архитектура. Ее основными элементами будут эволюционирующие «рабочие» данные Утилиты в виде отдельных внутренних представлений – описывающих различные стороны входного АК, а их связями – зависимость получения одних представлений из других в процессе выполнения стадий. Информационная архитектура Утилиты в схематичном виде приведена на рисунке 3.4 [79].

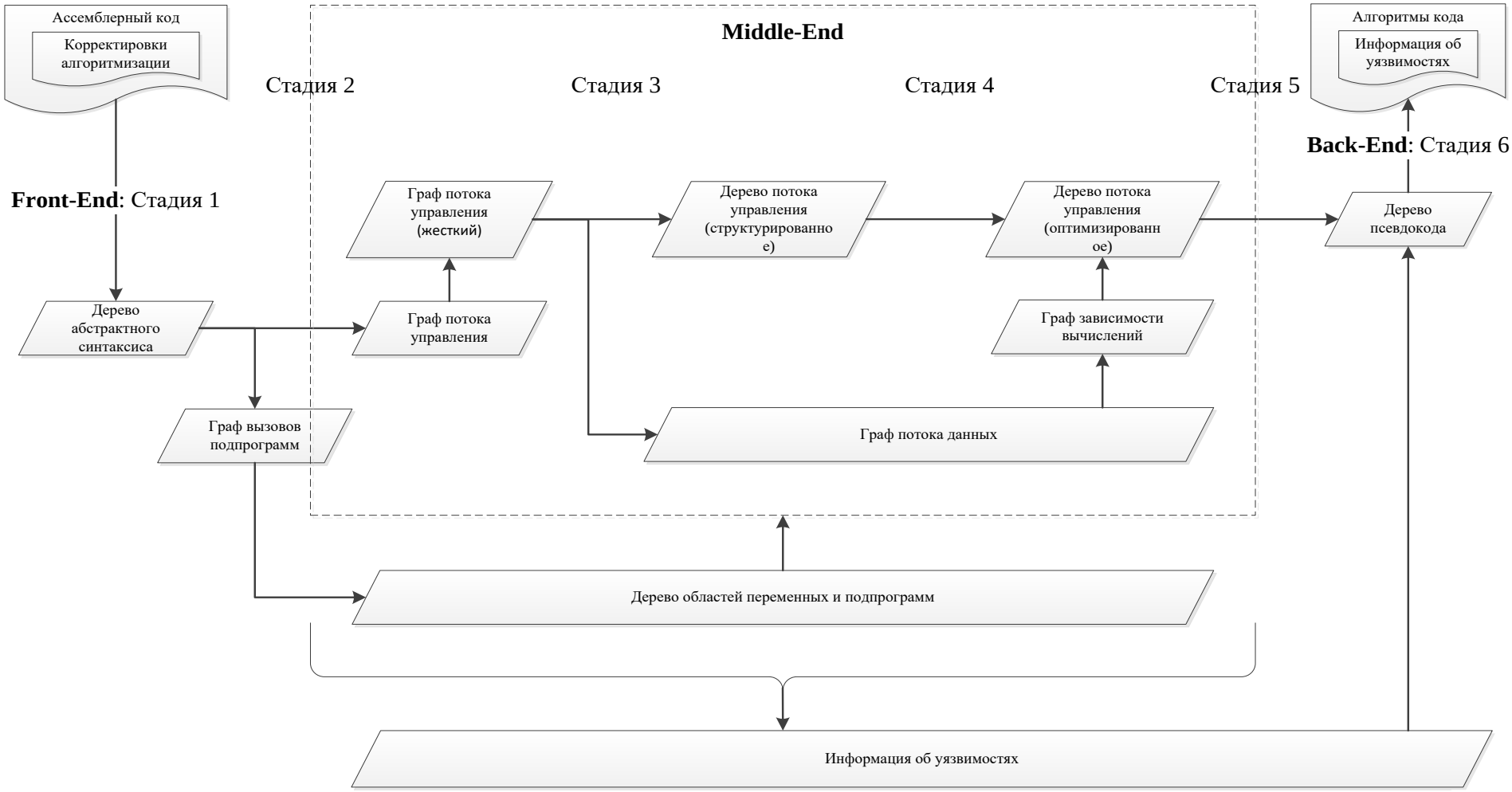

Рисунок 3.4 – Информационная архитектура Утилиты

Как хорошо видно по схеме, разделение внутренних представлений Утилиты по стадиям полностью соответствует аналогичному для функциональной архитектуры. Это абсолютно закономерно, поскольку каждая стадия функционирует согласно составляющим ее модулям, взаимодействующим посредством внутренних представлений.

Согласно предлагаемой информационной архитектуре существуют следующие преобразования между внутренними представлениями Утилиты. В фазе Front-End, определяемой единственной Стадией 1, производится разбор входного АК (и корректировок алгоритмизации) с последующим построением внутреннего представления дерева абстрактного синтаксиса, содержащего входной код в базово-формализованном виде. Затем выполняются стадии фазы Middle-End.

На Стадии 2 по дереву абстрактного синтаксиса собирается информация обо всех подпрограммах и их вызовах и строится граф взаимных вызовов подпрограмм, основным назначением которого является подготовка к построению SCT входного АК. Далее, во всей фазе Middle-End используется такое разделение, в котором все внутренние представления описывают лишь отдельную подпрограмму, что позволяет реализовать алгоритмы их обработки более обобщенно. Полный список же таких представлений для всех подпрограмм может быть получен путем обхода данного SCT. Также, на Стадии 2 каждое поддерево абстрактного синтаксиса, относящееся к отдельной подпрограмме, преобразуется к графу ее потока управления – сначала классическому, состоящему из базовых блоков, а затем и к «жесткому», имеющему дополнительные служебные узлы для унификации управляющих конструкций. На Стадии 3 производится структуризация жесткого графа управления – перевод в форму Насси-Шнейдермана – позволяющая описывать входной АК в виде дерева; безусловные переходы GOTO в нем считаются исключениями и задаются с помощью пары узлов «переход-метка». Также, на этой стадии строится граф потока данных, используемый в дальнейшем. На Стадии 4 производится оптимизация дерева потока управления с использованием графа зависимостей вычислений, полученного по графу потока данных. На Стадии 5, являющейся завершающей в фазе Middle-End, производится генерация

внутреннего представления псевдокода, описывающего входной АК; дерево такого псевдо-кода является окончательным и готовым для генерации по нему алгоритмов.

На единственной Стадии 6 фазы Back-End генерируется текстовое описание глобальных переменных и алгоритмов кода по внутреннему представлению в виде РСТ тривиальным способом; информация об уязвимостях в коде добавляется в выходное описание также на этой стадии.

Внутреннее представление информация об уязвимостях не имеет отдельного четко-выделенного контейнера (списка, дерева, графа), поскольку отметки об уязвимостях добавляются в граф или дерево управления в момент обнаружения посредством специальных узлов и хранятся в нем вплоть до генерации алгоритмов кода.

Опишем элементы информационной архитектуры (а именно – эволюционирующие данные в виде отдельных внутренних представлений) более детально. Для примеров представлений будем использовать простейшую функцию нахождения максимального их двух чисел, АК которой для процессора PowerPC (размером 24 байта) с комментариями на языке С имеет следующий вид:

```
                            // int max2(int X, int Y)
            max2:           // { /* X(R3), Y(R4), T(R5) */
0x00000000: cmpw r3, r4     //   if(X > Y)
0x00000004: ble label_1     //   {
0x00000008: mr r5, r3       //     T = X;
0x0000000C: b label_2       //   }
            label_1:        //   else {
0x00000010: mr r5, r4       //     T = Y;
            label_2:        //   }
0x00000014: mr r3, r5       //   return T;
0x00000018: blr             // }
```

АК описывает функцию, принимающую на входе 2 параметра (X, Y) в регистрах (R3, R4). После сравнения параметров, значение максимального из них присваивается локальной переменной T в регистре R5, которая впоследствии также присваивается регистру возвращаемого функцией значения – R3.

3.2.1 Входной ассемблерный код

Поскольку Утилита на вход должна принимать помимо АК еще и корректировки алгоритмизации, то целесообразно использовать специально-расширенный в этих интересах синтаксис языка ассемблер. При этом, исходя из того, что АК генерируется на предыдущем этапе Метода автоматически – с помощью скриптов в продукте IDA Pro – это реализуется достаточно простым способом.

Опишем расширение синтаксиса ассемблера на примерах без приведения формальной грамматики, поскольку расширение является интуитивно-понятным.

Во-первых, АК подпрограмм размещается в С-подобных блоках «{…}» с указанием имени подпрограммы «funct()». Так, для текущего примера код на расширенном синтаксисе ассемблера будет иметь следующий вид:

```
max2(){
    0x00000000:  cmpw r3, r4
    ...
    0x00000018:  blr
}
```

Примечание. Здесь и далее знак «…» означает пропущенный в интересах уплотнения записи код, а не специальный элемент синтаксиса.

Утилитой такой текст будет восприниматься, как АК подпрограммы с именем «max2».

Во-вторых, имеется возможность корректировки алгоритмизации путем явного задания частичного или полного списка параметров подпрограммы с помощью указания имени параметра и регистра процессора, через который он передается. Так, для используемого примера, Эксперт-М может указать, что подпрограмма «max2» принимает 2 параметра: X в регистре R3 и Y в регистре R4 следующим образом:

```
max2(X:r3, Y:r4){
    0x00000000:  cmpw r3, r4
    ...
    0x00000018:  blr
}
```

Здесь корректировкой считается именно С-подобное задание параметров подпрограммы: «X: r3, Y: r4». Корректировка будет использоваться Утилитой для пользовательского уточнения сигнатуры.

Также Эксперт-М может указать лишь часть из параметров, оставив возможность определения остальных Утилите – с помощью элемента синтаксиса «…» вместо отсутствующих параметров – следующим образом:

```
max2(X:r3, ...){
    0x00000000:  cmpw r3, r4
    ...
    0x00000018:  blr
}
```

Таким образом, Утилита в обязательном порядке отведет лишь одному параметру подпрограммы с именем X регистр R3; остальные же параметры и их регистры Утилита определит самостоятельно. Также возможно указание подпрограммы без параметров с помощью ключевого слова «none»:

```
max2(none){
    0x00000000:  cmpw r3, r4
    ...
    0x00000018:  blr
}
```

Синтаксически, такая корректировка является корректной, хотя и приведет к неверной алгоритмизации Утилитой текущего примера.

В-третьих, возможно явное указание регистров, используемых для возвращаемых значений подпрограммы, аналогичным с параметрами способом, но перед именем подпрограммы, следующим образом:

```
(r3) max2(X:r3, Y:r4){
    0x00000000:  cmpw r3, r4
    ...
    0x00000018:  blr
}
```

Такая запись сигнатуры подпрограммы указывает Утилите, что результат вычислений в подпрограмме «max2» возвращается через регистр R3. Аналогично указанию параметров подпрограммы возможно использование ключевого слова «none», означающего в данном контексте, что функция не возвращает никаких значений.

В-четвертых, возможно явное указание абсолютного адреса подпрограммы, используемого различными оптимизациями, с помощью символа «&» и самого адреса между параметрами подпрограммы и началом ее блока:

```
(r3) max2(X:r3, Y:r4) & 0x00001000 {
    0x00000000:  cmpw r3, r4
    ...
    0x00000018:  blr
}
```

Данная запись означает, что подпрограмма «max2» расположена по адресу 0x00001000. Необходимо отметить, что адреса, указанные перед инструкциями в блоке тела подпрограммы (для текущего примера с 0x00000000 до 0x00000018 с шагом 4), никак не используются Утилитой, носят лишь стилистический характер и генерируются автоматически скриптом для продукта IDA Pro.

В-пятых, возможно явное указание глобальных переменных с опциональным заданием их абсолютного адреса (что используется различными оптимизациями) следующим образом: в начале указывается ключевое слово VAR, затем идет имя глобальной переменной с опциональными символом «&» и ее абсолютным адресом, с завершающим символом «;». Пример задания глобальной переменной GLOB по адресу 0x00002000 является следующим:

```
// Declaration of Global Variables
var GLOB & 0x000020000;
```

В текущем примере подпрограмма «max2» не использует глобальных переменных, поэтому данная корректировка далее никак не упоминается.

Также, для полноценного представления возможностей работы Утилиты будем использовать изначальный пример без каких-либо корректировок, т. е. подпрограмму с сигнатурой «max2()».

3.2.2 Внутренние представления

После разбора текстового ассемблерного Представления вплоть до генерации текстового алгоритмизированного весь код в Утилите имеет формализованный виде согласно ее различным внутренним представлениям, приведенным на информационной архитектуре (см. рисунок 3.4). Описание внутренних представлений Утилиты и их «снимки» для примера приведены в Приложении В.

3.2.3 Выходной алгоритмизированный код

Утилита на выходе генерирует текстовое описание алгоритмов входного АК с использованием специального синтаксиса, подобного языку С, но имеющего ряд отличий, повышающих компактность и воспринимаемость кода. Выбор языка С в качестве базы для описания алгоритмов обусловлен его широким распространением, доказанной временем эффективностью разработки алгоритмов, заложенной структурированностью и большим количеством *удачных* синтаксических конструкций. Алгоритмизированный код для текущего примера при использовании Утилиты в режиме компактного стиля генерации алгоритмов следующий:

```
(r3) max2(w1_, w2_) {
    if (w1_ <= w2_)?true {
        w1_ = w2_;
    }
    return (w1_);
}
```

Очевидно, что восстановленный таким образом алгоритм довольно хорошо описывает логику работы исходного АК тестовой программы и доказывает следующую базовую работоспособность Утилиты. Во-первых, входные параметры подпрограммы «max2» были определены корректно – две переменные w1_ и w2_. Во-вторых, выходной параметр подпрограммы хранится в регистре R3, что является стандартным для кода, генерируемого компиляторами для процессора PowerPC. В-третьих, в коде не используется временная переменная T, присутствующая в исходном АК на регистре R5. В-четвертых, в восстановленном алгоритме соптимизирована одна из веток условного перехода, что делает его более компактным и воспринимаемым. И, в пятых, конечный код содержит расширение проверки условия с помощью постфикса «?true», очищая УС алгоритма от свойств проверки условия.

При использовании Утилиты в стандартном режиме (без установки компактного стиля генерации), алгоритм будет иметь следующий служебный вид:

```
(r3) max2(w1_:r3, w2_:r4) rp:lr {
    if (w1_ <= w2_)?true {
        w1_ = w2_;
    } else {
```

```
    }
    return rp:lr(w1_:r3);
}
```

Как хорошо видно, отличие стандартного режима генерации от компактного заключается в том, что для всех переменных указываются реальные процессорные регистры их размещения. Так, первый входной параметр подпрограммы W1_ передается через регистр R3, второй – W2_ через R4, адрес возврата подпрограммы RP хранится в регистре LR, а подпрограмма возвращает значение в W1_ через регистр R3.

Отличительные особенности предлагаемой нотации выходного представления Утилиты (помимо отсутствующих типов и других конструкций, несущественных для описания сути алгоритмов) описываются с помощью следующего расширения языка C для описания алгоритмов кода [77]. Также, поскольку типы переменных в восстановленном алгоритме не используются, то для декларации глобальных переменных используется ключевое слово «var», как показано в следующем примере:

```
// Declaration of Global Variables
var GLOBAL;
```

Слово «var» аналогично указанию типа переменной в C-языке.

«Умный» формат имен переменных по умолчанию

Имена восстановленных переменных, не заданные явно через корректирующие данные, составляются следующим образом. В качестве основы имени берется буква машинного типа («w» для машинного слова, «b» – для бита, «d» – для остальных случаев). К основе добавляется постфикс «_» в случае входных параметров, префикс «_» в случае выходных, префикс «@» для временных переменных. Таким образом, для текущего примера имена переменных W1_ и W2_ означают, что они определяются машинным словом и являются входными параметрами.

Процессорные регистры размещения переменных

Для всех локальных переменных возможно указание реальных регистров процессора, на котором они размещаются. Так, для текущего примера переменные W1_ и W2_ являются входными параметрами подпрограммы, передаваемыми через регистры R3 и R4 соответственно, притом первая также возвращает результат вычисления подпрограммы через регистр R3; а для хранения адреса возврата подпрограммы используется служебная переменная RP, заданная регистром LR.

Битовый доступ

В случае доступа к отдельному биту переменной используется синтаксис, подобный доступу к элементу структуры с использованием оператора «.», а именно – VAR.N_BIT. Например, запись

```
cr.2
```

означает доступ (для чтения или записи) к 2-му биту переменной CR, в котором, как правило, хранится результат проверки двух регистров на равенство предыдущей инструкцией (такой, как CMPW).

Универсализация циклов

Для стандартизации и расширения возможных способов описания циклов используется следующая их шаблонная запись:

```
PRE_CONDITION loop{
    ...
} POST_CONDITION;
```

которая задает, помимо самого тела цикла (в блоке «{ … }»), условие входа в него перед первой операцией цикла и условие выхода после последней. Такая запись позволяет одинаковым образом записывать, как классические конструкции циклов WHILE-DO (с пре-условием):

```
(cond == true) loop {
    ...
};
```

и DO_WHILE (с пост-условием):

```
loop {
    ...
} (cond == true);
```

но также более сложные много-условные (с пре-пост-условием):

```
(cond1 == true) loop {
    ...
} (cond2 == true);
```

и бесконечные циклы (без условий):

```
loop {
    ...
}
```

Данное расширение языка связано с тем, что подобная ситуация с условиями выхода из цикла имеет достаточно частое отражение в реальных примерах АК, и целесообразно их вид оставлять и в восстановленном коде, а не притягивать к стандартному для языка С.

Многоуровневый выход из циклов

Одной из частых ситуаций в АК является условный переход из многократно-вложенного цикла за пределы всего каскада циклов, что не описывается адекватным способом на языке С. Для разрешения такой ситуации синтаксис дополнен расширением оператора BREAK с указанием количества уровней, на которые необходимо подняться – в языке С поднятие происходит только на один уровень. Пример использования синтаксиса может быть следующим:

```
loop {
    x += 1;
    loop {
        y += 1;
        if (x * y == 100) {
            break (2);    // Выход из двух уровней цикла – на метку end
        }
    }
}
end:;
```

Согласно примеру, каждый из циклов инкрементирует собственную переменную (X и Y); притом, когда во вложенном цикле произведение переменных станет равно числу 100, то произойдет выход из обоих циклов одновременно, т. е. переход на метку END.

Возвращение нескольких параметров из подпрограммы

Поскольку в некоторых случаях подпрограммы возвращают многобайтные значения через структуры, которые крайне редко (и не так необходимо) определяемы по машинному коду, то целесообразно описывать сигнатуру таких подпрограмм, как имеющие возвращаемое значение в виде последовательности регистров; такая информация позволит более корректно восстанавливать другие подпрограммы, использующие результаты работы данной. В расширенном синтаксисе список возвращаемых регистров задается посредством «(…)» на месте классического С-подобного типа возвращаемого значения функции. Для текущего примера список регистров может быть описан, как:

```
(r3) max2...,
```

что соответствует одному возвращаемому значению в регистре R3.

Абсолютные адреса размещения переменных

В АК, полученном из работающего образа устройства, все глобальные переменные и подпрограммы (также, как и их инструкции), уже имеют абсолютные адреса – в отличие от кода, генерируемого по исходному в процессе компиляции, поскольку для него назначение адресов происходит на этапе *линковки*. Адреса могут быть использованы как Утилитой для подстановки соответствующих им переменных в выражения, так и при ручном анализе восстановленного кода Экспертом-М – для понимания связи между алгоритмами и обрабатываемыми областями данных или корректировок процесса алгоритмизации. Для указания адреса используется символ «&» с последующим абсолютным адресом. Например, следующее указание корректировок в АК:

```
var GLOBAL & 0x00001234;
max2() & 0x00005678 {
    ...
```

означающее, что глобальная переменная GLOBAL расположена по адресу 0x00001234, а подпрограмма «max2» – по адресу 0x00005678, приведет к следующему виду восстановленных алгоритмов:

```
var GLOBAL & 0x1234;
(r3) max2(w1_:r3, w2_:r4) rp:lr & 0x5678 { ...
```

Поскольку практически вся собранная информация хранится в формализованном виде, а генерация текстового описания алгоритмов производится по древовидному представлению, то отображение новых существенных особенностей и сторон алгоритма реализуемо достаточно просто – путем внесения необходимой информации в РСТ (созданием новых или обновлением свойств старых узлов) с незначительной корректировкой генерации по дереву текстового описания.

3.3 Разработка программной архитектуры Утилиты

Опишем детали реализации прототипа Утилиты (далее – Прототип) с точки зрения основных алгоритмов работы ее модулей, строения ИК, использованных средств его разработки и отладки – то есть программную архитектуру [81].

3.3.1 Основные алгоритмы модулей

Функциональная архитектура представляет Утилиту в виде шести последовательно выполняемых стадий, сгруппированных по трем фазам (рисунок 3.1). Каждая стадия содержит несколько модулей (включая анализаторы первой стадии); взаимодействие последних с модулями противоположных стадий осуществляется посредством данных, представленных информационной архитектурой и поддерживающих S-модель (рисунок 3.4). Также, есть модули, сквозные для стадий – обмен данными с ними происходит на протяжении всей работы. Основные 23 алгоритма, соответствующие модулям функциональной архитектуры, приведены далее. Принцип работы и блок-схемы алгоритмов приведены в Приложении Г.

3.3.2 Структура исходного кода

Основная часть ИК Прототипа написана на языке C++ с четким делением на заголовочные и компилируемые файлы для каждого основного класса объектно-ориентированного программирования (далее – ООП) или их группы. Также, активно используются конструкции *namespace* для логического объединения клас-

сов. Большинство ИК разработано вручную, хотя отдельные его части являются модификацией открытых реализаций требуемого функционала (например, разборщик файлов формата JSON [161]).

Весь ИК делится на четыре следующих проекта. Проект статической библиотеки Common является общим для большой группы приложение и содержит вспомогательный функционал, такой, как функции конвертации различных типов данных, поддержку битовых последовательностей, расширенную поддержку работы с файлами и потоками, функции для работы с текстом, программно-языковые определения различных часто используемых типов, констант, макросов. Проект статической библиотеки Json реализует функционал по разбору и генерации данных в формате JSON и используется в Прототип для отладочных целей. Проект статической библиотеки Core является главным, поскольку содержит весь основной функционал Прототипа. Использование библиотечного типа позволяет запускать процесс алгоритмизации не только из командной строки ОС, но и любыми другим способами (например, из графического приложения на C#). Проект консольного приложения Console является оболочкой для запуска библиотеки Core, производящей разбор аргументов командной строки и передающий их вместе с входным ассемблером в основную функцию библиотеки Core.

3.3.3 Средства сборки и отладки исполняемого кода

Прототип разработан в графической среде разработки Microsoft Visual Studio 2010 (далее – MSVS2010) с подключением дополнительных средств построения кода; в ней же производилась отладка кода. Среда разработки MSVS2010 была выбрана по причине широкого распространения, возможности бесплатного использования, наличия гибких настроек, удобной визуальной среды, полноценной поддержки современных языков программирования и возможности подключения дополнительного инструментария.

В качестве генераторов кода использовались стандартные компиляторы, ассемблеры и линкеры, входящие в состав MSVS2010. Также, для реализации Ста-

дии 1 Прототипа производилась практически полная генерация кода ее модулей – лексического и синтаксического анализаторов, с помощью соответствующих подключенных утилит: генератора лексического анализатора Flex и генератора синтаксического анализатора Bison, работающих совместно. Исходные лексика и грамматика входного ассемблера, используемые генераторами, создавались вручную на специальных языках; они соответствуют синтаксису ассемблера PowerPC, генерируемого продуктом IDA Pro.

Для отладки кода использовались встроенные в MSVS2010 средства. Помимо этого было разработанно ПС DotViewerRT, представляющее собой графическое приложение, отображающее в режиме РМВ граф, заданный на языке DOT [162] во внешнем файле, поддерживаемый пакетом утилит GraphViz [163]. Такие графы создавались в режиме отладки для большинства внутренних представлений Прототипа; при этом, узлы графа имели специальную окраску согласно выполняемым действиям (например, модули оптимизации раскрашивали затрагиваемые узлы в зеленый и красный цвета). Работа же в реальном времени DotViewerRT позволяла в процессе пошаговой отладки ИК в MSVS2010 видеть динамику изменения графа. Описанная схема отладки представлена на рисунке 3.5.

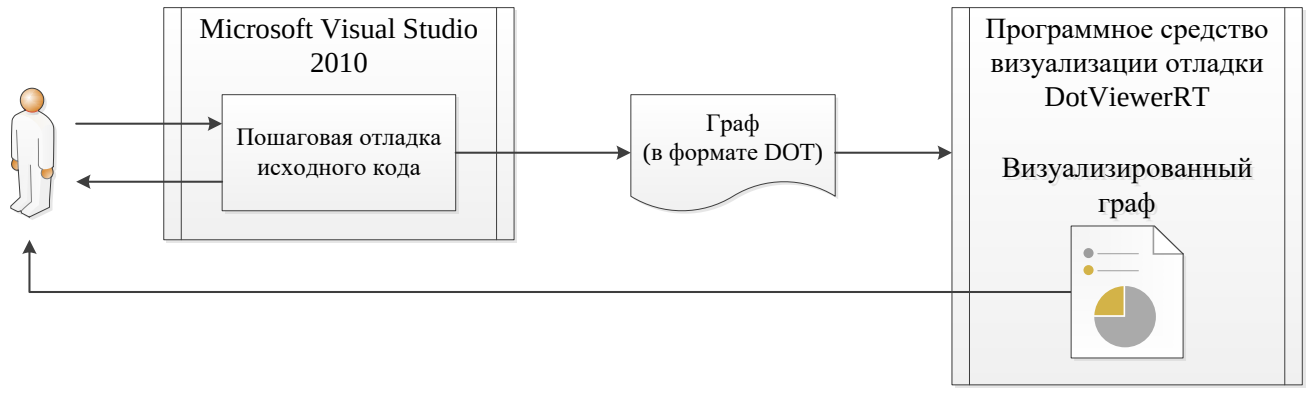

Рисунок 3.5 – Схема отладки исходного кода Прототипа
с помощью программного средства визуализации DotViewerRT

Необходимо отметить, что подобная схема динамической отладки сложных структур данных (т. е. пошаговая визуализация графа с помощью внешнего ПС) аналогов практически не имеет.

3.4 Примеры тестирования работоспособности и текущее состояние прототипа

Приведем примеры практического применения Прототипа, демонстрирующие его «базовую» работоспособность. Также обозначим текущее состояние разработки Утилиты и перспективы ее развития.

Реализованный прототип имеет следующий формат запуска с опциями, управляющими настройкой работы:

```
> utility.exe <test.asm [-s] [-c] [-d],
```

где utility.exe является именем исполняемого файла Прототипа, файл test.asm содержит АК для восстановления алгоритмов, символ «<» перед именем файла указывает ОС необходимость передачи содержимого файла во входной поток консольного приложения: Имеются следующие необязательные опция: «-s» – для генерации восстановленных алгоритмов в компактном стиле (без указания регистров процессора и т. п.), «-c» – для добавления комментариев в код восстановленных алгоритмов, «-d» – для отладочного вывода в консоль.

Работоспособность Прототипа на примере стандартной и сложной функций, а также применимость для поиска уязвимостей обосновывается далее. ИК примера, его АК для процессора PowerPC, блок-схема алгоритма, командная строка запуска Прототипа и результаты работы (включая их предварительный анализ), а также вносимые Экспертом-М корректировки приведены в Приложении Д.

3.4.1 Пример 1 (простой). Функция нахождения максимального из трех чисел

Рассмотрим пример, демонстрирующий работоспособность Прототипа на стандартной функции нахождения максимального из трех чисел. Подпрограмма, как правило, имеет однотипный вид, не зависящий от языка программирования и процессора выполнения.

Согласно результатам работы Прототипа по восстановлению АК примера (см. Приложение Д) можно сделать следующие выводы. Во-первых, алгоритм, полученный Прототипом, соответствует исходному и хорошо воспринимаем че-

ловеком. Во-вторых, на основании первой итерации работы Прототипа Эксперт-М может внести корректировки и получить еще более оптимальное (с позиции восприятия) описание алгоритма. И, в-третьих, Прототип произвел оптимизацию самой логики алгоритма, упростив ее примерно на 60% (согласно соотношению строк входного исходного и выходного алгоритмического кода).

Также можно отметить наличие в восстановленных алгоритмах элементов предложенного и описанного расширения языка С в виде конструкций условных переходов вида «if (CONDITION)?true», которые повышают быстроту первоначальной воспринимаемости алгоритмов.

3.4.2 Пример 2 (сложный). Функция алгоритма определения старшего бита числа

Продемонстрируем работоспособность Прототипа на нетривиальной функции нахождения старшего ненулевого бита числа «быстрым» алгоритмом.

Согласно результатам работы Прототипа по восстановлению АК примера (см. Приложение Д) можно сделать следующие выводы. Во-первых, алгоритм, полученный Прототипом на первой итерации, подобен исходному, хорошо воспринимаем человеком, хотя и имеет отличие в количестве возвращаемых параметров. И, во-вторых, на основании анализа результатов первой итерации, Эксперт-М может явно внести корректировки для получения более точного вида алгоритмов – указав верные (с его точки зрения) возвращаемые параметры.

Также можно отметить наличие в восстановленных алгоритмах элементов предложенного и описанного расширения языка С в виде конструкций цикла с пост-условием «loop{ … } (COND)», которое повышает быстроту первоначальной воспринимаемости алгоритма.

3.4.3 Пример 3 (специализированный). Функция с внедренной закладкой

Продемонстрируем использование Прототипа для поиска уязвимостей в МК для функции проверки корректности пароля. Функциональность такого поиска реализована лишь в качестве доказательства применимости Прототипа, и поэтому

пример можно считать условно-существующим, хотя он потенциально возможен в ПО реальных ТКУ.

Согласно результатам работы Прототипа по восстановлению примера АК с внедренной закладкой (см. Приложение Д) можно сделать следующие выводы. Во-первых, Прототип корректно восстановил работоспособную (т. е. исполняемую в реальности) часть алгоритма, не смотря на его неполноценность по причине «разрушения». Во-вторых, что в контексте будущего поиска уязвимостей особенно важно, Прототип обнаружил факт «разрушения» алгоритма путем внедрения чужеродного кода и отобразил предупреждающий комментарий об этом в точке, максимально приближенной к месту «разрушения». И, в-третьих, граф потока управления из отладочного вывода Прототипа отражает факт внедрения закладки по «разрушенной» структуре алгоритма, что может способствовать росту эффективности поиска и анализа уязвимостей.

3.4.4 Реализованный функционал и перспектива доработки Утилиты

Реализованный в текущей версии Утилиты функционал состоит из следующих ее основных возможностей [74]:

– поддержка инструкций АК (в текущем варианте – для PowerPC) на 90%;

– поддержка специфичных конструкций, генерируемых IDA Pro на 60%;

– поддержка основных корректировочных данных;

– выделение базовых СМД, таких, как циклы и ветвления;

– оптимизация, включающая удаления лишних меток, переходов, пересылок данных, упрощение выражений;

– лаконизация, включающая использование инкрементации/декрементации, доступа к элементам массива и отдельным битам, подстановку идентификаторов глобальных переменных вместо их адресов, применение конструкций расширенного синтаксиса языка С;

– поиск уязвимостей и сигнализация о них (в текущем варианте поддерживается следующее детектирование уязвимостей: разрушение структуры алгоритма

после внедрения закладки и попытка записи в память, указанную как «только для чтения»; попытка изменения переменных, указанных экспертом как доступных только для чтения);

– генерация деклараций глобальных переменных и подпрограмм;

– генерация специальной низкоуровневой информации, включающей реальные регистры размещения переменных (в частности входных и выходных параметров) и адресов возврата из подпрограмм;

– отладочный вывод для внутренних представлений на всех стадиях Утилиты, в том числе и для отладки динамических изменений сложных структур (графов);

– генерация блок-схем восстановленных алгоритмов (с информацией о потенциальных уязвимостях);

– генерация архитектуры ПрК в виде его деления на модули и разделяемые переменные с указанием связей по данным и управлению.

В интересах приведения Утилиты в состояние полноценного использования на практике, даже при условии недоработок в оптимизационных и лаконизационных модулях, необходима реализация в Утилите следующих возможностей.

Во-первых, доведение поддержки инструкций входного ассемблера (в текущем варианте – основного набора для PowerPC) и специфичных конструкций, генерируемых IDA Pro, таких например, как структуры (распознаваемые вручную) и стек функции до 100%. Во-вторых, поддержка вспомогательных корректировочных данных для повышения точности распознавания СМД (например, указание циклов и их условий). В-третьих, выделение сложных СМД, таких, как итерационные переменные циклов, конструкции SWITCH. В-четвертых, выделение высокоуровневых СМД, таких, как модули и общая архитектура.

Утилиту целесообразно развивать по следующим направлениям, которые для ее практического использования не являются обязательными, но значительно улучшают «потребительские свойства» – это: развитие механизма отладки кода; расширение принципиальных возможностей корректировок алгоритмизации; детектирование основных СУ и ВУ.

Выводы по разделу 3

В качестве объекта исследовался метод алгоритмизации машинного кода на предмет автоматизации процесса его выполнения с помощью специального программного средства алгоритмизации.

В ходе исследования:

1) Разработана функциональная архитектура программного средства алгоритмизации, состоящая из следующих элементов: последовательно-выполняемые фазы с их делением на стадии, модули и их взаимодействия, а так же входные, выходные и определяющие S-модель внутренние данные.

2) Для каждой стадии описаны модули архитектуры со следующим назначением: лексический, синтаксический и семантический анализатор на стадии 1; выделение подпрограмм, построение графа потока управления и выделение глобальных переменных на стадии 2; построение графа потока данных, уточнение сигнатур подпрограмм и выделение структурных метаданных на стадии 3; оптимизация структурных метаданных, дерева потока управления и вычислений на стадии 4; генерация псевдокода глобальных переменных и подпрограмм, лаконизация псевдокода на стадии 5; генерация текстового описания глобальных переменных, подпрограмм и информации об уязвимостях на стадии 6.

3) Задан базовый сценарий функционирования программного средства алгоритмизации и ее глобальные настройки в виде флагов, влияющих на отладочный вывод, комментирование алгоритмов, параметры лаконизации алгоритмизированного представления и компактности его вывода, генерацию вспомогательной низкоуровневой информации.

4) Разработана информационная архитектура программного средства алгоритмизации, задающая организацию и формализацию ее внутренних данных. Описаны синтаксисы входного ассемблерного кода и выходного текстового описания алгоритмов, а также формализованные внутренние данные, используемые Утилитой.

5) Разработана программная архитектура программного средства алгоритмизации, описывающая алгоритмы ее основных модулей. Описана технология программной реализации его прототипа в виде строения исходного кода, а также использованных средств разработки и отладки.

6) Произведено практическое применение прототипа программного средства алгоритмизации на базовых примерах ассемблерного кода функций: нахождения максимального их трех чисел, алгоритма определения старшего бита числа и проверки пароля с внедренной закладкой. Обозначено состояние разработки Утилиты и перспективы ее развития.

Основным научным результатом, изложенным в третьем разделе, является архитектура программного средства алгоритмизации машинного кода, суть которой заключается в пофазном и помодульном описании средства с позиции взаимоувязанных функциональной, информационной и программной составляющих.

Частным научным результатом, изложенными в третьем разделе, является базовое тестирование разработанного прототипа программного средства алгоритмизации. Этот результат непосредственно был использован для подтверждения успешности выбранного способа автоматизации метода.

Основное содержание раздела и полученных научных результатов изложено в работах автора [62, 69, 72, 74, 77-79, 81].

4 ОЦЕНКА ЭФФЕКТИВНОСТИ АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА

4.1 Целевой способ поиска уязвимостей в машинном коде с применением алгоритмизации

Целевой способ поиска уязвимостей в МК может быть схематично описан следующим образом. Изначально, к исследуемому МК применяется предложенный Метод; используя специально ПС – Утилиту – он создает алгоритмизированное Представление (автоматизировано с возможностью ручной корректировки). А затем, по этому Представлению (хранящему архитектуру и алгоритмы) производится ручной поиск уязвимостей.

Произведем сравнительную оценку предложенного способа поиска с применением алгоритмизации МК с известными, используя введенные в пункте 1.1.2 критерии эффективности. Предназначенность алгоритмизированного Представления для последующего поиска СУ и ВУ соответствует выполнению Критериев 1 и 2. Применение программных средств для выполнения «рутинной работы» по алгоритмизации МК (т. е. не требующей высокого уровня субъективности) – продукта IDA Pro на Этапе 1 и Утилиты на Этапе 2 – соответствует выполнению Критерия 3. Применение на Этапе 3 труда Эксперта-М для анализа полученного Представления, как и последующий ручной поиск уязвимостей, соответствует выполнению Критерия 4. Используемая обобщенная форма алгоритмизированного Представления, имеющая подобный языку С синтаксис и адаптированный для представления алгоритмов МК, соответствует выполнению Критерия 5. Метод алгоритмизации позволяет обрабатывать большие объемы данных, поскольку в принципы его работы заложена последовательная обработка всех инструкций машинного кода с их структуризацией, восстановлением архитектуры, что соответствует выполнению Критерия 6. Этап 4, обеспечивающий итерационную работу алгоритмизации с корректировкой входных данных под требования Эксперта-М, соответствует выполнению Критерия 7. Использование специального синтак-

сиса алгоритмизированного Представления, повышающего компактность и воспринимаемость кода, а так же его гармонизация Инженером-М на Этапе 5, соответствует выполнению Критерия 8. Дополнительная информация об потенциальных уязвимостях, предоставляемая в результате алгоритмизации, соответствует выполнению Критерия 9. Как простота проведения алгоритмизации Инженером-М, так и отсутствие специальных требований (например, знание инструкций процессора выполнения) к Эксперту-М, анализирующему полученное представление, означают выполнение Критерия 10.

Таким образом, соответствие способа поиска уязвимостей с применением алгоритмизации всем критериям присваивает ему оценку, равную 10-ти балам. Для наглядности результаты такой оценки представлены в виде гистограммы на рисунке 4.1.

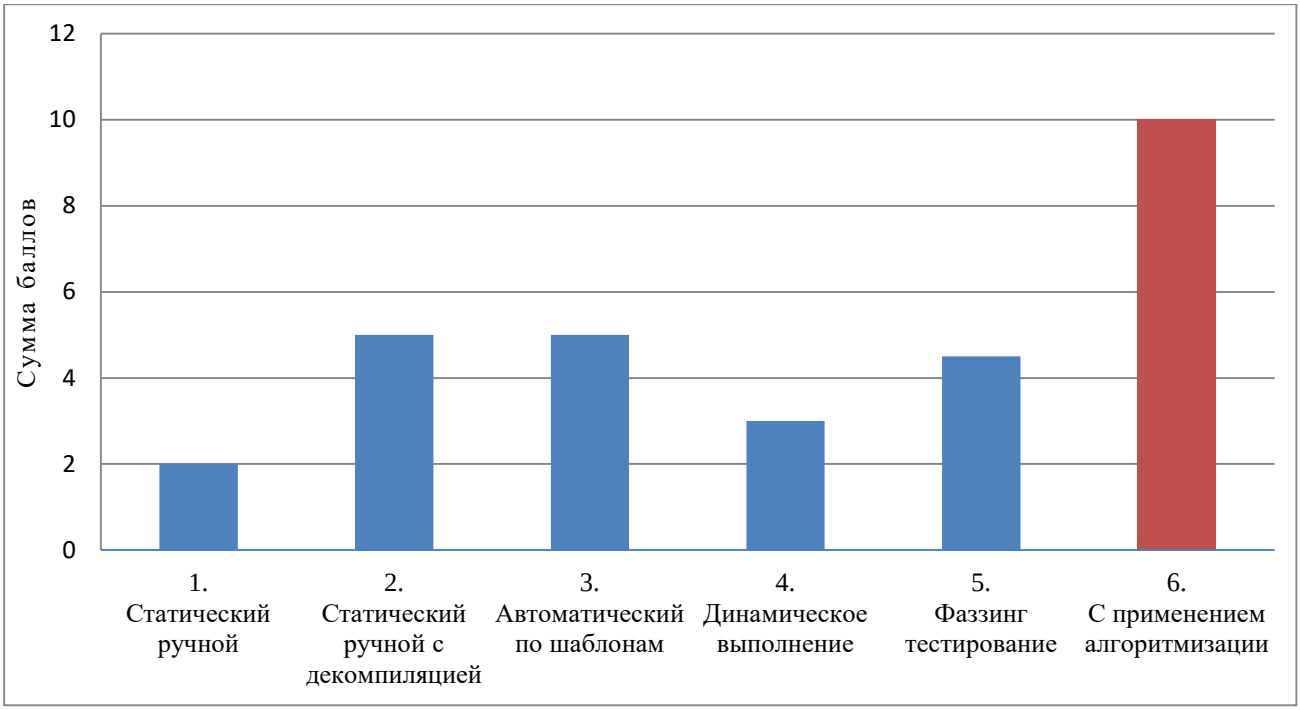

Рисунок 4.1 – Гистограмма критериальной оценки способа поиска уязвимостей с применением алгоритмизации и альтернативных

Принимая (в виде допустимого упрощения) значимость критериев одинаковой можно утверждать, что предложенный способ поиска уязвимостей превосходит ближайший аналог в 2 раза. Таким образом, существенное повышение эффективности поиска СУ и ВУ можно считать доказанным. Тем не менее, приведенная критериальная оценка является достаточно грубой и в ряде случаев не удовлетво-

рительной. Поскольку основополагающим элементом целевого способа поиска является Метод, то необходимо произвести более детально оценку эффективности именно его. Метод является достаточно сложной и многогранной сущностью и поэтому целесообразно произвести его разностороннюю оценку с использованием различных объектов целевого способа, а именно: средства алгоритмизации, получаемого представления и последующего поиска уязвимостей.

Для разноаспектной оценки Метода предлагается соответствующий комплекс научно-методических средств, состоящий из следующих: методики оценки потребительских свойств (далее – Методика 1) и функциональных возможностей (далее – Методика 2) средства алгоритмизации, критерии оценки алгоритмизированности и метрика понятности получаемого представления, методика оценки эффективности поиска уязвимостей по алгоритмизированному представлению (далее – Методика 3). Схема такой оценки представлена на рисунке 4.2.

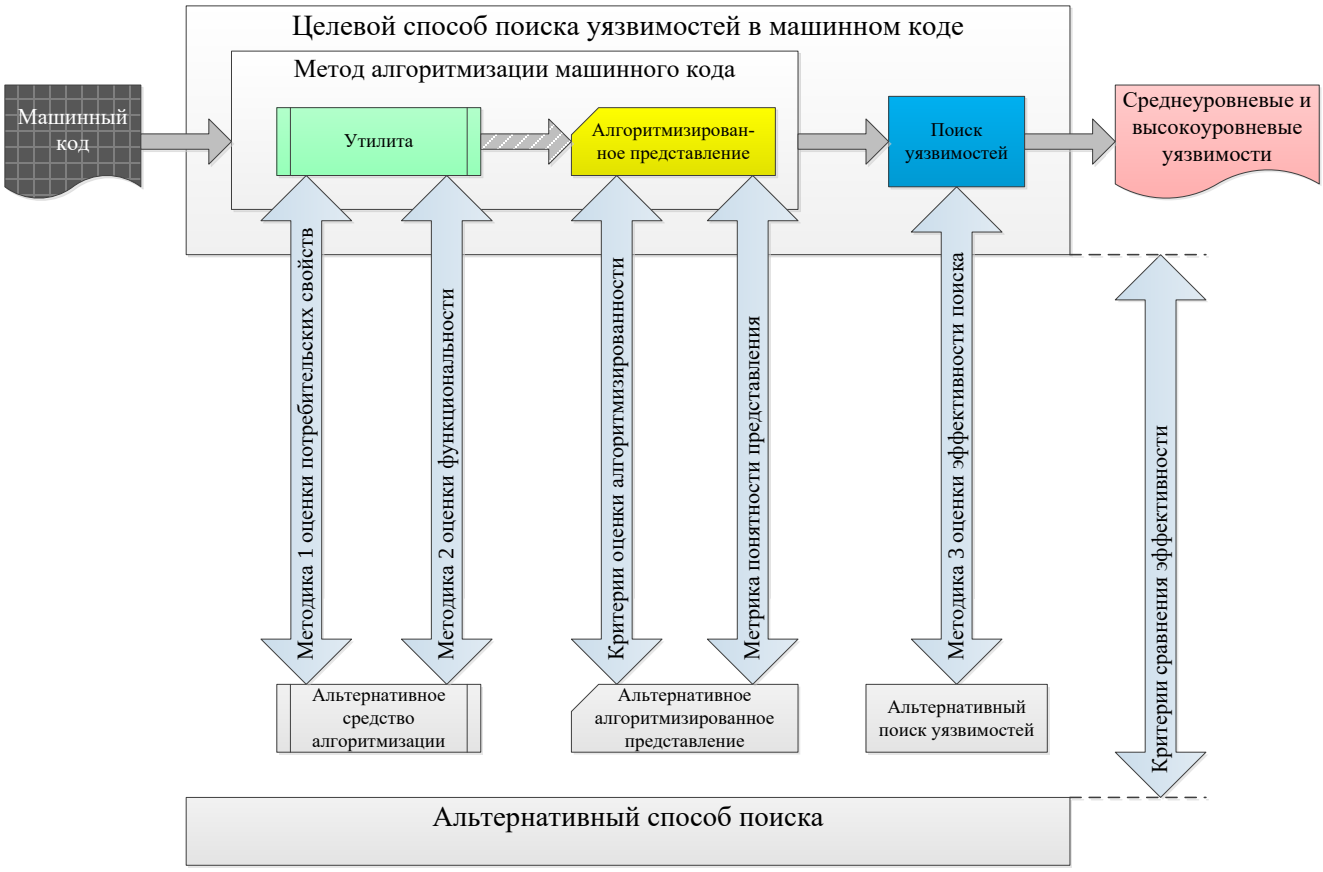

Рисунок 4.2 – Схема оценки Метода на базе целевого способа поиска уязвимостей в МК

Заявленные методики, критерии и метрика, а также полученные результаты, приводятся в подразделах 4.2–4.6.

4.2 Разработка и применение методики оценки потребительских свойств средства алгоритмизации

Основополагающее ПС Метода – Утилита – предназначено для автоматизации процесса алгоритмизации, а, следовательно, предполагает работу с ним человека. Для обоснования сделанного выбора в пользу средства необходимо оценить его потребительские свойства по сравнению с альтернативными. Для этого предлагается соответствующая Методика 1 и приводятся результаты ее применения [65].

4.2.1 Выбор и обоснование способа оценки

Потребительские свойства средства являются, как правило, многокритериальной системой оценок, носящий в основном субъективный характер. При этом сами критерии носят неколичественный характер и зачастую не могут быть вычислены аналитически. Само средство (любого рода: программное, аппаратное, ручное, автоматическое и т. п.) предназначено для применения высококвалифицированными экспертами, обладающими достаточными знаниями для его использования, способными делать обоснованные заключения (оценки).

В рамках методики оценки обозначим совокупность потребительских свойств средства, как некой меры удобства его использования для решения задачи алгоритмизации, определяемой по совокупности разно-важных критериев. Одно средство будет иметь большее удобство от использования, чем другое, в случае, если получение с его помощью алгоритмов МК по совокупности критериев более предпочтительно, учитывая целевую функцию – поиск уязвимостей. Некоторые показатели критериев имеют абсолютные значения, некоторые же оцениваются субъективно. При этом часть из них крайне трудно ранжируема и подходит лишь для парного сравнения. Ко всему, многие показатели являются взаимно противоречивыми – улучшение одного из них приводит к ухудшению другого. Например, часто увеличение скорости работы со средством уменьшает достоверность полу-

чаемых результатов (из-за снижения концентрации Эксперта-М – человеческий фактор); зависимость трудоемкости при этом носит нелинейный характер.

С учетом выбранного класса методов экспертной оценки и на основании введенного понятия удобства использования удовлетворительным решением для оценки средств (из известных) может быть Метод анализа иерархий (далее – МАИ) [164]. Порядок применения метода следующих шагов: построение иерархии модели проблемы (далее – Иерархия), определение приоритетов элементов, синтез глобальных приоритетов альтернатив и принятие решения. Классическая модель Иерархии состоит из трех уровней. Верхний уровень является главной целью выбора, нижний – множеством вариантов (альтернатив) достижения цели, а средний – критериями, влияющими на достижение цели альтернативами. При этом средний уровень может быть разбит на отдельные подуровни критериев.

Применим МАИ к текущей задаче. В нашем случае главной целью считается выбор средства алгоритмизации МК, имеющего наивысшее удобство использования для последующего поиска уязвимостей. Альтернативы представляют собой существующие средства, среди которых необходимо сделать выбор; критерии же определяют те характеристики средств, которые напрямую влияют на потребительские свойства. Одноуровневость множества критериев является достаточной. Назначив приоритеты всем элементам Иерархии и проведя согласно МАИ математические операции свертки, получим глобальные приоритеты альтернатив; выбор максимального значения даст искомое средство, обладающее максимальным удобством использования. Описанная задача выбора между средствами алгоритмизации обладает неоспоримой новизной, так как не имеет готовых решений. Это приводит к необходимости разработки соответствующей методики оценки на базе МАИ – Методики 1 – практически «с нуля», а не к поиску существующих.

4.2.2 Этапы методики оценки

Для создания полноценной методики оценки на основе МАИ зададим шаги ее выполнения: выбор альтернатив, формирование критериев и вычисление их приоритетов. Получение локальных приоритетов альтернатив будет происходить

в процессе применения Методики 1, а вычисление глобальных – на заключительном этапе с помощью линейной свертки (согласно МАИ).

Этап 1. Выбор альтернатив для оценки

На первом этапе необходимо проанализировать существующие на данный момент средства алгоритмизации и отобрать те, из которых необходимо выбрать наиболее удобное для использования. В общем случае, альтернативы должны состоять из всех известных на текущий момент средств (программных, аппаратных, ручных, автоматических, иных). В качестве разумного упрощения можно в качестве альтернатив использовать классы средств. В случае если средство алгоритмизации может быть отнесено сразу к нескольким классам, будем считать, что каждому классу соответствует лишь часть его функционала. Для этого произведем деление всех известных средств алгоритмизации на классы.

Класс 1. Ручное средство алгоритмизации можно считать наипростейшим с точки зрения технологий и наисложнейшим по трудозатратам, поскольку оно применяется без каких-либо программных инструментов, а лишь за счет физического и умственного труда специалиста по оценке. Суть способа сводится к анализу АК, осознанию принципов его работы и ручному воссозданию алгоритмов. Ярким примером является ручной анализ МК «низкоуровневыми» программистами и «хакерами» с последующим документированием алгоритмов.

Класс 2. Визуализатор блок-схем является развитием ручного способа, добавляющее в него инструмент автоматического создания графического изображения алгоритмов. Примером могут служить достаточно редкие дизассемблеры МК, включающие в себя графический визуализатор (иногда и редактор) получаемого кода в виде блок-схем, такие как AsmEditor [165], Visustin [166]. Также к данному классу можно отнести как классические среды программирования на различных диалектах ассемблера со встроенным визуализатором, так и достаточно простые утилиты генерации графов по АК.

Класс 3. Следующим развитием визуализаторов блок-схем можно считать добавление возможности управления процессом их создания, то есть наличие интерактивности. Это может быть достигнуто, например, с помощью изменения АК

с параллельным перестроением графов алгоритмов или настройкой критериев выбора элемента блок-схемы в случае альтернативных вариантов. Наиболее известным представителем данного класса можно считать интерактивный дизассемблер – продукт IDA Pro. Помимо основной своей задачи, он синхронно строит блок-схемы ассемблерных подпрограмм, позволяет изменять МК, добавлять комментарии и программные типы, видоизменять блок-схемы путем сокрытия ее несущественных элементов и др.

Класс 4. Отдельным направлением – реверс-инжинирингом – в инструментальных средствах можно считать утилиты декомпиляции. Их основным назначением служит попытка получения ИК по имеющемуся МК или хотя бы создание его аналога. В качестве примера работающих декомпиляторов можно указать следующие: Boomerang, C4Decompiler, DDC, Rec Studio 4 и SmartDec. Отдельно выделим упомянутый ранее продукт IDA Pro и его плагин декомпиляции Hex-Rays (далее, IDA+Hex-Rays) как раз и предназначен для получения ИК по МК, дизассемблированному в нем же.

Класс 5. Внесение элемента коммерциализации (и связанного с ней расширения области применения) в разработку декомпиляторов привело к появлению возможности управления процессом получения ИК, то есть к их полноценной интерактивности. Это позволило улучшить человеко-восприятие алгоритмов декомпилированного кода путем увеличения гибкости самого процесса. Из всех упомянутых декомпиляторов необходимым и достаточным набором интерактивных возможностей обладает лишь продукт IDA Pro с плагином Hex-Rays, который будем считать единственным представителем данного класса средств.

Класс 6. Приближением средств декомпиляции к заданной предметной области являются утилиты, на выходе которых создается некий псевдокод, предназначенный именно для описания алгоритмов, а никак не для восстановления ИК с последующей перекомпиляцией. На данный момент остается плохо развитыми не только теория создания таких ПС, но даже и сам формат псевдокода представления алгоритмов. Хотя известных инструментальных средств данного класса не существует, тем не менее, примером можно считать двухэтапное применение де-

компиляторов с отдельным преобразованием кода на их выходном языке программировании в псевдокод. Впрочем, в этом случае теряется интерактивность из-за необходимости сложного согласования этапов, не дающего удовлетворительного управления всем процессом алгоритмизации.

Класс 7. Последним элементом в списке средств алгоритмизации, являются утилиты преобразования МК в описание алгоритмов, позволяющие управлять самим процессом. В отличие от декомпиляторов в язык программирования, они стремятся предоставлять лишь необходимую, лаконичную и сформированную нужным образом информацию об алгоритмах. При этом они являются более гибкими по сравнению с двухэтапными средствами, состоящими из раздельно работающих утилит; наличие же в выходном коде пометок о потенциальных уязвимостях повышает их поисковые возможности. К данному классу может быть отнесена только разработанная Утилита.

Этап 2. Формирование критериев оценки

Важнейшим этапом в создании Методики 1 является формирование критериев оценки, поскольку их необходимость и достаточность позволит определить наиболее удобное для использования средство алгоритмизации. Применительно к текущей задаче оценки введем следующие критерии.

1) Скорость работы. Чрезмерная продолжительность применения средства может свести на нет другие преимущества, например, достоверность восстанавливаемых алгоритмов (и, как следствие, найденных уязвимостей). Это в полной мере относится к ручному способу алгоритмизации кода, который может дать удовлетворительные результаты только после длительного периода времени. Для отдельного средства критерий имеет объективную оценку – время работы на шаблонных примерах, а для класса – субъективную, основанную на сравнении принципов их работы.

2) Трудоемкость. Чрезмерная сложность в использовании средства резко усиливает влияние человеческого фактора на качество выполняемой работы. В рамках данного критерия, графические ПС зачастую выигрывают у консольных.

Критерий имеет субъективную оценку, основанную на сравнении трудоемкостей использования средств.

3) Представление алгоритмов. Конечным результатом работы средства является алгоритм; при этом его вид никак не специфицирован и определяется концепцией самого средства. Удачный выбор формата алгоритмизированного Представления напрямую влияет на его человеко-восприятие, поэтому критерий имеет субъективную оценку. Сравнение средств по этому критерию даст явно лучшие результаты для блок-схем и псевдо-языков алгоритмов по сравнению с не предназначенными для этого Представлениями.

4) Информация об уязвимостях. Наличие в восстановленных алгоритмах дополнительной информации об уязвимостях напрямую влияет на успешность их поиска. Например, пометки о потенциальных уязвимостях в алгоритмах кода позволяют соответствующим Экспертам-М обнаружить их с большей вероятностью и за более короткое время. Критерий имеет логическую оценку, основанную на наличии соответствующего функционала.

5) Поддерживаемые процессоры. Использование средства, поддерживающего целое семейство процессоров МК, более разумно, чем «заточенного» лишь под один тип, так как само оно и сделанные наработки будут применимы и в случае исследования МК той же исходной программы, но скомпилированной для другого процессора. Критерий имеет объективную оценку, основанную на количестве поддерживаемых процессоров МК.

6) Применимость для различных типов уязвимостей. Уязвимости ранее были разделены на типы по структурному уровню их нахождения в коде: НУ, СУ и ВУ. И если уязвимости первого типа утрачивают свою актуальность из-за наличия большого количества автоматических средств их проверки и поиска, то обнаружение уязвимостей второго типа до сих пор является высокоприоритетным. Ошибки же в архитектуре (третий тип) так и вовсе оставлены без внимания, что обосновывается как высокой степенью их субъективной характеризации, так и малым количеством выявляемых на практике, хотя последствия от угроз для дан-

ного типа в разы превосходят остальные. Критерий имеет объективную оценку, основанную на количестве «искомых» типов уязвимостей.

7) Актуальность средства. Каким бы быстроработающим, низкотрудоемким с удачной нотацией Представления алгоритмов не было средство, однако моральное и техническое устаревание не позволит ему конкурировать по этому критерию с современными. Критерий имеет объективно-субъективную оценку и может быть оценен по наличию технической и методической поддержки средства производителем, дистрибьюторами и/или «коммьюнити».

8) Достоверность результатов. Алгоритмы, получаемые средством, должны соответствовать изначальным, реализованным с помощью МК. Данный критерий является очевидным, но трудно оцениваемым. Тем не менее, наличие работоспособного набора тестов и возможность их проверки положительно влияет на потребительские свойства средства. Критерий имеет логическую оценку, основанную на наличии тестирования средства.

9) Экономическая целесообразность. Полный расчет стоимости владения средством является отдельной, достаточно сложной технико-экономический задачей, поэтому будем оценивать критерий субъективно, основываясь на дешевизне средства. В этом случае бесплатно-распространяемые средства будут предпочтительнее коммерческих.

10) Распространенность. Данный критерий, соответствующий популярности, имеет важное значение для оценки, поскольку так или иначе влияет на большинство других путем их «усиления». Так, популярный продукт априори лучше поддерживается производителем, имеет дополнительное внешнее тестирование и поддержку современных процессоров. Критерий имеет субъективную оценку, основанную на множестве факторов влияния на известность: реклама, наличие демо- и онлайн-версий, упоминание в «блогосфере» и на конференциях, ссылки в научных статьях и т. п.

Этап 3. Определение локальных и глобальных приоритетов, выбор альтернативы

Используя выбранные на первом этапе альтернативы и заданные на втором критерии, необходимо провести оценки каждой альтернативы по критериям, произведя вычисление их локальных приоритетов. В случае объективной оценки альтернативы по критерию, ее локальному приоритету назначается соответствующее значение; факт наличия у средства некой характеристики также является объективной оценкой – логической (булевой), принимающей значения «истина» или «ложь». При наличии только субъективной оценки применяется метод попарных сравнений с использованием шкалы относительной важности.

Глобальные приоритеты альтернатив вычисляются с помощью локальной свертки всех приоритетов Иерархии, а именно заданных изначально приоритетов критериев и определенных в процессе локальных приоритетов альтернатив по каждому критерию. В результате каждой альтернативе будет назначен ее глобальный приоритет, учитывающий, как важность каждого критерия средства алгоритмизации, так и соответствие средства этому критерию.

Заключительным является выбор альтернативы по наибольшему глобальному приоритету – данное средство будет считаться обладающим наиболее удобным для алгоритмизации МК в интересах последующего поиска уязвимостей.

4.2.3 Исходные данные и результаты оценки

Разработанная Методика 1 позволяет сделать выбор наиболее удобного для использования средства алгоритмизации МК из предлагаемых классов на основании объективных и субъективных оценок по сформулированным критериям. Система последних может быть легко модифицирована под задачу аналогичной оценки любых средств, обрабатывающих МК, путем изъятия специализированных критериев под номерами 3, 4 и 6.

В качестве альтернатив воспользуемся минимально-достаточным их набором, а именно – разработанной Утилитой (имеющей реализацию в виде прототи-

па) и ее ближайшим конкурентом IDA Pro (включая плагин Hex-Rays). Все остальные с очевидностью будут иметь заведомо худшие потребительские свойства (т. е. заведомо меньшее значение глобального приоритета). Схема такой иерархической структуры оценки приведена на рисунке 4.3.

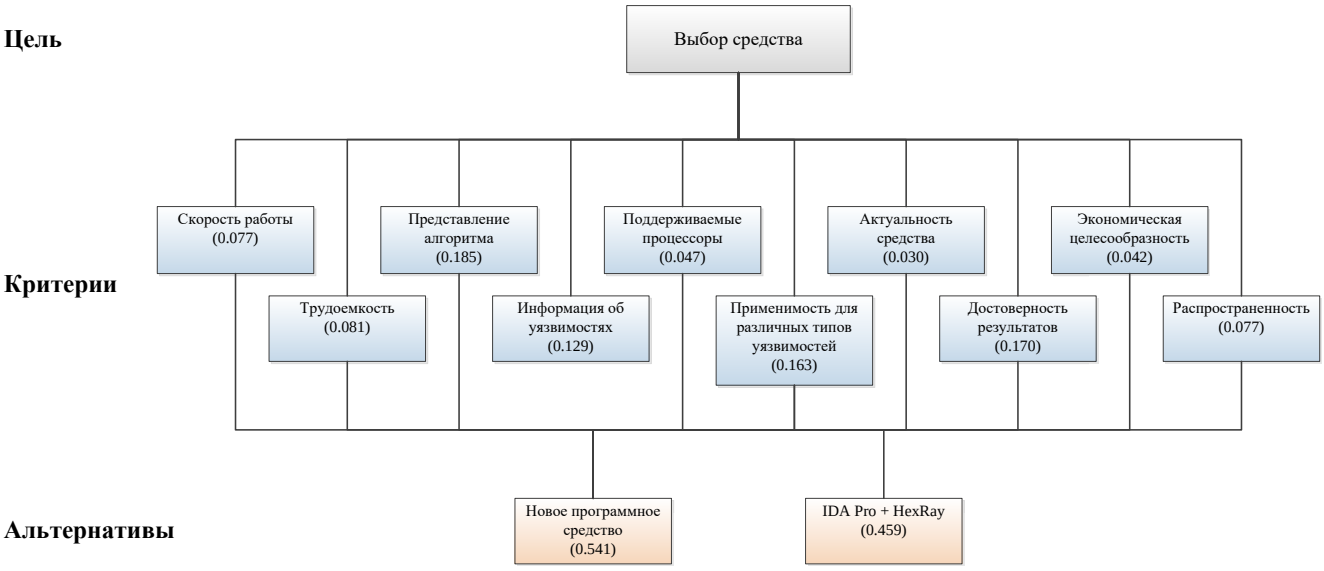

Рисунок 4.3 – Иерархия структуры оценки

Произведем попарное сравнение введенных критериев относительно их важности для выбора средства. В качестве результатов оценки будем использовать шкалу сравнений МАИ:

– 1, если одинаковы;

– 3, если умеренное превосходство;

– 5, если значительное превосходство;

– 7, если сильное превосходство;

– 9, если очень сильное превосходство;

– 2, 4, 6, 8 для промежуточных значений;

– 1/2, 1/3, …, 1/8, 1/9 для обратного сравнения.

Получим, таким образом, матрицу попарных сравнений критериев (далее – Матрицу) в виде:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix},$$

где a_{nm} – результат сравнения критериев n и m. При этом, следуя значениям шкалы сравнений МАИ верны следующие соотношения:

$$a_{ii} = 1,$$
$$a_{nm} = {}^{1}/_{a_{mn}}.$$

Согласно проведенной экспертной оценке, Матрица попарных сравнений критериев имеет вид, записанный в виде таблице 4.1 (сокращение К_N соответствует введенному критерию с номером N).

Таблица 4.1 – Матрица парных сравнений критериев потребительских свойств средств алгоритмизации

Матрица А		**Критерии**									
		К_1	**К_2**	**К_3**	**К_4**	**К_5**	**К_6**	**К_7**	**К_8**	**К_9**	**К_10**
Критерии	**К_1**	1	1/4	1/6	1/5	1	1/6	1/6	1/3	1/3	3
	К_2	4	1	1	1/2	3	1	2	1/2	1	3
	К_3	6	1	1	1/2	6	1	2	1	4	6
	К_4	5	2	2	1	6	1	1	1/2	2	4
	К_5	1	1/3	1/6	1/6	1	1/5	1/2	2	1/2	1
	К_6	6	1	1	1	5	1	4	1/4	3	4
	К_7	6	1/2	1/2	1	2	1/4	1	1/4	1	2
	К_8	3	2	1	2	1/2	4	4	1	3	3
	К_9	3	1	1/4	1/2	2	1/3	1	1/3	1	4
	К_10	1/3	1/3	1/6	1/4	1	1/4	1/2	1/3	1/4	1

Вычислим нормализованный вектор приоритетов (вес критериев), используя формулу среднего геометрического элементов каждой m-той строки Матрицы (здесь и далее S означает размер матрицы):

$$b_m^S = \sqrt[S]{\prod_{n=1}^{S} a_{nm}},$$

поделенную на сумму средних следующим образом:

$$p_m^S = \frac{b_m}{\sum_{i=1}^{S} b_i}.$$

Промежуточные и конечные расчеты для Матрицы парных сравнений критериев с размером S = 10 представлены в таблице 4.2.

Таблица 4.2 – Расчет компонент нормализованного вектора приоритетов критериев

		$\prod_{n=1}^{10} a_{nm}$	**Среднее геометрическое** b_m^{10}	**Компонент нормализованного вектора приоритетов** p_m^{10}
Критерии (m)	**К_1**	0.00008	0.38792	0.03238
	К_2	18	1.33514	0.11144
	К_3	864	1.96631	0.16412
	К_4	480	1.85406	0.15475
	К_5	0.00093	0.49734	0.04151
	К_6	360	1.80148	0.15036
	К_7	0.37500	0.90657	0.07567
	К_8	864	1.96631	0.16412
	К_9	0.33333	0.89596	0.07478
	К_10	0.00005	0.37011	0.03089
	Сумма		**11.98120**	**1.0**

Таким образом, значения приоритетов введенных критериев были определены следующим образом: скорость работы – 0.03238, трудоемкость – 0.11144, представление алгоритма – 0.16412, информация об уязвимостях – 0.15475, поддерживаемые процессоры – 0.04151, применимость для различных типов уязвимостей – 0.15036, актуальность средства – 0.07567, достоверность результатов – 0.16412, экономическая целесообразность – 0.07478, распространенность – 0.03089.

Оценим согласованность локальных приоритетов. Вычисление главного собственного значение Матрица дает следующий результат:

$$\lambda_{max} = \sum_{n=1}^{10} \left(\sum_{m=1}^{10} a_{nm} \right) \times b_n = 11.45968.$$

В этом случае индекс согласованности равен:

$$\mu = \frac{\lambda_{max} - 10}{10 - 1} = 0.16219,$$

а отношение согласованности (с учетом случайного индекса 1.49 для 10-ти критериев) равно:

$$\gamma = \frac{\mu}{1.49} = 0.10885.$$

Таким образом, отношение согласованности соответствует 11%, что является допустимым (т. е. не превосходящим 10-15%) и подтверждает корректность заполнение Матрицы. Аналогичным образом были определены приоритеты альтернатив – Утилиты и IDA Pro с плагином Hex-Rays – по каждому из критериев. Промежуточные и конечные расчеты для каждого из приоритетов представлены в таблице 4.3.

Таблица 4.3 – Расчет приоритетов альтернатив по каждому из критериев

		$\prod_{n=1}^{2} a_{nm}$		**Среднее геометрическое** b_m^2		**Компонент нормализованного вектора приоритетов** p_m^2	
Альтернативы		**Утилита**	**IDA+ Hex-Rays**	**Утилита**	**IDA+ Hex-Rays**	**Утилита**	**IDA+ Hex-Rays**
Критерии (m)	**К_1**	1	1	1	1	0.5	0.5
	К_2	2	0.5	1.41421	0.70711	0.66667	0.33333
	К_3	4	0.25	2	0.5	0.8	0.2
	К_4	3	0.33333	1.73205	0.57735	0.75	0.25
	К_5	0.25	4	0.5	2	0.2	0.8
	К_6	5	0.2	2.23607	0.44721	0.83333	0.16667
	К_7	0.5	2	0.70711	1.41421	0.33333	0.66667
	К_8	0.2	5	0.44721	2.23607	0.16667	0.83333
	К_9	3	0.33333	1.73205	0.57735	0.75	0.25
	К_10	0,14286	7,00000	0,37796	2,64575	0,125	0,87500

Глобальные приоритеты были получены с помощью линейной свертки, расчеты которых представлены в таблице 4.4.

Таблица 4.4 – Расчет глобальных приоритетов альтернатив

	Критерии и их локальный приоритет p_m										**Глобальный приоритет**
	K1	**K2**	**K3**	**K4**	**K5**	**K6**	**K7**	**K8**	**K9**	**K10**	
	0.032	0.111	0.164	0.155	0.042	0.15	0.076	0.164	0.075	0.031	
Утилита	0.5	0.667	0.8	0.75	0.2	0.833	0.333	0.167	0.75	0.125	0.584
IDA+ Hex-Rays	0.5	0.333	0.2	0.25	0.8	0.167	0.667	0.833	0.25	0.875	0.416

В случае Утилиты значение глобального приоритета было вычислено как 0.584, что превосходит 0.416 для IDA Pro.

Таким образом, практическое применение разработанной методики на группе алгоритмизирующих средств, включая новое ПС (Утилиту), показало очевидное преимущество последнего с точки зрения удобства использования по сравнению с аналогами.

4.3 Разработка и применение методики оценки функциональности средства алгоритмизации

Назначением Утилиты является получение такого представления, по которому Эксперта-М мог бы понять архитектуру и алгоритмы исследуемого МК. Очевидно, что как само конечное представление, так и способы превращения в него исходного, могут быть различными и уже реализованными в альтернативных средствах. Для обоснования превосходства Утилиты с позиции реализованного в ее Прототипе функционала необходимо произвести сравнение производимой им алгоритмизации с ближайшими аналогами. Для этого предлагается соответствующая Методика 2 – и приводятся результаты ее применения.

4.3.1 Выбор и обоснование способа оценки

Функциональность Утилиты задает преобразования МК, результаты которых могут быть определены по форме и содержанию СМД в конечном представлении, исходя из их содержания в начальном. Таким образом, в качестве способа

оценки может быть выбрано сравнение результатов работы оцениваемого и альтернативных средств алгоритмизации для базовых тестов, каждый из которых служит для проверки отдельного эффекта от алгоритмизации. Сравнение результатов может быть произведено экспертно-бальным методом.

В качестве ПрК, с помощью которого заданы тесты, может выступать как ИК, так и получаемый из него компилированием АК, или же получаемый ассемблированием МК. Использование МК крайне трудоемко для составления тестов, поскольку код имеет бинарную нечитаемую форму. Использование ИК не позволит точно задать суть тестирования, поскольку применяемые компиляторы будут генерировать не тождественный АК – добавляя избыточные инструкции (например, излишнее сохранение переменных на стеке) или изменяя алгоритмы (например, предсказание значений переменных в результате оптимизации). АК же позволит, как вручную составлять тесты (естественно, при наличии соответствующего навыка), так и делать алгоритмизируемый МК полностью им соответствующим.

4.3.2 Этапы методики оценки

Поскольку ближайшим конкурентом Утилиты можно считать IDA+Hex-Rays, то методику можно обоснованно адаптировать для сравнения именно этих двух ПС. Методика оценки – Методика 2 – состоит из следующих этапов. Первые шесть осуществляются для каждого из базовых тестов, последний для всех.

Этап 1. Базовый тест компилируется с получением МК для выбранного процессора (Утилита поддерживает PowerPC). Полученное таким образом Представление МК считается исходным для проведения алгоритмизации.

Этап 2. МК теста дизассемблируется с помощью IDA Pro с частичным восстановлением СМД, используя его встроенные средства.

Этап 3. Используя разработанный IDC-скрип для IDA Pro производится генерация ассемблерного Представления, подходящего для Утилиты.

Этап 4. Утилита применяется для ассемблерного Представления МК теста с получением его алгоритмизированного.

Этап 5. Аналогично Этапу 4, дизассемблированный код декомпилируется с помощью IDA+Hex-Rays в С-подобный псевдо-ИК.

Этап 6. Представления, полученных на предыдущих этапах, раздельно проверяются на предмет того, как в них были восстановлены СМД, присутствующие в исходном коде базового теста. Результаты проверок заносятся в сводную таблицу с указанием следующих значений: «-2» – восстановление произведено не корректно (в том числе уязвимость не найдена), «-1» – имеются некритичные отличия, «0» – практически полное соответствие, «+1» – полностью восстановлен и имеет более «удачный вид».

Этап 7. Сводная таблица проверок тестов с позиции восстановления их СМД средствами анализируется следующим образом. Для каждого из средств суммируются значения, поставленные по результатам их восстановления каждого из тестов. Средство с наибольшей суммой (не учитывая ее знак или равенство нулю) считается более функциональным с позиции алгоритмизации МК для последующего поиска уязвимостей.

Схема методики оценки с исходными, конечными и промежуточными данными, приведена на рисунке 4.4.

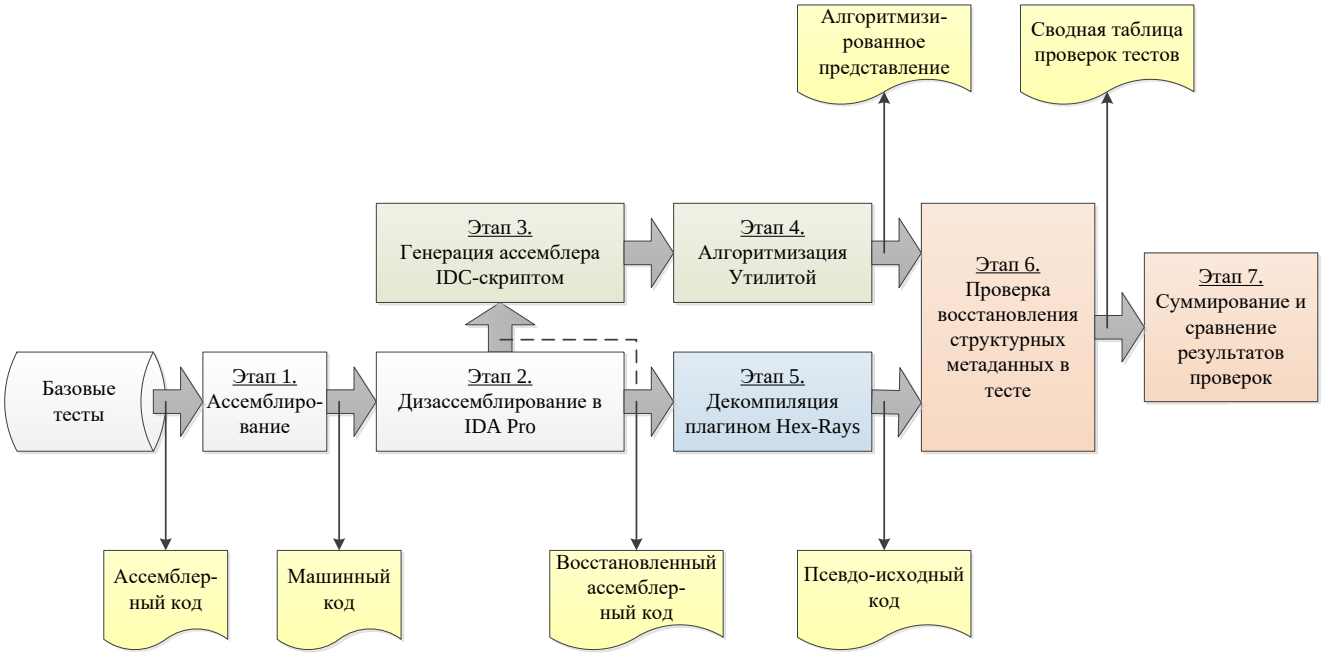

Рисунок 4.4 – Методика оценки функциональности программных средств алгоритмизации – Утилиты и IDA+Hex-Rays

4.3.3 База тестов и результаты проверок

База состоит из 9-ти ключевых тестов, каждый из которых предназначен для отражения эффекта алгоритмизации (заключенных в названии теста). Для каждого теста (в Приложении Е) описывается его предназначение и алгоритм работы, созданный вручную АК для процессора PowerPC, а так же прообраз ИК для пояснения работы теста. Также приводятся результаты работы Утилиты и IDA+Hex-Rays [73]. Сводная таблица проверок тестов для каждого из средств приведена далее (таблица 4.5).

Таблица 4.5 – Результаты проверок тестов для Утилиты и IDA+Hex-Rays

ПС	**Тест 1**	**Тест 2**	**Тест 3**	**Тест 4**	**Тест 5**	**Тест 6**	**Тест 7**	**Тест 8**	**Тест 9**	**Итог**
Утилита	-1	-1	+1	0	0	0	-1	+1	0	-1
IDA+Hex-Rays	0	0	0	-1	-1	-1	-2	0	-2	-7

Таким образом, условно считая, что веса всех тестов в общем итоговом сравнении ПС алгоритмизации примерно равные, Утилита имеет предпочтительное суммарное количество -1 балов против -7 балов для IDA+Hex-Rays. Данный результат усиливается тем, что практически все отставания Утилиты от IDA+Hex-Rays решаются путем добавления в нее необходимого функционала, являясь при этом чисто инженерной (программистской) задачей, не требующей изменения в архитектуре существующей реализации. В то же время, Утилита имеет возможности, применимые для автоматизированного поиска среднеуровневых уязвимостей.

4.4 Выбор и обоснование системы критериев оценки алгоритмизированности машинного кода

Поскольку восстановление алгоритмов и архитектуры ПрК, даже в рамках Метода, возможно различными способами с получением вариаций его формы, то необходима оценка алгоритмизированности исходного МК. Это позволит, как

объективно сравнить два Представления для одного МК, так и понять Эксперту-М, что полученное представление подходит для дальнейшего поиска уязвимостей.

Хотя оценка алгоритмизированности и является достаточно субъективным процессом, тем не менее, возможна ее частичная объективизация путем введения параметров, максимально не зависящих от различия мнений Экспертов-М [167-172]. Существует определенное количество метрик ПрК, частично применимых для оценки «качества» ПО; однако, они не подходят для оценки алгоритмизированности МК по следующим причинам. Во-первых, такие метрики предназначены в основном для оценки ИК. Во-вторых, они направлены на определение субъективного качества кодирования программистом (и, как следствие, его профессионального уровня), а не объективного – формы Представления кода. И, в-третьих, их основное применение заключается в будущей поддержке кода и его рефакторинге. Таким образом, они будут давать недостаточно корректные результаты для сгенерированного кода алгоритмов; сравнение же результатов метрик для результатов работы различных способов алгоритмизации МК так и вовсе даст приблизительное равенство (например, любым способом не будут генерироваться комментарии к коду, граф потока управления будет соответствовать машинному, количество функций и их взаимные вызовы будет идентичными и т. п.).

4.4.1 Метрики программного кода

Для выбора параметров (и, соответственно, критериев) оценки алгоритмизированности получаемого кода возьмем за основу и рассмотрим существующие и наиболее распространенные метрики ПрК [173], имеющие непосредственное отношение к описанию восстановленного алгоритмизированного Представления.

1) SLOC. Метрика (от англ. Source Line Of Code – количество строк кода), основана на подсчете количества строк кода в программе. Достоинством метрики является возможность оценки объема кода (статического и динамического), а недостатком – сильная зависимость от используемого языка и стиля программирования. В метрике выделяют два основных показателя: количество логических и

физических строк кода. Первый измеряет количество именно строк исходного текста программы, а второй – отдельных команд (для языка С, команды, как правило, разделяются знаками ‘;’ и границами блоков).

2) ОО-метрика. Объектно-ориентированная метрика используется исключительно для программ, созданных с применением ООП и может определяться сложностью классов (например, по количеству методов в нем), глубиной дерева наследования, количеством «детей» наследования и т. п. Достоинством, которое в зависимости от условий также переходит и в недостаток, является полная зависимость от применения ООП в ПрК.

3) Метрика Холстеда. Метрика основана на подсчете количества строк ПрК и содержащихся в них синтаксических элементов. Ее основными параметрами являются количество уникальных операторов и операндов, а также, общее их количество. В частности, по метрике вычисляется такой показатель, как усилие программиста при разработке кода. Достоинства и недостатки соответствуют метрике SLOC с тем исключением, что метрика Холстеда частично компенсирует недостатки, возникающие в случае нестандартных стилей программирования, таких, как написание нескольких функциональных блоков кода в одну строку.

4) Метрика цикломатической сложности. Наиболее распространенной метрикой оценки ПрК является определение цикломатической сложности, вычисленной по графу потока управления. Метрика показывает полное количество проходов выполнения программы по всем возможным условным ветвлениям. Недостатком метрики можно считать невозможность различения циклических и условных конструкций, а также отсутствие учета сложности логических выражений.

5) Метрика Чепина. Метрика используется для оценки сложности потока управления данными и оценивает характер использования переменных ввода/вывода. Метрика определяется такими параметрами, как количество вводимых переменных, модифицируемых или создаваемых внутри программы, управляющих и неиспользуемых. Очевидной особенностью метрики является учет лишь потока управления данных вне зависимости от потока управления кода.

6) Метрика Джилба. Метрика основана на определении насыщенности программы УС, такими, как IF-THEN-ELSE; циклы в данном случае являются частным случаем таких конструкций. Метрика достаточно хорошо отражает простоту ПрК, однако является слишком грубой; например, для случая десятка последовательных и вложенных конструкций IF-THEN-ELSE, метрика вычислит одинаковый результат, хотя во втором случае, очевидно, программа будет более сложной.

7) Метрика связанности. Метрика основана на определении взаимосвязи отдельных множеств сущностей ПрК. Так, в ней различаются: связанность по данным, структуре данных, управлению, общей области (глобальным переменным); логическая связанность сообщениями; подклассовая связанность и т. п. Достоинством метрики является возможность оценки качества отдельных модулей или целой архитектуры ПрК (что не достаточно хорошо делают другие метрики), недостатком же – слишком высокий используемый уровень абстракции (например, метрика может показать высокое качество для кода, написанного в одну строку с множеством дублируемых операций).

Следует отметить, что вопрос метрик алгоритмизированности кода является архисложным; однако он не требует полноценного решения, поскольку в рамках текущей задачи необходимо лишь создание базиса, который позволил бы сравнить ограниченный набор методов алгоритмизации. Успешностью выбранных параметров в этом случае будет являться однозначность в определении наиболее эффективного метода.

4.4.2 Объективные и субъективные параметры оценки

С позиции ручного анализа алгоритмизированного Представления МК с целью поиска уязвимостей можно выделить следующие параметры описания алгоритмов, поделенные на группы по признаку участия в них мнения человека – объективные и субъективные. На основании проведенного анализа существующих метрик ПрК в интересах оценки алгоримизированности кода будут следующие объективные параметры:

- количество существенных ошибок, в результате чего восстановленный алгоритм, не соответствует исходному;

- количество корректно выделенных простых управляющих СМД (одиночных условий) по отношению к исходному;

- количество корректно выделенных сложных управляющих СМД (вложенных условий, циклов) по отношению к исходному;

- количество корректно выделенных сигнатурных СМД (параметров подпрограмм, возвращаемых значений) по отношению к исходному;

- количество корректно выделенных вызовов подпрограмм по отношению к исходному;

- количество операторов безусловного перехода GOTO (в данном случае алгоритмизация более эффективна при меньшем количестве полученных операторов GOTO, поскольку это делает алгоритм более структурированным);

- количество используемых в выделенных алгоритмах переменных (в данном случае алгоритмизация более эффективна при меньшем количестве полученных переменных, поскольку делает суть алгоритма более понятной);

- количество излишних языковых конструкций (в данном случае алгоритмы тем более понятны, чем меньше в них несущественных текстовых участков с описанием – например, типов переменных, неиспользуемых вычислений и т. п.);

- количество корректно обнаруженных уязвимостей по отношению к исходным;

- количество дублируемой информации, которая визуально увеличивает форму алгоритма при неизменности содержания, что ухудшает его воспринимаемость;

- количество излишне-сложных конструкций, т. е. тех, которые могут быть заменены в рамках используемого языка описания алгоритмов на значительно более лаконичные (например, использование явного сравнения булевой переменной с 0-м вместо неявного при возврате из подпрограммы);

- циклическая сложность, определяемая, как количество линейно независимых маршрутов через ПрК,

а также следующие субъективные параметры:

- уровень абстракции алгоритмов, означающий насколько конечная детализация описания алгоритма необходима и достаточна для его понимания;
- плотность описания алгоритмов, означающая насколько большой объем алгоритма охватывается и воспринимается в заданном участке текста.

Объективные параметры могут быть использованы для формальной оценки и сравнения Метода с аналогами. Субъективные параметры могут быть использованы для экспертной оценки удовлетворительности описания алгоритмов, полученных с применением Метода.

4.5 Расчет метрики понятности представлений программного кода

Поскольку основная работа Эксперта-М заключается в поиске СУ и ВУ по алгоритмизированному представлению МК, то эффективность такого анализа будет определяющей для эффективности всего Метода. Поскольку анализ производится полностью вручную, то на его успешность непосредственно влияет удачность такого представления – его *понятность* для эксперта.

Смысл термина понятность сам по себе является крайне субъективным и его определение считается отдельной достаточно сложной задачей [174]. Обзор научных работ данной области показал отсутствие каких-либо значимых решений. Таким образом, была создана оригинальная метрика понятности описания (в том числе и представления ПрК), основанием для которой послужили следующие рассуждения и математические выкладки, приведенные далее [68]. Также, с помощью метрики было произведено сравнение представлений, используемых Утилитой и ближайшим аналогом – IDA+Hex-Rays.

4.5.1 Этапы осмысления информации

Понятность некого описания можно сопоставить с простотой единого осмысления заключенной в нем информации. Цель осмысления заключается в получении целостного содержания информации по форме ее описания; в данном

случае *форма* и *содержание* представляют диалектическую пару категорий. Такой процесс может быть условно поделен на 3 этапа, каждый из которых оперирует информацией в собственном смысловом представлении [175]. Очевидно, что первое представление получается из исходного описания информации, а последнее воссоздает конечный ее смысл. Для простоты будем считать, что информация состоит из отдельных элементов – каждый из которых имеет внутреннее содержание (собственный смысл), представляемое в некой форме (текущее описание).

Начальное описание информации является совокупностью (зачастую плохо систематизированной) форм элементов – многострочный текст для ПрК. Так, последний может быть представлен символьной строкой «*X* += 3;» (пример является сквозным в дальнейшем описании процесса).

Первым этапом процесса осмысления информации является анализ исходного описания – его Восприятие, посредством которого человек производит отображение внешнего реального мира во внутренний собственный. Смысловое представление данного этапа состоит из совокупности форм элементов информации без каких-либо знаний об их содержании. Пример ПрК на данном этапе преобразуется человеком из символьной строки в совокупность соответствующих лексем иерархического вида: «[*X*] <– [+][=] –> [3]»; при этом, точный смысл лексем пока не определен.

На втором этапе к формам элементов информации «навешивается» их содержание, т. е. осуществляется Понимание элементов. При этом, поскольку этап отражает субъективную деятельность человека, содержание (по связанной форме) может быть получено только из понятийного аппарата субъекта. Таким образом, осуществляется сопоставление истинного содержания элемента информации с известным человеку посредством аналогии его форм. Так, смысловое представление этапа состоит из совокупности форм элементов с назначенным человеком элементарным содержанием. Пример ПрК на данном этапе будет иметь следующий вид:

```
[X] -> (Переменная)
[+][=] -> (Сложение)
[3] -> (Число)
```

И хотя смысл каждой лексемы уже задан, тем не менее, общий смысл примера все так же не определен.

На третьем этапе – Интерпретации, используя понятые содержания элементов, синтезируется единое содержание исходной информации, являясь согласованным целым для человека. Смысловое представление этапа содержит только совокупность содержаний элементов в некой системе, отражающей итоговый смысл. Для примера ПрК оно будет соответствовать выражению «(Применение операции сложения к переменной)». Такой смысл полностью понятен человеку.

Следуя аналогии представлений и примеру обособленный смысл понятой информации, без привязки к человеку, может быть записан, как «(Увеличение переменной)». Описанная схема процесса осмысления информации человеком со сквозным примером приведена на рисунке 4.5.

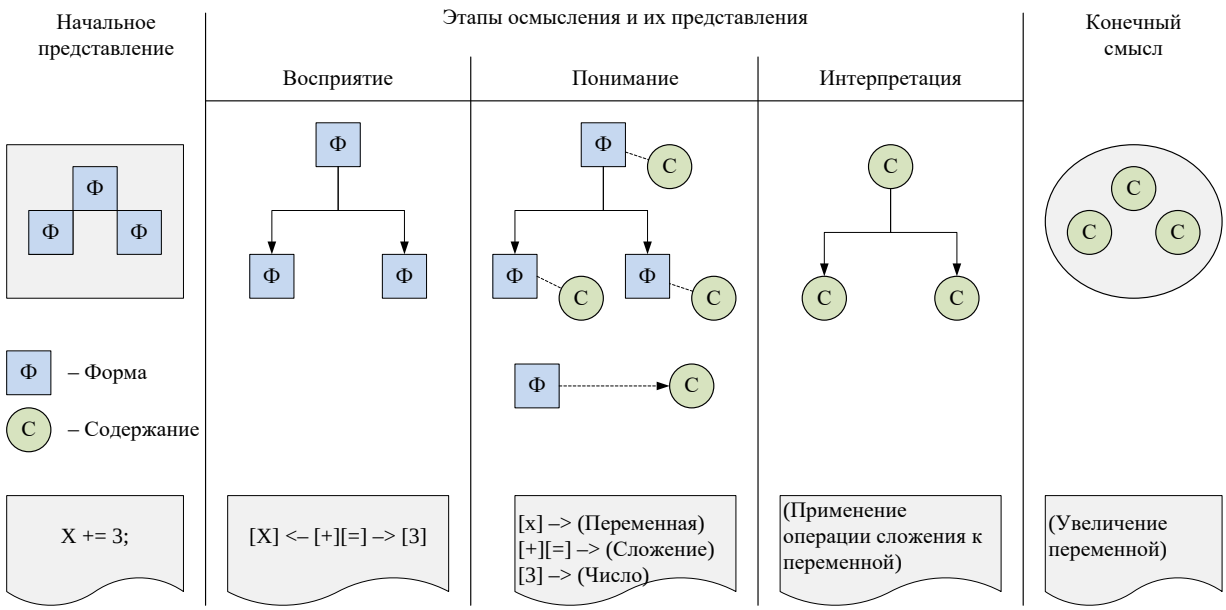

Рисунок 4.5 – Схема осмысления информации человеком

4.5.2 Критерии понятности представлений

Согласно описанным этапам процесса осмысления – преобразованию начальной формы информации в конечный ее смысл – понятность человеком исходного описания полностью определяется понятностью каждого из промежуточ-

ных представлений. Такие представления используются человеком для следующего – *осмысление хранящейся в них информации согласно известной ему логики*. Последнее позволяет выделить три следующие критерия понятности:

1) Содержательность, или наличие необходимой и достаточной информации; ее недостаток ухудшит корректность подаваемой информации, а избыток лишней (так называемый «сахар») приведет к запутыванию и выполнению человеком лишней работы;

2) Человеко-ориентированность, или удобство информации для человека, ее привычность; чем более сложными будут приемы, используемые в представлении, тем затруднительнее будет его понять;

3) Логичность, или последовательность и предсказуемость подачи информации; выявление логики позволит человеку осознавать еще не полученную информацию, «забегать вперед», как и успешнее хранить уже осознанную.

Точное значение каждого критерия для отдельных этапов осмысления информации приведено в таблице 4.6.

Таблица 4.6 – Значение критериев понятности для этапов осмысления информации

	Восприятие	**Понимание**	**Интерпретация**
Содержательность	Наличие особенностей (без «сахара») в формах конструкций для анализа	Соответствие объема смысла конструкций и размеров ее формы	Наличие требуемого (и только) смысла конструкций для общего синтеза
Человеко-ориентированность	Читаемость и привычность языка представления	Сложность необходимого понятийного аппарата	Привычность смысла конструкций для единого осмысления человеком
Логичность	Последовательность в нотации описания конструкций	Типовые закономерности в сопоставлении конструкциям их смысла	Использование смысловых шаблонов

4.5.3 Метрика понятности описания

Совокупность критериев понятности и этапов осмысления позволяет ввести метрику понятности описания (в том числе и исследуемого представления – описывающего алгоритмы и архитектуру ПрК) в виде 3-х компонентного вектора:

$$M = \langle P, U, I \rangle,$$

где P, U и I – мера Восприятия (Perception), Понимания (Understanding) и Интерпретации (Interpretation) исходного описания. Очевидно, что большие значения метрики будут означать и высокую эффективность понимания соответствующего представления на каждом этапе осмысления.

Поскольку понятность каждого смыслового представления этапа можно определить по введенным критериям, то компоненты вектора вычисляются следующим образом:

$$\begin{cases} P = F_m(C_P, H_P, L_P) \\ U = F_m(C_U, H_U, L_U), \\ I = F_m(C_I, H_I, L_I) \end{cases} \quad (1)$$

где C_m, H_m, L_m – значения критериев Содержательности (Content), Человекоориентированности (Human-Oriented) и Логичности (Logical) для соответствующих мер m из P, U и I; а $F_m()$ – функция вычисления меры по критериям.

Абсолютные значения компонент вектора трудно определимы; тем не менее, в интересах оценки эффективности достаточно сравнение понятностей «конкурирующих» представлений (а именно, Метода и ближайшего аналога – продукта IDA Pro с плагином декомпиляции Hex-Rays), т. е. вычислить их разницу:

$$\Delta M = M_1 - M_2 = \langle \Delta P, \Delta U, \Delta I \rangle. \quad (2)$$

Выделив у теоретического множества оцениваемых представлений уникальные особенности, можно получить тем самым их влияние на общую понятность по каждому из критериев, а именно: $S_i^{C_m}$, $S_i^{H_m}$ и $S_i^{L_m}$, где i – специальный индекс i-ой особенности.

Тогда вычисление критериев из (1) может быть записано, как:

$$\begin{cases} C_m = F_c\left(S_1^{C_m}, \cdots, S_N^{C_m}\right) \\ H_m = F_c\left(S_1^{H_m}, \cdots, S_N^{H_m}\right), \\ L_m = F_c\left(S_1^{L_m}, \cdots, S_N^{L_m}\right) \end{cases} \quad (3)$$

где $F_c()$ – функция вычисления значения критерия понятности представления по всем его особенностям.

Исходя из особенностей оцениваемых представлений и их анализа экспертом сделаем следующие упрощения.

Во-первых, равнозначность и независимость критериев понятности *C*, *H* и *L* позволяет утверждать, что их относительный вклад в меры *P*, *U* и *I* будет одинаковым и раздельным – критерии имеют равный приоритет. Таким образом функция вычисления мер из (1) является аддитивной и линейной, имеющей следующий вид:

$$F_m(C,H,L) \equiv F_m(C) + F_m(H) + F_m(L) \equiv F_m(C+H+L). \quad (4)$$

Во-вторых, выборка особенностей, в рамках одного представления не влияющих на другие, позволит вычислять вклад каждой особенности в критерии понятности также независимо, и, следовательно, функция $F_c()$ из (3) будет иметь вид:

$$F_c\left(S_1^{K_m},\cdots,S_N^{K_m}\right) \equiv F_{c1}\left(S_1^{K_m}\right) + \cdots + F_{cN}\left(S_N^{K_m}\right) \equiv \sum_{i=1}^{N} F_{ci}\left(S_i^{K_m}\right), \quad (5)$$

где *K* – индекс, соответствующий общим критериям C, *H* и *L*; $F_{ci}()$ – функция вычисления вклада каждой *i*-ой особенности в критерий понятности представления.

В-третьих, будем считать при сравнении метрик в (2) что особенности у второго представления отсутствуют, а все особенности первого задают его отличие от второго (произведя нормировку влияния особенностей первого представления по второму и перейдя к их относительным значениям), т. е. выражение (5) будет следующим:

$$F_C\left(S_1^{K_m},\cdots,S_N^{K_m}\right) = 0, \text{ для } M \equiv M_2. \quad (6)$$

Таким образом, будет произведен переход от абсолютных метрик описаний к относительной метрики разницы их понятностей.

Согласно выражениям (1), (3)–(6), разница метрик (2) приобретает вид:

$$\Delta M = \langle F_m(\Delta C_P + \Delta H_P + \Delta L_P), F_m(\Delta C_U + \Delta H_U + \Delta L_U), F_m(\Delta C_L + \Delta H_L + \Delta L_L)\rangle$$
$$= \langle \sum_{i=1}^{N}\sum_{K\in C,H,L}\Delta F_{ci}\left(S_i^{K_P}\right), \sum_{i=1}^{N}\sum_{K\in C,H,L}\Delta F_{ci}\left(S_i^{K_U}\right), \sum_{i=1}^{N}\sum_{K\in C,H,L}\Delta F_{ci}\left(S_i^{K_I}\right)\rangle.$$

Смысл выражения $\Delta F_{ci}\left(S_i^{\mathrm{k}m}\right)$ согласно упрощению (6) заключается в относительном влиянии *i*-ой особенности первого представления на значение критерия понятности *k* для меры *m* по сравнению со вторым представлением. В качестве возможных значений выражения целесообразно ввести следующую оценочную шкалу влияния: +2 – всегда влияет положительно; +1 – как правило, улучшает; 0 – никак не влияет; -1 – как правило, ухудшает; -2 – всегда влияет отрицательно.

Разумеется, шкала во многом субъективна; тем не менее, она корректно отражает логическую ситуацию.

4.5.4 Расчет метрики понятности для метода алгоритмизации

Рассчитаем введенную разницу метрик (2) для представления, получаемого Методом, и обобщенного представления кода, получаемого декомпиляторами и основанного на результатах работы продукта IDA Pro с плагином Hex-Rays (таблица 4.7). Особенности представления Метода по сравнению с «конкурентом» (имеющие обозначения O_1…O_15) выделены путем анализа синтаксиса и семантики языков представления, функциональных возможностей, принципов работы, а также динамических свойств уязвимостей и т. п. Таким образом, будет получена метрика понятности первого представления относительно второго.

Таблица 4.7 – Расчет метрики понятности для алгоритмизированных представлений Метода относительно IDA Pro + Hex-Rays

Особенности представления	**Восприятие (P)**			**Понимание (U)**			**Интерпретация (I)**			**Сумма**
	C	**H**	**L**	**C**	**H**	**L**	**C**	**H**	**L**	
O_1. Работа с битами	0	1	0	1	1	0	-1	0	0	2
O_2. Универсальные циклы	-2	1	-1	1	1	1	2	1	1	5
O_3. Лаконизация конструкций	1	1	0	2	-1	0	1	0	0	4
O_4. Корректировка восстановления алгоритмов	-2	-1	1	-1	-2	1	1	2	-1	-2
O_5. Корректировка для предсказания уязвимостей	-2	-1	1	-1	-2	1	1	1	-1	-3
O_6. Информация об уязвимостях	1	1	0	-2	1	0	1	2	0	4

Особенности представления	Восприятие (P)			Понимание (U)			Интерпретация (I)			Сумма
	C	H	L	C	H	L	C	H	L	
O_7. Многоуровневый выход из цикла	1	-1	1	0	-1	1	1	1	1	4
O_8. Глобальные переменные с адресами	1	-1	0	1	-1	0	-1	0	1	0
O_9. Выделение модулей	1	1	1	1	-1	1	2	1	1	8
O_10. Единый файл с представлением	-1	-1	-1	0	0	-1	0	-2	-1	-7
O_11. Отсутствие поддержки типов	-1	-2	-1	0	1	-1	1	0	0	-3
O_12. Классическое структурное программирование	-2	-1	1	-1	1	2	2	-1	1	2
O_13. Указание регистров процессора (доп.)	1	-2	0	-1	-1	0	-1	0	1	-3
O_14. Диаграмма модульной архитектуры (доп.)	1	1	0	-1	0	1	1	0	1	4
O_15. Отсутствие встроенной блок-схемы алгоритмов (доп.)	-1	-1	-1	1	1	-1	0	-1	-2	-5
Сумма	-3	-4	2	-1	-4	6	10	5	4	15
	-5			1			19			

Результаты проведенного сравнения могут быть также представлены на 3-х мерной диаграмме, для которой горизонтальные оси соответствуют особенностям и критериям, а вертикальная – влиянию первых на вторые согласно введенной шкале (рисунок 4.6).

Как хорошо видно из таблицы 4.7, хотя общее Восприятие представления ухудшилось, тем не менее, Понимание остается таким же, а Интерпретация имеет более высокий показатель. Вектор метрики соответствует следующему:

$$M = \langle -5, 1, 19 \rangle,$$

«близость» же представлений может быть оценена с помощью модуля вектора, а именно:

$$L = |M| = \sqrt{P^2 + U^2 + I^2} = \sqrt{(-5)^2 + 1^2 + 19^2} = \sqrt{387} \approx 19.7. \quad (7)$$

Значение выражения (7) имеет смысл только при сравнении близостей нескольких представлений (более 2-х) и означает уровень различности их понимания человеком.

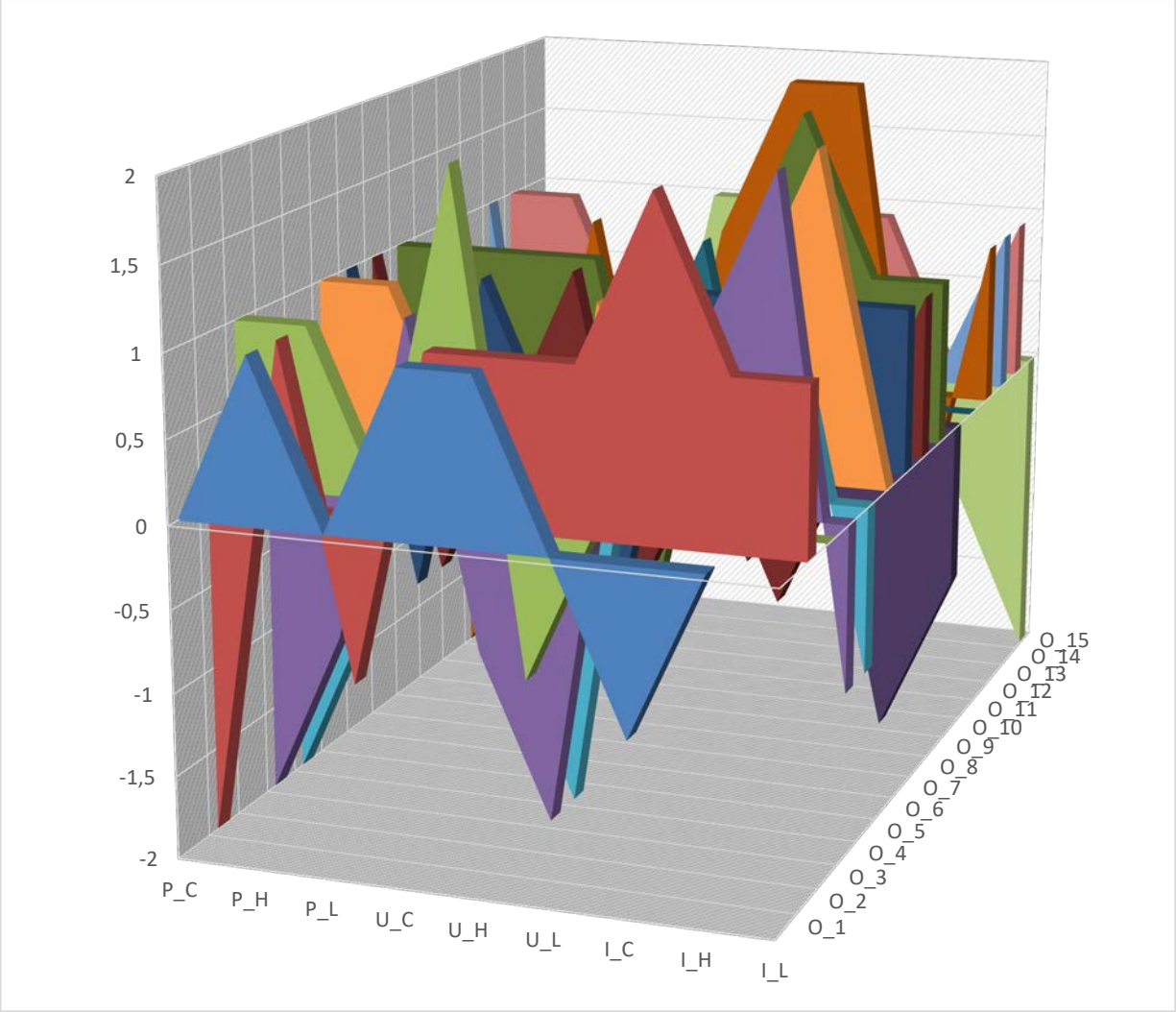

Рисунок 4.6 – Гистограмма сравнения алгоритмизированных представлений Метода и IDA Pro+Hex-Rays

Значимость каждого из этапов в процессе осмысления представления человеком является отдельным вопросом. Тем не менее, можно предположить два следующих способа определения их взаимной важности и влияния на итоговую понятность.

Во-первых, при краткосрочной работе с исходным представлением на первое место выходит уровень его Восприятия – без него человек не сможет даже разобрать текст на лексемы, за которым следует уровень Понимания – новые неизвестные формы также не дадут никакой информации для Интерпретации и синтезирования конечного смысла. В этом случае отрицательная мера Восприятия при практически нулевой мере Понимания могут перевесить высокую меру Интерпретации, сделав представление Метода менее понятным, чем альтернативные.

Во-вторых, при долгосрочной работе подразумевается, что эксперт изначально будет обучен (единовременно и за адекватное конечное время) основам работы с представлением, что таким образом значительно снизит непривычность Восприятия (при неизменной обычности Понимания). А высокая мера Интерпретации позволит осознавать исходное представление значительно эффективнее. Исходя из больших размеров МК и общей сложности задачи поиска уязвимостей, второй подход к определению итоговой понятности представления его алгоритмов и архитектуры считается более корректным. Таким образом, алгоритмизированное представление МК, получаемое Методом более выгодно.

4.6 Разработка методики оценки эффективности поиска уязвимостей с использованием алгоритмизации машинного кода

Разработанный Метод является частью целевого способа, осуществляющего поиск уязвимостей с использованием алгоритмизации МК. Следовательно, оценить Метод с точки зрения применимости для достижения конечной цели – повышения эффективности поиска уязвимостей в МК – можно только косвенным образом, через способ поиска.

Оценка Метода с позиции поиска уязвимостей возможна отдельной, предлагаемой далее, Методикой 3, вычисляющей количественный показатель найденных уязвимостей в МК для выбранного способа поиска [63]. Сравнение результатов применения такой методики в случае данного и других способов поиска по МК позволит обоснованно указать рейтинг каждого из них, косвенно оценив при этом и входящий в состав данного способа метод алгоритмизации.

В качестве эффективности поиска уязвимостей будем использовать введенную ранее, а именно – скорость обнаружения уязвимостей. Тогда, затрата разумно-ограниченного времени на поиск уязвимости будет соответствовать обнаружению уязвимости, а затрата чрезмерно большого времени – невозможность ее обнаружить. Такая интерпретация термина позволяет перейти от задачи расчета эффективности теоретической возможности выявления любой уязвимости в услови-

ях бесконечного времени (которая практически всегда равно 100%) к более реалистичной – поиску уязвимостей за разумное время экспертами с типовыми аналитическими способностями.

4.6.1 Условия и этапы оценки

В качестве условий для выполнения Методики 3 выделим следующие. Во-первых, необходимо создать группы тестов, содержащие типовые уязвимости (только для СУ и ВУ), согласно успешности выявления которых и будет оцениваться выбранный способ. Во-вторых, поиск уязвимостей в МК должен осуществляться соответствующими экспертами каждым из способов – путем ручного анализа АК, с применением ПС декомпиляции или иными. И, в-третьих, результатом применения методики оценки должен быть численный коэффициент, определяющий эффективность поиска уязвимостей с использованием заданного способа. Таким образом, этапы методики логично разделить на две части. Первая часть независима от способа поиска и может быть выполнена единожды – на подготовительном этапе; ее результатом будет набор указанных групп тестов. Вторая часть будет зависеть от каждого способа поиска, выполняться на заданном наборе тестов и количественно определять найденные уязвимости. В результате, каждому из способов будет сопоставлен некий числовой коэффициент эффективности, по которому они могут быть сравнены. Рассмотрим этапы Методики 3 более подробно.

Подготовительный этап. Создание групп тестов

Суть подготовительного этапа сводится к созданию групп тестов, являющихся неизменными для основных этапов Методики 3. Под тестами из каждой группы будем подразумевать АК подпрограммы и глобальных переменных, содержащих отдельную уязвимость. Пример ИК на языке С, по которому может быть получен АК такого теста, приведен ниже.

```
const char *admin_password = “SuperPassword”;  // length = 13+1
```

```
int check_admin_password(const char *psw)
{
    int res;
    res = strncmp(psw, admin_password, 5); // length for compare = 5
    return res;
}
```

Примечание. Уязвимость в приведенном примере заключается в том, что в реализации функции сравнения пароля администратора с введенным участвует лишь 5 символа, вместо 14-ти. Эта СУ может быть не случайной, а созданной программистом преднамеренно для упрощения подбора пароля.

Деление на группы произведем с помощью таблицы, полученной путем сопоставления следующих списков:

- каталога категорий уязвимостей Национальной Базы Данных Уязвимостей (National Vulnerability Database, NVD), являющейся информационным хранилищем правительства США с автоматизированным управлением;
- набора из двух элементов, соответствующих случайному и преднамеренному источнику возникновению уязвимостей.

Каталог NVD состоит из 34-х категорий, две из которых не будут рассматриваться, как не применимые. Таким образом, размер таблицы будет равен 32 × 2, что соответствует 64-м группам тестов. Очевидно, что усредненные тесты из каждой группы должны в некотором смысле иметь одинаковую сложность.

Каждой группе тестов целесообразно присвоить определенный коэффициент угрозы K_T, влияющей на последствия от реализации содержащихся в них уязвимостей. Коэффициент логично вычислять на основании свойств столбцов и строк, в которых расположена соответствующая группа. Так, сопоставим с каждой группой уязвимостей коэффициент опасности K_V следующим образом: $K_V = 1$ для низко-опасных уязвимостей, $K_V = 2$ для средне-опасных и $K_V = 3$ для высоко-опасных. Значение $K_V = 0$ целесообразно зарезервировать для тестов, не имеющих уязвимостей. Случайные же и преднамеренные источники будут лишь по-разному влиять на коэффициент угрозы, а именно через коэффициент усиления K_S со значениями 1 и 2 соответственно.

Этап 1. Применение способа поиска к машинному коду

Оцениваемый способ должен быть применен для каждого теста всех групп, созданных на Подготовительном этапе. К сожалению, способы поиска имеют существенные различия в применении и поэтому детализация этапа невозможна. Каждый способ должен быть применен экспертами с целью обнаружения в МК заданной уязвимости за разумно-ограниченный промежуток времени. После обнаружения уязвимости (или невозможности этого в случае нехватки времени или недостаточности квалификации) пометка о результате должна быть занесена в таблицу с указанием экспертно-субъективной характеристики сложности поиска. Ей может служить коэффициент сложности поиска K_D со следующими значениями: 0 в случае невозможности обнаружения уязвимости, 1 – если эксперт посчитал процесс поиска достаточно сложным, 2 – если поиск оказался вполне типичным, и 3 – если обнаружение уязвимости носило тривиальный характер. Коэффициент сложности поиска для всей группы можно получить усреднением по ее отдельным тестам. Пример части такой таблицы (для случая низко-опасных групп уязвимостей и равных сложностей поиска по группам) показан ниже (таблица 4.8).

Таблица 4.8 – Пример таблицы отчета о сложности поиска уязвимостей K_D для группы тестов

Источник уязвимости	**Категория уязвимостей NVD**				
	NVD_1, $K_V = 1$	NVD_2, $K_V = 1$	NVD_3, $K_V = 1$	NVD_4, $K_V = 1$	NVD_5, $K_V = 1$
Случайный, $K_S = 1$	0	1	2	3	2
Преднамеренный, $K_S = 2$	1	1	2	3	3

Таким образом, по окончании этапа для всех оцениваемых способов будут установлены формализованные характеристики по поиску каждой из групп уязвимостей.

Этап 2. Вычисление коэффициента эффективности метода

Объединив значения внесенных в таблицу коэффициентов K_D для всех групп (например, с помощью их умножения на вес K_T (полученному из K_V и K_S) с последующим алгебраическим суммированием) можно получить единый коэффициент K_E, дающий общую оценку эффективности конкретного способа поиска уязвимостей в МК. Пример вычисления коэффициента K_E для таблицы 4.8 будет следующим:

$$K_E = 1 \times (1 \times (0 + 1 + 2 + 3 + 2) + 2 \times (1 + 1 + 2 + 3 + 3)) = 28.$$

Отметим, что значение коэффициента в данном случае безразмерно и имеет смысл лишь в контексте его сравнения с полученными для других способов.

Достоинством данной методики является сведение плохо формализуемой оценки поиска уязвимостей в МК к числовому значению, притом с учетом, как опасностей уязвимостей, так и их источников возникновения.

Недостатками же методики является достаточно сильное «смазывание» эффектов каждой группы тестов с уязвимостями и использование субъективных характеристик (хотя и определенных при помощи экспертов). Впрочем, если применение Методики 3 на реальных примерах покажет удовлетворительные результаты (то есть, различия в значениях коэффициента K_E будут существенными), то данный недостаток во внимание можно не принимать.

4.6.2 Пример оценки эффективности поиска уязвимостей с использованием разработанной методики

В качестве детализации особенностей применения Методики 3 и для обоснования ее работоспособности проведем мысленный эксперимент оценки ею Метода для группы тестов, содержащих одну уязвимость.

На Подготовительном этапе создания группы тестов в качестве примера ИК возьмем функцию проверки корректности пароля, аналогичную из специализированного примера в пункте 3.4.3, а именно следующую:

```
#define TRUE 1
#define FALSE 0
int check_password(const char *pw, const char *pw_ok) {
    if (strcmp(pw, "SuperPassword") == 0)
        return TRUE;
    if (strcmp(pw, pw_ok) == 0)
        return TRUE;
    return FALSE;
}
```

Примечание. Пример описывает реализацию алгоритма функции проверки пароля, возвращающую TRUE для совпадения и FALSE в противоположном случае. Уязвимость заключается в том, что функция вернет TRUE также и для встроенного в код пароля “SuperPassword”, что является так называемым «черным ходом» или «backdoor».

Уязвимость относиться к категории NVD «1: Authentication Issues» − неправильная аутентификации пользователя, и для ТКУ является высоко-опасной (K_V = 3). Источник уязвимости является преднамеренным (K_S = 2), что еще более усиливает «критичность» ее эксплуатации.

На Этапе 1 производиться экспертный поиск уязвимости по алгоритмизированному Представлению, полученному с помощью Метода и аналогичному следующему:

```
check_password(pw, pw_ok) {
    _w0 = 0
    if (strcmp(pw, "SuperPassword")).false? {
        _w0 = 1;
    } else if (strcmp(pw, pw_ok).false?) {
        _w0 = 1;
    }
    return _w0;
}
```

Очевидно, что эксперт без труда обнаружит аномалию в коде в виде проверки пароля на соответствие встроенному кодовому слову "SuperPassword". Таким образом, сложность поиска будет им оценена, как тривиальная, соответствующая K_D= 3. В этом случае, итоговый отчет эксперта в табличной форме будет иметь следующий вид (таблица 4.9).

Таблица 4.9 – Пример таблицы отчета о сложности поиска уязвимостей K_D для одного теста

Источник уязвимости	**Категория уязвимостей (NVD)**
	NVD_1, K_V = 3
Преднамеренный, K_S = 2	3

В результате выполнения Этапа 2 будет вычислена итоговая эффективность поиска уязвимостей с использованием алгоритмизированного Представления МК:

$$K_E = 2 \times (3 \times (3)) = 18.$$

Отметим, что данное значение эффективности для одной группы тестов является максимально-возможным. Аналогичным образом, пример применения Методики 3 может быть расширен как на все группы уязвимостей, так и на другие способы. Сравнение полученных значений K_E позволит выявить наиболее эффективный для поиска уязвимостей способ; для использующего алгоритмизированное Представление это будет означать достоверность эффективного применения разработанного Метода в интересах поиска уязвимостей в МК.

Выводы по разделу 4

В качестве объекта исследовались метод и программное средство алгоритмизации машинного кода на предмет целевой оценки их эффективности для осуществления поиска уязвимостей.

В ходе исследования:

1) Описан целевой способ поиска уязвимостей в машинном коде, состоящий из применения автоматизированного метода алгоритмизации с последующим ручным анализом полученного представления. Определены объекты целевого способа для оценки их эффективности.

2) Разработана методика оценки потребительских свойств средств алгоритмизации машинного кода для поиска уязвимостей на основе Метода анализа иерархий. В интересах метода произведены классификация всех известных средств алгоритмизации и формирование критериев оценки. Применение разработанной методики показало преимущество предложенного программного средства по сравнению с ближайшим аналогом.

3) Разработана методика оценки функциональности средств алгоритмизации машинного кода для поиска уязвимостей на основе экспертно-бального метода. Применение разработанной методики показало преимущество алгоритмизации, производимой средством, по сравнению с ближайшим аналогом.

4) Проанализированы метрики программного кода, имеющие непосредственное отношение к описанию алгоритмизированного представления: количества строк кода, объектно-ориентированная, Холстеда, цикломатической сложности, Чепина, Джилба, связности.

5) Выбрана и обоснована система критериев алгоритмизированности машинного кода, состоящая из объективных (количество ошибок, корректно выделенные структурные метаданные и вызовов подпрограмм, операторы безусловного перехода, используемые переменные, лишние или излишне-сложные конструкции, обнаруженные уязвимости, дублируемая информация, а так же циклическая сложность) и субъективных (уровень абстракции и плотность описания алгоритмов) параметров.

6) Создана метрика понятности представления программного кода, связанная с этапами его осмысления и их критериями понятности. Расчет метрик для алгоритмизированного представления машинного кода и ближайшего аналога показал объективное преимущество первого.

7) Разработана методика оценки эффективности поиска уязвимостей с использованием алгоритмизации машинного кода. Произведен мысленный эксперимент оценки методикой предложенного метода алгоритмизации.

Основным научным результатом, изложенным в четвертом разделе, является комплекс научно-методических средств оценки алгоритмизации машинного кода в интересах поиска уязвимостей, суть которого заключается в раздельной оценке объектов целевого способа поиска уязвимостей: средства алгоритмизации, получаемого представления машинного кода и его последующего ручного анализа.

Частным научным результатом, изложенным в четвертом разделе, является база тестов для комплексной проверки эффектов алгоритмизации. Этот результат непосредственно был использован в методике оценки функциональности средства алгоритмизации.

Основное содержание раздела и полученных научных результатов изложено в работах автора [63, 65, 68, 73].

ЗАКЛЮЧЕНИЕ

В работе исследовался машинный код с уязвимостями на предмет восстановления алгоритмов его работы.

В соответствии с целевой установкой были решены следующие задачи:

1) Проанализированы способы нейтрализации уязвимостей в программном коде;

2) Идентифицированы метаданные в машинном коде, соответствующие парадигмам разработки программного кода для телекоммуникационных устройств;

3) Исследованы модели машинного кода с позиции структурных метаданных;

4) Построена гипотетическая схема алгоритмизации машинного кода на базе его модели;

5) Синтезирован метод, реализующий схему алгоритмизации;

6) Разработана архитектура программного средства автоматизации метода алгоритмизации;

7) Реализован прототип программного средства алгоритмизации и произведено его базовое тестирование;

8) Сформированы критерии и созданы методики для оценки свойств алгоритмизации машинного кода;

9) Произведена комплексная оценка эффективности алгоритмизации в интересах поиска уязвимостей;

10) Предложены перспективы применения и развития метода алгоритмизации.

В ходе решения указанных задач были получены основные научные результаты, выносимые на защиту, а именно:

1) Структурная модель машинного кода с уязвимостями;

2) Метод алгоритмизации машинного кода;

3) Архитектура программного средства алгоритмизации машинного кода;

4) Комплекс научно-методических средств оценки алгоритмизации машинного кода в интересах поиска уязвимостей.

Полученные результаты являются достоверными, обладают необходимой степенью новизны, имеют теоретическую ценность и практическую значимость, апробированы и опубликованы в 31-м научном труде.

Кроме того, в работе получен ряд частных научных результатов. Во-первых, критерии эффективности известных способов поиска уязвимостей, структурная типизация уязвимостей программного кода, схема областей жизни уязвимостей в представлениях программного обеспечения, линейная L-модель и промежуточные S-модели машинного кода. Во-вторых, критерии и результаты сравнительной оценки существующих способов и подходов к поиску уязвимостей с точки зрения их применимости для метода алгоритмизации. В-третьих, базовое тестирование разработанного прототипа программного средства алгоритмизации. В-четвертых, база тестов для комплексной проверки эффектов алгоритмизации.

Совокупность полученных научных результатов свидетельствует о достижении поставленной цели исследования – повышена эффективность поиска средне- и высокоуровневых уязвимостей в машинном коде телекоммуникационных устройств путем его алгоритмизации.

Перспективность метода алгоритмизации машинного кода должна заключаться не только в его текущем применении, но и в теоретико-практической значимости новых потенциальных результатов. Таким образом, исследования могут быть продолжены по следующим направлениям.

Во-первых, существует востребованность в «глубоком» ручном поиске уязвимостей в машинном коде. В ряде случаев требуется осуществление поиска низкоуровневых уязвимостей в машинном коде ручным способом, для чего может быть взято за основу его алгоритмизированное представление (поскольку в структурной модели метаданные связаны с инструкциями процессора). Такое взаимоувязанное представление может быть успешно использовано экспертом для одновременного понимания, как общей логики работы кода, так и низкоуровневых деталей его реализации. Новым перспективным направлением развития метода ал-

горитмизации является разработка интерактивной среды для визуализации ассемблерного кода (аналогично продукту IDA Pro) со встроенным синхронизированным отображением его управляющих структур, алгоритмов, модулей и архитектуры.

Во-вторых, плановая проверка неизменности основного функционала программного кода может быть приравнена к мероприятию по обеспечению информационной безопасности. Одним из способов поддержания программного обеспечения в защищенном состоянии является проверка неизменности архитектуры и алгоритмов его работы, гарантирующая отсутствие внесенных средне- и высокоуровневых уязвимостей. При этом штатное обновление программного обеспечения, производящее небольшие (низкоуровневые) модификации кода, не будет вносить «шума» в результаты восстановления. Направление развития метода алгоритмизации заключается в создании механизма сравнения логики работы машинного кода на основании его алгоритмизированного представления, не учитывающего низкоуровневые отличия.

В-третьих, автоматизированный поиск средне- и высокоуровневых уязвимостей в машинном коде повысит скорость обработки кода без значительной потери достоверности результатов. Хотя метод алгоритмизации и не предназначен для непосредственного (т.е. в рамках своей работы) поиска уязвимостей автоматическим способом, тем не менее, разработка формальных шаблонов средне- и высокоуровневых уязвимостей и добавление механизма их обнаружения позволит реализовать такой функционал – что является отдельным направлением его развития. С точки зрения архитектуры программного средства алгоритмизации данный функционал реализуем, как в существующем Модуле поиска уязвимостей, так и в дополнительном – на Стадии 3.

В-четвертых, разработка концепции и формата алгоритмизированного представления позволит максимально эффективно воспринимать алгоритмы и архитектуру кода экспертом. Для этого необходимо проведение отдельного исследования на предмет формализации процесса и критериев осознания программного кода человеком, а также развития метрики понятности представлений кода.

Выполнение перспективных исследований по перечисленным направлениям позволит повысить общую безопасность программного обеспечения телекоммуникационных устройств по количественным показателям, таким как число найденных разноуровневых уязвимостей, время поиска, степень покрытия кода и др.

Качественное же повышение безопасности программного обеспечения может лежать в плоскости новых концептуальных подходов к разработке программного кода, а также среды его выполнения. Последние гипотетически могут быть сформированы путем перехода от классической дискретной схемы представлений программного кода к максимально непрерывной.

СПИСОК ЛИТЕРАТУРЫ

1. Буйневич, М.В. Организационно-техническое обеспечение устойчивости функционирования и безопасности сетей связи общего пользования / М.В. Буйневич, А.Г. Владыко, С.М. Доценко, О.А. Симонина. – СПб.: СПбГУТ, 2013. – 144 с.

2. Буйневич, М.В. Безопасность современных информационных технологий: монография / В.Л. Бройдо, В.Н. Бугорский, М.В. Буйневич [и др.]; под общ. ред. Е.В. Стельмашонок. – СПб.: СПбГИЭУ, 2012. – 408 с.

3. Буйневич, М.В. Решения проблемных вопросов информационной безопасности телекоммуникационных систем (по результатам научных исследований СПбГУТ) / М.В. Буйневич // Информационная безопасность в современном обществе – Росинфоком-2016: материалы научно-практической конференции. – 2016. – С. 33-36.

4. Гольдштейн, Б.С. Softswitch / А.Б. Гольдштейн, Б.С. Гольдштейн. – СПб.: БХВ-Петербург, 2006. – 368 с.

5. Гольдштейн, Б.С. Сети связи: учебник для ВУЗов / Б.С. Гольдштейн, Н.А. Соколов, Г.Г. Яновский. – СПб.: БХВ-Петербург, 2010. – 400 с.

6. Гольдштейн, Б.С. Законный перехват сообщений: подходы ETSI, CALEA и СОРМ / Б.С. Гольдштейн, В.С. Елагин // Вестник связи. – 2007. – № 3. – С. 66-72.

7. Казарин, О.В. Вредоносные программы нового поколения – одна из существенных угроз международной информационной безопасности / О.В. Казарин, Р.А. Шаряпов // Вестник РГГУ. Серия: Документоведение и архивоведение. Информатика. Защита информации и информационная безопасность. – 2015. – № 12(155). – С. 9-23.

8. Коржик, В.И. Основы криптографии: учебное пособие / В.И. Коржик, В.А. Яковлев // СПб.: Интермедия, 2016. – 296 с.

9. Коржик, В.И. Использование модели канала с шумом для построения стегосистемы / М. Алексеевс , В.И. Коржик, К.А. Небаева // Телекоммуникации. – 2013. – № S7. – С. 33-36.

10. Котенко, И.В. Открытые программные средства для обнаружения и предотвращения сетевых атак / А.А. Браницкий, И.В. Котенко // Защита информации. Инсайд. – 2017. – № 2(74). – С. 40-47.

11. Котенко, И.В. Исследование бот-сетей и механизмов защиты от них на основе имитационного моделирования / А.М. Коновалов, И.В. Котенко, А.В. Шоров // Известия Российской академии наук. Теория и системы управления. – 2013. – № 1. – С. 45.

12. Крук, Е.А. Распределенная верификация результата агрегации данных в сенсорных сетях / Е.А. Крук, А.Д. Фомин // Программные продукты и системы. 2007. – № 2. – С. 31.

13. Крук, Е.А. Точная корректирующая способность кодов Гилберта при исправлении пакетов ошибок / Е.А. Крук, А.А. Овчинников // Информационно-управляющие системы. – 2016. – № 1(80). – С. 80-87.

14. Кучерявый, А.Е. Сети связи пост-NGN / Б.С. Гольдштейн, А.Е. Кучерявый. – СПб.: БХВ-Петербург, 2013. – 160 с.

15. Кучерявый, А.Е. Метод обнаружения беспилотных летательных аппаратов на базе анализа трафика / Р.В. Киричек, А.А. Кулешов, А.Е. Кучерявый // Труды учебных заведений связи. – 2016. – Т. 2. – № 1. – С. 77-82.

16. Кучерявый, А.Е. Интернет вещей как новая концепция развития сетей связи / П.Н. Боронин, А.Е. Кучерявый // Информационные технологии и телекоммуникации. – 2014. – № 3(7). – С. 7-30.

17. Соколов, Н.А. Телекоммуникационные сети: монография в 4 частях. Ч. 4. Эволюция инфокоммуникационной системы / Н.А. Соколов. – М.: Альварес Паблишинг, 2008. – 191 с.

18. Соколов, Н.А. Системные аспекты организации ситуационного центра / А.В. Пинчук, А.И. Поташов, Н.А. Соколов // Вестник связи. – 2007. – № 5. – С. 62-65.

19. Юсупов, Р.М. Софтверизация – путь к импортозамещению? / С.В. Кулешов, Р.М. Юсупов // Труды СПИИРАН. – 2016. – № 3(46). – С. 5-13.

20. Юсупов, Р.М. Концептуальная и формальная модели синтеза киберфизических систем и интеллектуальных пространств / О.О. Басов, А.Л. Ронжин, Б.В. Соколов, Р.М. Юсупов // Известия высших учебных заведений. Приборостроение. – 2016. – Т. 59. – № 11. – С. 897-905.

21. Юсупов, Р.М. Разработки модельного законодательства для сферы информационной безопасности / И.Л. Бачило, М.А. Вус, Р.М. Юсупов, О.С. Макаров // Власть. – 2015. – № 11. – С. 5-9.

22. Язов, Ю.К. Информационные риски в условиях применения технологии виртуализации в информационно-телекоммуникационных системах / Ю.К. Язов, В.Н. Сигитов // Информация и безопасность. – 2013. – Т. 16. – № 3. – С. 403-406.

23. Яковлев, В.А. Распределение ключей в беспроводных сетях с подвижными объектами на основе использования MIMO каналов с квантованием фазы сигнала / П.Д. Мыльников, В.А. Яковлев // Проблемы информационной безопасности. Компьютерные системы. – 2016. – Т. 1. – С. 102-113.

24. Баранов, А.П. Возможности импортозамещения в компьютерных технологиях России / А.П. Баранов // Проблемы информационной безопасности. Компьютерные системы. – 2014. – № 4. – С. 9-16.

25. Баранов, А.П. Актуальные проблемы в сфере обеспечения информационной безопасности программного обеспечения / А.П. Баранов // Вопросы кибербезопасности. – 2015. – № 1(9). – С. 2-5.

26. Баранов, А.П. О практике применения в защищенных информационных системах программных продуктов иностранного производства / А.П. Баранов // Проблемы информационной безопасности. Компьютерные системы. – 2007. – № 1. – С. 11-14.

27. Еремеев, М.А. Анализ методов распознавания вредоносных программ / М.А. Еремеев, А.В. Кравчук // Вопросы защиты информации. – 2014. – № 3. – С. 44-51.

28. Еремеев, М.А. Использование средств статического анализа исходного кода программ при разработке и отладке приложений / М.А. Еремеев, С.С. Захар-

ченко // Информационные технологии на железнодорожном транспорте: доклады XIV Международной научно-практической конференции. – 2010. – С. 82-85.

29. Зегжда, Д.П. Подход к решению задачи защиты АСУ ТП от киберугроз / Д.П. Зегжда, Т.В. Степанова // Проблемы информационной безопасности. Компьютерные системы. – 2013. – № 4. – С. 32-39.

30. Зегжда, П.Д. Современные проблемы кибербезопасности / П.Д. Зегжда, А.Г. Ростовцев // Неделя науки СПбПУ: сборник тезисов докладов форума с международным участием. – 2014. – С. 262-268.

31. Иванников, В.П. Статический анализатор svace для поиска дефектов в исходном коде программ / А.И. Аветисян, А.А. Белеванцев, А.Е. Бородин, Д.М. Журихин, В.П. Иванников, В.Н. Игнатьев, М.И. Леонов // Труды Института системного программирования РАН. – 2014. – Т. 26. – № 1. – С. 231-250.

32. Иванников, В.П. Автоматический поиск ошибок синхронизации в приложениях на платформе Android / С.П. Вартанов, В.П. Иванников, М.К. Ермаков // Труды Института системного программирования РАН. – 2013. – Т. 24. – С. 191-206.

33. Корниенко, А.А. Показатели эффективности выявления уязвимостей при использовании метода проверки на модели / С.Е. Ададуров, С.С. Захарченко, А.А. Корниенко // Интеллектуальные системы на транспорте: материалы IV Международной научно-практической конференции. – 2014. – С. 211-213.

34. Ломако, А.Г. Метод выявления дефектов и недокументированных возможностей программ / М.А. Еремеев, А.Г. Ломако, В.А. Новиков // Информационное противодействие угрозам терроризма. – 2010. – № 14. – С. 46-49.

35. Ломако, А.Г. Подход к выявлению потенциально опасных дефектов в спецификациях документов, регламентирующих порядок создания и сертификации средств защиты информации / Д.Н. Бирюков, М.А. Еремеев, А.Г. Ломако, П.В. Мажников // Проблемы информационной безопасности. Компьютерные системы. – 2015. – № 3. – С. 7-16.

36. Осипов, В.Ю. Обоснование мероприятий информационной безопасности / И.А. Носаль, В.Ю. Осипов // Информационно-управляющие системы. – 2013. – № 2(63). – С. 48-53.

37. Швед, В.Г. Исследование угроз информационного воздействия руткит на динамические данные ядра посредством мониторинга гостевой операционной системы виртуальной машины / А.Н. Люльченко, А.А. Менщиков, В.И. Милушков, А.А. Митрушин, В.Г. Швед // Глобальный научный потенциал. – 2015. – № 10(55). – С. 29-32.

38. Швед, В.Г. Метод и модель анализа безопасности операционной системы от атак типа руткит / А.Н. Люльченко, А.А. Менщиков, В.И. Милушков, А.А. Митрушин, В.Г. Швед // Перспективы науки. – 2015. – № 10(73). – С. 100-103.

39. Аветисян, А.И. Восстановление структуры бинарных данных по трассам программ / А.И. Аветисян, А.И. Гетьман // Труды Института системного программирования РАН. – 2012. – Т. 22. – С. 95-118.

40. Аветисян, А.И. Масштабируемый и точный поиск клонов кода / А.Ш. Аветисян, А.А. Белеванцев, Ш.Ф. Курмангалеев, С.В. Саргсян // Программирование. – 2015. – № 6. – С. 9-17.

41. Падарян, В.А. Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ / В.В. Каушан, А.Ю. Мамонтов, В.А. Падарян, А.Н. Федотов // Труды Института системного программирования РАН. – 2015. – Т. 27. – № 2. – С. 105-126.

42. Трошина, Е.Н. Инструментальная среда восстановления исходного кода программы – декомпилятор TyDec / Е.Н. Трошина, А.В. Чернов // Прикладная информатика. – 2010. – № 4(28). – С. 73-97.

43. Трошина, Е.Н. Исследование и разработка методов декомпиляции программ: дис. ... канд. физ.-мат. наук: 05.13.11 / Трошина Екатерина Николаевна. – М., 2009. – 134 с.

44. Трошина, Е.Н. Структурный анализ в задаче декомпиляции / Е.О. Деревенец, Е.Н. Трошина // Прикладная информатика. – 2009. – № 4(22). – С. 87-99.

45. Чернов, А.В. О некоторых задачах обратной инженерии / К.Н. Долгова, А.В. Чернов // Труды Института системного программирования РАН. – 2008. – Т. 15. – С. 119-134.

46. Чернов, А.В. О некоторых задачах анализа и трансформации программ // А.А. Белеванцев, С.С. Гайсарян, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова, А.В. Чернов // Труды Института системного программирования РАН. – 2004. – Т. 5. – С. 7-40.

47. Cifuentes, C. Decompilation of binary programs / C. Cifuentes, K.J. Gough // Software: Practice and Experience. – 1995. – Vol. 25. – Iss. 7. – PP. 811-829.

48. Cifuentes, C. Reverse compilation techniques / C. Cifuentes. – Brisbane: Queensland University Of Technology, 1994. – 324 p.

49. Диасамидзе, С.В. Подходы к метрическому оцениванию программных средств на этапе разработки / С.В. Диасамидзе, С.Н. Подкасик // Интеллектуальные системы на транспорте: материалы IV Международной научно-практической конференции. – 2014. – С. 488-491.

50. Диасамидзе, С.В. Метод выявления недекларированных возможностей программ с использованием структурированных метрик сложности: дис. ... канд. техн. наук: 05.13.19 / Диасамидзе Светлана Владимировна. – СПб., 2012. – 161 с.

51. Марков, А.С. Эвристический анализ безопасности программного кода / А.С. Марков, В.А. Матвеев, А.А. Фадин, В.Л. Цирлов // Вестник Московского государственного технического университета им. Н.Э. Баумана. Серия: Приборостроение. – 2016. – № 1(106). – С. 98-111.

52. Марков, А.С. Реализационные основы сигнатурно-эвристического анализа безопасности программ / А.С. Марков, А.А. Фадин // Региональная информатика и информационная безопасность: сборник трудов. – СПб.: СПОИСУ, 2015. – С. 204-205.

53. Марков, А.С. Систематика уязвимостей и дефектов безопасности программных ресурсов / А.С. Марков, А.А. Фадин // Защита информации. Инсайд. – 2013. – № 3(51). – С. 56-61.

54. Новиков, Ф.А. Дискретная математика для программистов: учебное пособие / Ф.А. Новиков. – М.: Питер, 2008. – Сер. Учебник для вузов. – 383 с.

55. Новиков, Ф.А. Моделирование на uml: теория, практика, видеокурс / Д.Ю. Иванов, Ф.А. Новиков. – СПб.: Профессиональная литература. – Сер. Избранное Computer Science. – 2010. – 635 с.

56. Поляков, С.В. Матричный способ представления алгоритма / В.С. Поляков, С.В. Поляков // Теоретические, методологические и прикладные вопросы науки и образования: материалы Международной научно-практической конференции. – 2016. – С. 238-245.

57. Эдель, Д.А. Языковая модель исполнимых кодов / Д.А. Эдель // Доклады Томского государственного университета систем управления и радиоэлектроники. – 2010. – № 1(21). – С. 56-60.

58. Shneiderman, B. Visualizing change over time using dynamic hierarchies: TreeVersity2 and the StemView / J.A. Guerra-Gómez, M.L. Pack, C. Plaisant, B. Shneiderman // IEEE Transactions on Visualization and Computer Graphics. – 2013. – Vol. 19. – Iss. 12. – PP. 2566-2575.

59. Shneiderman, B. Flowchart techniques for structured programming / I. Nassi, B. Shneiderman // ACM Sigplan Notices. – 1973. – Vol. 8. – Iss. 8. – PP. 12-26.

60. Израилов, К.Е. Метод алгоритмизации машинного кода телекоммуникационных устройств / М.В. Буйневич, К.Е. Израилов // Телекоммуникации. – 2012. – № 12. – С. 2-6.

61. Израилов, К.Е. Модель прогнозирования угроз телекоммуникационной системы на базе искусственной нейронной сети / К.Е. Израилов // Вестник ИНЖЭКОНа. Серия: Технические науки. – 2012. – № 8(59). – С. 150-153.

62. Израилов, К.Е. Автоматизированное средство алгоритмизации машинного кода телекоммуникационных устройств / М.В. Буйневич, К.Е. Израилов // Телекоммуникации. – 2013. – № 6. – С. 2-9.

63. Израилов, К.Е. Укрупненная методика оценки эффективности автоматизированных средств, восстанавливающих исходный код в целях поиска уязвимо-

стей / А.Ю. Васильева, К.Е. Израилов, А.И. Рамазанов // Вестник ИНЖЭКОНа. Серия: Технические науки. – 2013. – № 8(67). – С. 107-109.

64. Израилов, К.Е. Структурная модель машинного кода, специализированная для поиска уязвимостей в программном обеспечении автоматизированных систем управления / М.В. Буйневич, К.Е. Израилов, О.В. Щербаков // Проблемы управления рисками в техносфере. – 2014. – № 3(31). – С. 68-74.

65. Израилов, К.Е. Методика оценки эффективности средств алгоритмизации, используемых для поиска уязвимостей / К.Е. Израилов // Информатизация и связь. – 2014. – № 3. – С. 39-42.

66. Израилов, К.Е. Архитектурные уязвимости моделей телекоммуникационных сетей [Электронный ресурс] / М.В. Буйневич, А.Г. Владыко, К.Е. Израилов, О.В. Щербаков // Научно-аналитический журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России». – 2015. – № 4. – С. 86-93.– Режим доступа: http://vestnik.igps.ru/wp-content/uploads/V74/14.pdf.

67. Израилов, К.Е. Проблемные вопросы нейтрализации уязвимостей программного кода телекоммуникационных устройств / М.В. Буйневич, К.Е. Израилов, Д.И. Мостович, А.Ю. Ярошенко // Проблемы управления рисками в техносфере. – 2016. – № 3(39). – С. 81-89.

68. Израилов, К.Е. Система критериев оценки способов поиска уязвимостей и метрика понятности представления программного кода / К.Е. Израилов // Информатизация и связь. – 2017. – № 3. – С. 111-118.

69. Izrailov, K. Method and utility for recovering code algorithms of telecommunication devices for vulnerability search / M. Buinevich, K. Izrailov // 16th International Conference on Advanced Communication Technology (ICACT-2014). – 2014. – PP. 172-176.

70. Izrailov, K. Method for partial recovering source code of telecommunication devices for vulnerability search / M. Buinevich, K. Izrailov, A. Vladyko // 17th International Conference On Advanced Communications Technology (ICACT-2015). – 2015. – PP. 76-80.

71. Izrailov, K. The life cycle of vulnerabilities in the representations of software for telecommunication devices / M. Buinevich, K. Izrailov, A. Vladyko // 18th International Conference On Advanced Communications Technology (ICACT-2016). – 2016. – PP. 430-435.

72. Izrailov, K. Method and prototype of utility for partial recovering source code for low-level and medium-level vulnerability search / M. Buinevich, K. Izrailov, A. Vladyko // 18th International Conference on Advanced Communication Technology (ICACT-2016). – 2016. – PP. 700-707.

73. Izrailov, K. Testing of Utilities for Finding Vulnerabilities in the Machine Code of Telecommunication Devices / M. Buinevich, K. Izrailov, A. Vladyko // 19th International Conference on Advanced Communication Technology (ICACT-2017). – 2017. – PP. 408-414.

74. Израилов, К.Е. Утилита восстановления алгоритмов работы машинного кода: свидетельство о государственной регистрации программы для ЭВМ / К.Е. Израилов. – рег. № 2013618433. – 09.09.2013.

75. Израилов, К.Е. Анализ состояния в области безопасности программного обеспечения / К.Е. Израилов // Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО-2013): сборник научных статей II Международной научно-технической и научно-методической конференции. – 2013. – С. 874-877.

76. Израилов, К.Е. Сравнительный анализ подходов к поиску уязвимостей в программном коде / М.В. Буйневич, К.Е. Израилов, Д.И. Мостович // Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО-2016): сборник научных статей V Международной научно-технической и научно-методической конференции. – 2016. – С. 256-260.

77. Израилов, К.Е. Расширение языка «С» для описания алгоритмов кода телекоммуникационных устройств [Электронный ресурс] / К.Е. Израилов // Информационные технологии и телекоммуникации. – 2013. – № 2(2). – С. 21-31. – Режим доступа: http://www.sut.ru/doci/nauka/review/2-13.pdf.

78. Израилов, К.Е. Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного

кода. Часть 1. Функциональная архитектура [Электронный ресурс] / М.В. Буйневич, К.Е. Израилов // Информационные технологии и телекоммуникации. – 2016. – Т. 4. – № 1. – С. 115-130. – Режим доступа: http://sut.ru/doci/nauka/review/20161/115-130.pdf.

79. Израилов, К.Е. Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного кода. Часть 2. Информационная архитектура [Электронный ресурс] / К.Е. Израилов // Информационные технологии и телекоммуникации. – 2016. – Т. 4. – № 2. – С. 86-104. – Режим доступа: http://sut.ru/doci/nauka/review/20162/86-104.pdf.

80. Израилов, К.Е. Категориальный синтез и технологический анализ вариантов безопасного импортозамещения программного обеспечения телекоммуникационных устройств [Электронный ресурс] / М.В. Буйневич, К.Е. Израилов // Информационные технологии и телекоммуникации. – 2016. – Т. 4. – № 3. – С. 95-106. – Режим доступа: http://sut.ru/doci/nauka/review/20163/95-106.pdf.

81. Израилов, К.Е. Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного кода. Часть 3. Модульно-алгоритмическая архитектура [Электронный ресурс] / К.Е. Израилов // Информационные технологии и телекоммуникации. – 2016. – Т. 4. – № 4. – С. 104-121. – Режим доступа: http://www.sut.ru/doci/nauka/review/20164/104-121.pdf.

82. Израилов, К.Е. Архитектурные уязвимости программного обеспечения / К.Е. Израилов // Шестой научный конгресс студентов и аспирантов СПбГИЭУ (ИНЖЭКОН-2013): сборник тезисов докладов научно-практической конференции факультета информационных систем и экономике и управлении «Инфокоммуникационные технологии и математические методы». – 2013. – С. 35.

83. Израилов, К.Е. Алгоритмизация машинного кода телекоммуникационных устройств как стратегическое средство обеспечения информационной безопасности / К.Е. Израилов // Национальная безопасность и стратегическое планирование. – 2013. – № 2(2). – С. 28-36.

84. Израилов, К.Е. Рассмотрение представлений кода программ с позиций метаданных / К.Е. Израилов // Фундаментальные исследования и инновации в национальных исследовательских университетах: материалы Всероссийской научно-методической конференции. – 2012. – Т. 5. – С. 176-180.

85. Израилов, К.Е. Модель машинного кода, специализированная для поиска уязвимостей / М.В. Буйневич, К.Е. Израилов, О.В. Щербаков // Вестник Воронежского института ГПС МЧС России. – 2014. – № 2(11). – С. 46-51.

86. Израилов, К.Е. Язык описания модели безопасности телекоммуникационной сети / А.Ю. Васильева, К.Е. Израилов // Новые информационные технологии и системы (НИТИС-2012): сборник трудов X Международной научно-технической конференции. – 2012. – С. 272-275.

87. Израилов, К.Е. Метод и программное средство восстановления алгоритмов машинного кода телекоммуникационных устройств для поиска уязвимостей / К.Е. Израилов // Региональная информатика (РИ-2014): материалы XIV Санкт-Петербургской Международной конференции. – 2014. – С. 140-141.

88. Израилов, К.Е. Поиск уязвимостей в различных представлениях машинного кода / К.Е. Израилов // Информационная безопасность регионов России (ИБРР-2015): материалы IX Санкт-Петербургской межрегиональной конференции. – 2015. – С. 157.

89. Израилов К.Е. Исследование и моделирование угроз безопасности цифровой телекоммуникационной сети: отчет о НИР шифр «Цифровая угроза-2012» / М.В. Буйневич, К.Е. Израилов. – СПбГУТ, 2012. – рег. № 047-12-054. – 219 с.

90. Израилов К.Е. Разработка предложений по организационно-техническому обеспечению устойчивости функционирования сетей связи, защиты сетей связи от несанкционированного доступа к ним и передаваемой посредством их информации, методов проверки и определение перечня нарушений целостности, устойчивости функционирования и безопасности единой сети электросвязи Российской Федерации: отчет о НИР / М.В. Буйневич, А.Г. Владыко, К.Е. Израилов [и др.]. – СПб.: СПбГУТ, 2012; рег. № 034-12-054. – 177 с.

91. Андрушкевич, Д.В. Подход к построению защищенных распределенных сетей обработки данных на основе доверенной инфраструктуры / Д.В. Андрушкевич, В.М. Зима, С.В. Новиков // Труды СПИИРАН. – 2015. – № 1. – С. 34-57.

92. Буйневич, М.В. Беспроводные широкополосные сети vs безопасность / М.В. Буйневич, К.А. Горохова, О.А. Тиамийу // Фундаментальные исследования и инновации в национальных исследовательских университетах: материалы XVI Всероссийской научно-методической конференции. – СПб.: СПбГПУ, 2012. – С. 170-173.

93. Тиамийу, О.А. К вопросу о моделировании механизма доверенной маршрутизации / О.А. Тиамийу // Актуальные проблемы инфотелекоммуникаций в науке и образовании: материалы II-ой Междунар. науч.-техн. и науч.-метод. конф. – СПб.: СПбГУТ, 2013. – С. 879-882.

94. Борисов, М.А. Основы программно-аппаратной защиты информации / М.А. Борисов, И.В. Заводцев, И.В. Чижов. – М.: УРСС, 2013. – 370 с.

95. Щеглов, А.Ю. Защита компьютерной информации от несанкционированного доступа / А.Ю. Щеглов. – СПб.: Наука и техника, 2004. – 384 с.

96. Стандарт DES (FIPS 46-3) [Электронный ресурс]. – Режим доступа: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

97. Ян, С.Й. Криптоанализ RSA / С.Й. Ян. – М.: Ижевск: НИЦ «Регулярная и хаотическая динамика», Ижевский институт компьютерных исследований, 2011. – 312 с.

98. Стандарт MD5 (RFC 1321) [Электронный ресурс]. – Режим доступа: https://tools.ietf.org/html/rfc1321.

99. Лавриков, И.В. Обзор результатов анализа хеш-функций ГОСТ Р 34.11–2012 / И.В. Лавриков, Г.Б. Маршалко, В.И. Рудской, С.В. Смышляев, В.А. Шишкин // Проблемы информационной безопасности. Компьютерные системы. – 2015. – № 4. – С. 147-153.

100. Олифер В.Г. Компьютерные сети. Принципы, технологии, протоколы / В.Г. Олифер, Н.П. Олифер.– СПб.: Питер, 2010. – Глава 24. Сетевая безопасность. – С. 887-902. – 944 с.

101. Diffie, W. New Directions in Cryptography / W. Diffie, M.E. Hellman // IEEE Transactions on Information Theory. – 1976. – Vol. 22. – Iss. 6. – PP. 644–654.

102. Шнайер, Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си / Б. Шнайер. – М.: Триумф, 2002. — 816 с.

103. Семенов, Ю.А. Протокол SSL. Безопасный уровень соединителей [Электронный ресурс] / Ю.А. Семенов. – 2000. – № 1. – Режим доступа: http://book. itep.ru / 6/ssl_65.htm.

104. Астахова, Л.В. Защита облачной базы персональных данных с использованием гомоморфного шифрования / Л.В. Астахова, Н.А. Ашихмин, Д.Р. Султанов // Вестник ЮУрГУ. Серия: Компьютерные технологии, управление, радиоэлектроника. – 2016. – Т. 16. – № 3. – С. 52–61.

105. Миронов, С.В. Тестирование компиляторов на программные закладки / С.В. Миронов // Информационные технологии. – 2008. – № 8. – С. 61-64.

106. Гайсарян, С.С. Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения / С.С. Гайсарян, В.В. Каушан, Ш.Ф. Курмангалеев, А.Р. Нурмухаметов // Труды Института системного программирования РАН. – 2014. – Т. 26. – № 3. – С. 113-126.

107. Дрогин, В.В. Метод противодействия эксплуатации ошибок переполнения на основе доработки компилятора / В.В. Дрогин // Известия ЮФУ. Технические науки. – 2008. – № 1(78). – С. 113-114.

108. Марков, А.С. Концептуальные основы построения анализатора безопасности программного кода / А.С. Марков, А.А. Фадин, В.Л. Цирлов // Программные продукты и системы. – 2013. – № 1. – С. 10.

109. Чернов, А.В. Исследование информационной защищенности мобильных приложений / Я.А. Александров, Л.К. Сафин, К.Н. Трошина, А.В. Чернов // Вопросы кибербезопасности. – 2015. – № 4(12). – С. 28-37.

110. Среда выполнения Java Virtual Machine [Электронный ресурс]. – Режим доступа: http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

111. Среда выполнения Common Language Runtime [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/en-us/library/8bs2ecf4.

112. Программа JD-GUI [Электронный ресурс]. – Режим доступа: http://jd.benow.ca.

113. Программа .NET Reflector [Электронный ресурс]. – Режим доступа: http://www.reflector.net.

114. Мутилин, В.С. Анализ типовых ошибок в драйверах операционной системы Linux / В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов // Труды Института системного программирования РАН. – Т. 22. – 2012. – С. 349-374.

115. Общие уязвимости и воздействия (Common Vulnerabilities and Exposures) [Электронный ресурс]. – Режим доступа: https://cve.mitre.org.

116. Национальная база данных уязвимостей (National Vulnerabilities Database) [Электронный ресурс]. – Режим доступа: https://nvd.nist.gov.

117. Аграновский, А.В. Преобразование программного кода для использования уязвимостей переполнения буфера / А.В. Аграновский, С.И. Карнюша, Р.Н. Селин // Известия ЮФУ. Технические науки. – 2006. – № 7(62). – С. 92-96.

118. Юричев, Д. Reverse Engineering для начинающих [Электронный ресурс] / Д. Юричев. – 2017. – 992 с. – Режим доступа: https://beginners.re/RE4B-RU.pdf.

119. Жадаев, А.Г. Антивирусная защита ПК / А.Г. Жадаев. – СПб.: БХВ-Петербург, 2010. – 224 с.

120. Городняя, Л.В. О классификациях парадигм программирования и параллельном программировании / Л.В. Городняя // Образовательные ресурсы и технологии. – 2016. – № 2(14). – С. 138-144.

121. Городняя, Л.В. Парадигма программирования: курс лекций / Л.В. Городняя. – Новосибирск: НГУ, 2015. – 206 с.

122. Пароннджанов, В.Д. Язык Дракон. Краткое описание / В.Д. Пароннджанов. – М., 2009. – 124 с.

123. Набор инструкций PowerPC [Электронный ресурс]. – Режим доступа: http://www.nxp.com/assets/documents/data/en/reference-manuals/MPC82XINSET.pdf.

124. Леошкевич, И.О. Получение архитектурно-независимой семантики исполняемого кода / И.О. Леошкевич // Безопасность информационных технологий. – 2009. – № 4. – С. 120-124.

125. Падарян, В.А. Моделирование операционной семантики машинных инструкций / А.И. Кононов, В.А. Падарян, М.А. Соловьев // Программирование. – 2011. – Т. 37. – № 3. – С. 50-64.

126. Balakrishnan, J. Intermediate-representation recovery from low-level code / G. Balakrishnan, J. Lim, T. Reps // Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics based Program Manipulation (PEMP '06). – 2006. – PP. 100-111.

127. Формат DWARF [Электронный ресурс]. – Режим доступа: http://dwarfstd.org.

128. Шудрак, М.О. Методика декомпиляции бинарного кода и ее применение в сфере информационной безопасности / В.В. Золотарев, И.А. Лубкин, М.О. Шудрак // Безопасность информационных технологий. – 2012. – № 3. – С. 75-80.

129. Плагин Hex-Rays [Электронный ресурс]. – Режим доступа: https://www.hex-rays.com.

130. Программа IDA Pro [Электронный ресурс]. – Режим доступа: https://www.hex-rays.com/products/ida.

131. Программа Boomerang [Электронный ресурс]. – Режим доступа: http://boomerang.sourceforge.net.

132. Программа ExeToC Decompiler [Электронный ресурс]. – Режим доступа: https://sourceforge.net/projects/exetoc.

133. Программа REC Studio 4 [Электронный ресурс]. – Режим доступа: http://www.backerstreet.com/rec/rec.htm.

134. Программа Reko [Электронный ресурс]. – Режим доступа: https://github.com/uxmal/reko.

135. Программа RelipmoC [Электронный ресурс]. – Режим доступа: https://sourceforge.net/projects/relipmoc.

136. Программа Retargetable Decompile [Электронный ресурс]. – Режим доступа: https://retdec.com.

137. Программа SmartDec [Электронный ресурс]. – Режим доступа: http://decompilation.info.

138. Программа Snowman [Электронный ресурс]. – Режим доступа: https://derevenets.com.

139. Ахо, А. Компиляторы. Принципы, технологии, инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М.: Вильямс, 2008. – 1185 с.

140. Программа Immunity Debugger [Электронный ресурс]. – Режим доступа: https://www.immunityinc.com/products/debugger.

141. Библиотека libdisasm [Электронный ресурс]. – Режим доступа: http://bastard.sourceforge.net/libdisasm.html.

142. Библиотека Udis86 [Электронный ресурс]. – Режим доступа: http://udis86.sourceforge.net.

143. Падарян, В.А. Методы поиска ошибок в бинарном коде: техн. отчет (2013-1) / В.В. Каушан, Ю.В. Маркин, В.А. Падарян, А.Ю. Тихонов. – М.: Институт системного программирования РАН, 2013.– 79 с.

144. Козачок, А.В. Разработка эвристического механизма обнаружения вредоносных программ на основе скрытых марковских моделей / А.В. Козачок // Проблемы информационной безопасности. Компьютерные системы. – 2016. – Т. 2. – С. 126-133.

145. Amini, P. Fuzzing: Brute Force Vulnerability Discovery / P. Amini, A. Greene, M. Sutton. – USA: Addison-Wesley Professional, 2007. – 576 p.

146. Zhang, S. Combined static and dynamic automated test generation / Y. Bu, M.D. Ernst, D. Saff, S. Zhang // Proceedings of the 2011 ACM International Symposium on Software Testing and Analysis. – 2011. – PP. 353-363.

147. Aiken, A. Static error detection using semantic inconsistency inference / A. Aiken, I. Dillig, T. Dillig // Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07). – 2007. – PP. 435-445.

148. Бородин, А.Е. Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ / А.Е. Бородин, Р.Р. Мулюков // Труды Института системного программирования РАН. – 2016. – Т. 28. – № 5. – С. 145-158.

149. Дудина, И.А. Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа / И.А. Дудина // Труды Института системного программирования РАН. – 2015. – Т. 28. – № 5. – С. 119-134.

150. Вартанов, С.П. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных / С.П. Вартанов, А.Ю. Герасимов // Труды Института системного программирования РАН. – 2014. – Т. 26. – № 1. – С. 375-394.

151. Chou, A. Archer: using symbolic, path-sensitive analysis to detect memory access errors / A. Chou, D.R. Engler, Y. Xie // ESEC / SIGSOFT FSE. – 2003. – PP. 327-336.

152. Программа Flex [Электронный ресурс]. – Режим доступа: https://github.com/westes/flex.

153. Программа Bison [Электронный ресурс]. – Режим доступа: http://www.gnu.org/software/bison.

154. Стандарт DIN 66261 [Электронный ресурс]. – Режим доступа: http://www.fzt.haw-hamburg.de/pers/Abulawi/Struktogrammsinnbilder.pdf.

155. Белеванцев, А.А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ / А.А. Белеванцев, Е.А. Велесевич // Труды Института системного программирования РАН. – 2015. – Т. 27. – № 2. – С. 53-64.

156. Гавриленко, С.Ю. Использование языка XML для промежуточного представления программы / С.Ю. Гавриленко // Вестник Национального технического университета Харьковский политехнический институт. Серия: Информатика и моделирование. – 2008. – № 24. – С. 19-24.

157. Карпулевич, Е.А. Использование различных представлений java-программ для статического анализа / Е.А. Карпулевич // Труды Института системного программирования РАН. – 2015. – Т. 27. – № 6. – С. 151-158.

158. Koschke .R An intermediate representation for integrating reverse engineering analyses / J.-F. Girard, R. Koschke, M. Wurthner // 5th Working Conference on Reverse Engineering. – 1998. – PP. 241–250.

159. Stanier, J. Intermediate representations in imperative compilers: A survey / J. Stanier, D. Watson // ACM Computing Surveys (CSUR). – 2013. – Vol. 45. – Iss. 3. – Article No. 26.

160. Foster, J. Understanding source code evolution using abstract syntax tree matching / J. Foster, M. Hicks, I. Neamtiu // Proceedings of the 2005 International workshop on Mining software repositories (MSR '05). – 2005. – PP. 1-5.

161. Формат JSON [Электронный ресурс]. – Режим доступа: http://www.json.org.

162. Формат DOT [Электронный ресурс]. – Режим доступа: http://www.graphviz.org/Documentation.php.

163. Программа GraphViz [Электронный ресурс]. – Режим доступа: http://www.graphviz.org

164. Саати Т.Л. Принятие решений. Метод анализа иерархий / Т.Л. Саати. – М.: Радио и связь, 1989. – 316 с.

165. Программа AsmEditor [Электронный ресурс]. – Режим доступа: http://asmeditor.sourceforge.net.

166. Программа Visustin [Электронный ресурс]. – Режим доступа: http://www.aivosto.com/visustin.html.

167. Ледовских, И.Н. Метрики сложности кода: техн. отчет (2012-2) / И.Н. Ледовских. – М.: Институт системного программирования РАН, 2012. – 22 с.

168. Abran, A. Software Metrics and Software Metrology / A. Abran. – Hoboken, NJ: Wiley-IEEE Computer Society Press, 2010. – 348 p.

169. Henry, S. Software structure metrics based on information flow / S. Henry, D. Kafura // IEEE Transactions on Software Engineering. – 1981. – Vol. 5. – PP. 510-518.

170. Oviedo, E.I. Control Flow, Data Flow and Program Complexity / E.I. Oviedo // Proceedings of COMPSAC'80. – 1980. – PP. 146-152.

171. Chapin, N. An entropy metric for software maintainability / N. Chapin // Twenty-Second Annual Hawaii International Conference on System Sciences. – 1989. – Vol. II: Software Track. – PP. 522–523.

172. Schneidewind, N.F. Methodology for validating software metrics / N.F. Schneidewind // IEEE Transactions on Software Engineering. – 1992. – Vol. 18. – Iss. 5. – PP. 410-422.

173. Звездин, С.В. Проблемы измерения качества программного кода / Звездин С.В. // Вестник Южно-Уральского государственного университета. Серия: Компьютерные технологии, управление, радиоэлектроника. – 2010. – № 2(178). – С. 62-66.

174. Мельникова, Е.В. Научное осмысление природы информации / О.А. Мельников, Е.В. Мельникова // Научно-техническая информация: Серия 1. – 2011. – № 6. – С. 1-7.

175. Серова, Т.С. Осмысление понимания и фиксация предметного содержания текста как программы письменного перевода / Т.С. Серова // Вестник Нижегородского государственного лингвистического университета им. Н.А. Добролюбова. – 2009. – № 5. – С. 105-108.

ПРИЛОЖЕНИЕ А. Исходный код IDC-скрипта генерации АК

Приводится исходный код IDC-скрипта, исполняемый в продукте IDA Pro, генерирующий ассемблерный код текущей подпрограммы с синтаксисом, поддерживаемым Утилитой.

```
// Generate asm for current function

#include <idc.idc>

#define DEF_FILE_NAME "funct.asm"

static processFunction(ea, file){
  auto functName, functStart, functEnd, comment, index;

  functName = GetFunctionName(ea);
  functStart = GetFunctionAttr(ea, FUNCATTR_START);
  functEnd = GetFunctionAttr(ea, FUNCATTR_END);

  comment = GetFunctionCmt(ea, 0);
  if(comment != "")
    fprintf(file, "// %s\n", comment);
  fprintf(file, "// Function '%s' {0x%08lX-0x%08lX}\n", functName, functStart,
functEnd);
  fprintf(file, "%s(){\n", functName);

  for(ea = functStart; (ea < functEnd) && (ea != BADADDR); ea = NextHead(ea, BA-
DADDR)){
    auto label, instr, lineA, refC;

    // Prior line (only one line)
    lineA = LineA(ea, 0);
    if(lineA != 0)
      fprintf(file, "// %s", lineA);

    // Label
    label = NameEx(BADADDR, ea);
    if(label != 0)
      fprintf(file, "\n0x%.8X: %s:\n", ea, label);

    // Instructions
    instr = GetDisasm(ea);
    if(instr == "")
      continue;

    if((index = strstr(instr, "#")) != -1){
      instr = substr(instr, 0, index) + "//" + substr(instr, index + 1, -1);
    }

    fprintf(file, "0x%.8X: %s\n", ea, instr);
  }

  fprintf(file, "}\n");
}
```

```
static main() {
  auto ea;
  auto file, fileName;
  auto idArray, index, indexLast;

  fileName = AskFile(1, DEF_FILE_NAME, "Enter file name for generate function's as-
sembler");
  if(fileName == "")
    return -1;

  file = fopen(fileName, "w");
  if(file == 0){
    Message("Can't open file '%s'\n", fileName);
    return -1;
  }

  processFunction(ScreenEA(), file);

  fclose(file);
  return 0;
}
```

ПРИЛОЖЕНИЕ Б. Модульная структура функциональной архитектуры Утилиты

Приведены Стадии работы Утилиты, реализующие их модули, назначение, выполняемые функции и гипотетические примеры входных и выходных данных последних, а также специальные требования к реализации приведены в Приложении Б.

Стадия 1

Стадия предназначена для разбора входного АК и построения его внутреннего представления в фазе Front-End. Стадия представляет собой классический пример фазы компилятора и состоит из 3-х модулей: лексического, синтаксического и семантического анализаторов. Первый предназначен для разбиения входного потока символов АК на отдельные *лексемы* (подобно сборке букв в слова), осуществляя тем самым базовую формализацию. Второй собирает отдельные лексемы, сопоставляя их с заранее заданными правилами (подобно словесным предложениям). Для такого разбора используется формальная грамматика, заданная синтаксисом входного языка с помощью рекурсивных правил; последние дополняются пользовательскими действиями (на используемом языке программирования), выполняемыми при *сборке* правил – т. е. при соответствии правилу входных лексем. В результате строится дерево абстрактного синтаксиса, отражающее входной код в полностью формализованном и структурированном виде. Для разбора смысловых значений правил предназначен третий модуль стадии – семантический анализатор. И хотя в классических компиляторах ему отводится значимое место, в текущей архитектуре модуль может считаться условным, поскольку он выполняет лишь вспомогательные действия, такие, как добавление в дерево абстрактного синтаксиса узлов, хранящих результаты операций сравнения для соответствующих инструкций процессора. В результате работы модулей будет создано дерево абстрактного синтаксиса входного АК, подходящее для обработки на дальнейших стадиях.

Необходимо отметить, что, хотя стадия и является зависимой от процессора выполнения (поскольку в ней использован синтаксис входного АК), однако реализацию анализаторов возможно сделать таким образом, чтобы генерируемое ими абстрактное дерево не использовало инструкции процессора, а оперировало абстрактными операциями и переменными. Например, следующий АК для процессора PowerPC:

```
LI R0, 0x1          ; R0 = 1
ADDI R1, R0, 0x2    ; R1 = R0 + 0x2
```

можно уже в процессе синтаксического анализа преобразовать в следующее абстрактное дерево (рисунок Б.1), аналогичное процессорно-независимому коду:

```
X = 0x1;
Y = X + 0x2
```

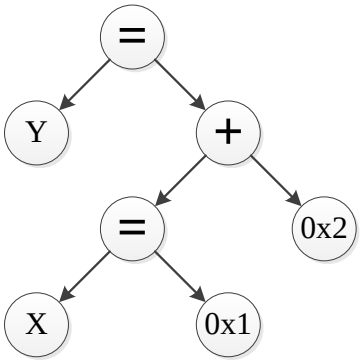

Рисунок Б.1 – Пример абстрактного дерева простейшего выражения

Такое обеспечение процессорной инвариантности последующего уровня от Front-End позволяет разрабатывать обобщенные (шаблонные) алгоритмы, что повышает, как переносимость кода, так и его надежность.

Стадия 2

Стадия является первой в фазе Middle-End и предназначена для начальной обработки дерева абстрактного синтаксиса и разделения его на различные слои, содержащие информацию о подпрограммах, потоке управления, глобальных и локальных переменных и др. Такое преобразование является побочным результатом работы следующих модулей стадии. Во-первых, Модуль выделение подпрограмм анализирует дерево абстрактного синтаксиса, идентифицируя подпрограммы и занося информацию о них в SCT. Во-вторых, используя результаты идентификации, Модуль построения графа потока управления создает соответствующий

граф, определяющий все переходы внутри тела подпрограммы (т. е. УС алгоритма). Граф потока управления целесообразно преобразовать к так называемому «жесткому» (путем добавления служебных узлов, строго задающих закономерности в графе), имеющему более формальную структуру, чем первый – это необходимо для упрощения реализации алгоритмов его обработки. В нем, например, в начало веток условного ветвления добавлены служебные узлы, чтобы ветка всегда имела, по крайне мере, один узел – в «нежестком» графе пустая ветка не будет иметь узлов, что усложнит алгоритмы ее обработки необходимыми проверками наличия узлов и т. п. В процессе построения потока управления анализируются вызовы внешних подпрограмм, что позволяет параллельно строить и граф их вызовов. И в-третьих, анализ дерева абстрактного синтаксиса Модулем выделения глобальных переменных идентифицирует и заносит информацию о последних также в SCT. Данное дерево является упрощенным аналогом дерева областей видимости, используемого в компиляторах. Таким образом, SCT хранит информацию, как обо всех переменных, так и о подпрограммах, включая косвенно графы потока управления последних (через ссылки). Обработка дерева абстрактного синтаксиса, аналогичного следующему примеру С-подобного псевдокода:

```
var GLOBAL;
FUNCT(R1) {
  R0 = R1 + GLOBAL ;
  return R0;
}
```

построит графа потока управления и занесет данные в SCT, как показано на рисунке Б.2.

Стадия 3

Стадия предназначена для выделения СМД во внутреннем представлении S-модели, полученном на предыдущих стадиях. Для этого, в частности, Модуль построения графа потока данных создает привязанную к графу потока управления информацию о временах жизни переменных, их значениях, первых/последних точках использования и т. п. Данная информация может быть использована как на данной стадии, так и на последующих.

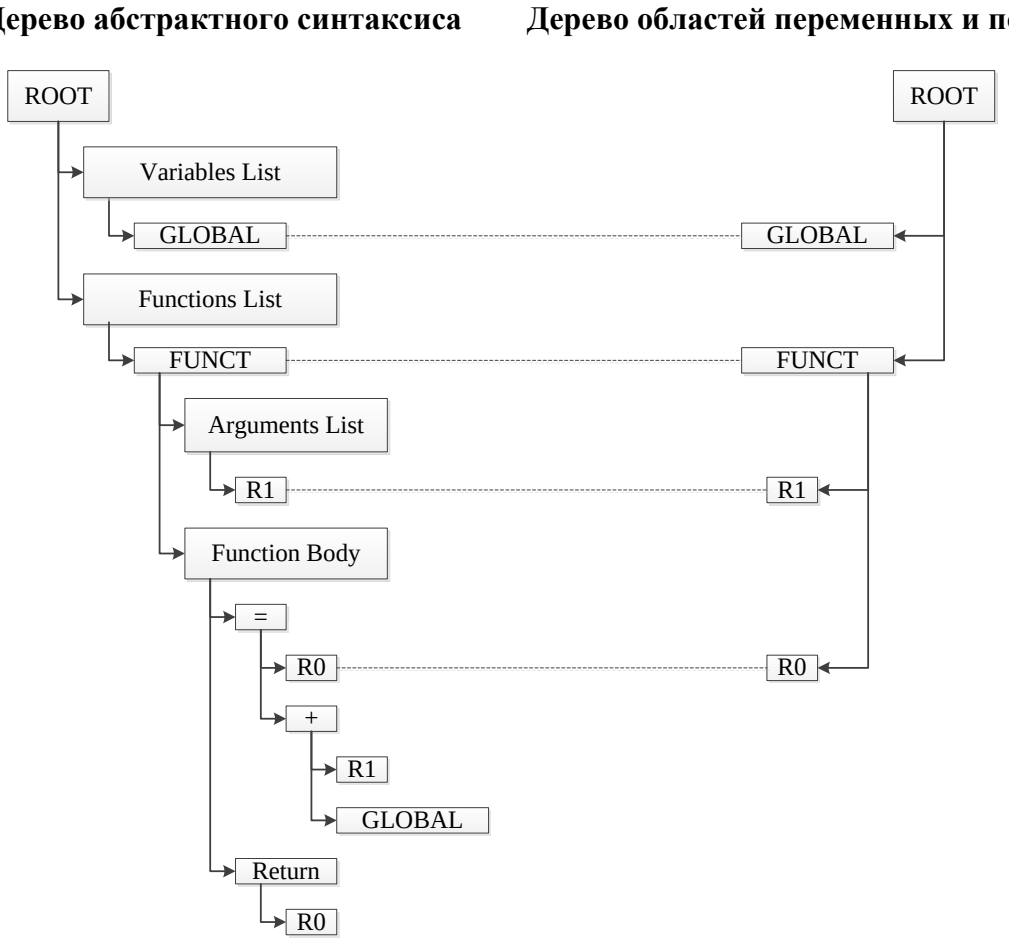

Рисунок Б.2 – Пример взаимосвязи дерева абстрактного синтаксиса и дерева областей переменных и подпрограмм

Модуль уточнения сигнатуры подпрограмм анализирует граф потока управления и, используя граф потока данных, предсказывает такие свойства сигнатуры, как входные и выходные параметры. В рамках АК под параметрами подпрограммы подразумеваются наборы регистров, с помощью которых подпрограмма получает внешние значения и выдает результаты своих вычислений. Например, для следующего примера ассемблерной подпрограммы на С-подобном псевдо-коде:

```
??? FUNCT(???) {
  R0 = R1 + R2 ;
  return R0;
}
```

очевидно, что, скорее всего подпрограмма FUNCT() принимает на входе параметры посредством двух регистров – R1 и R2, поскольку они используются без какой-либо явной инициализации, и возвращает на выходе результат вычислений

через регистр R0 поскольку ему присваивается значение без какого-либо последующего использования; вариант же генерации неоптимального кода современным компилятором практически исключается. Так, используя данную логику, Модуль уточнения сигнатуры для вышеприведенного примера определит сигнатуру подпрограммы следующим образом:

```
( R0 ) FUNCT( R1, R2 );
```

Необходимо отметить, что хотя для завершенных сигнатур подпрограммы (с точки зрения, например, языка С) требуются также типы аргументов и возвращаемого значения, тем не менее, для описания логики работы алгоритмов они не существенны и данным модулем не восстанавливаются. Тем не менее, такое восстановление возможно.

Основным модулем стадии (и, в некотором смысле, всей Утилиты), является Модуль выделения СМД, параллельно переводящий графовидный поток управления подпрограммы в древовидную форму, подобную диаграммам Насси-Шнейдермана. Основные функции модуля состоят из выделения всех условных переходов (включая управляющие конструкции, ветки и завершающие метки) и циклов на графе потока управления (включая условия выхода из цикла и переходы по итерациям). Анализ модуль осуществляет с применением рекурсий и раскраски графов; затем делается их перестроение на использование конструкций Насси-Шнейдермана. Примеры такого перестроения для простейшего условного перехода и цикла показаны на рисунке Б.3.

Ситуация, когда граф потока управления не может быть сведен к древовидной структуре (например, при наличии оператора безусловного перехода GOTO) считается вырожденной и описывается в дереве потока управления введением дополнительной связи (рисунок Б.4).

Как хорошо видно на примерах (рисунки Б.3 и Б.4), переход к древовидному внутреннему представлению потока управления увеличивает структурируемость формы кода, что должно положительно сказаться на его восприятии человеком.

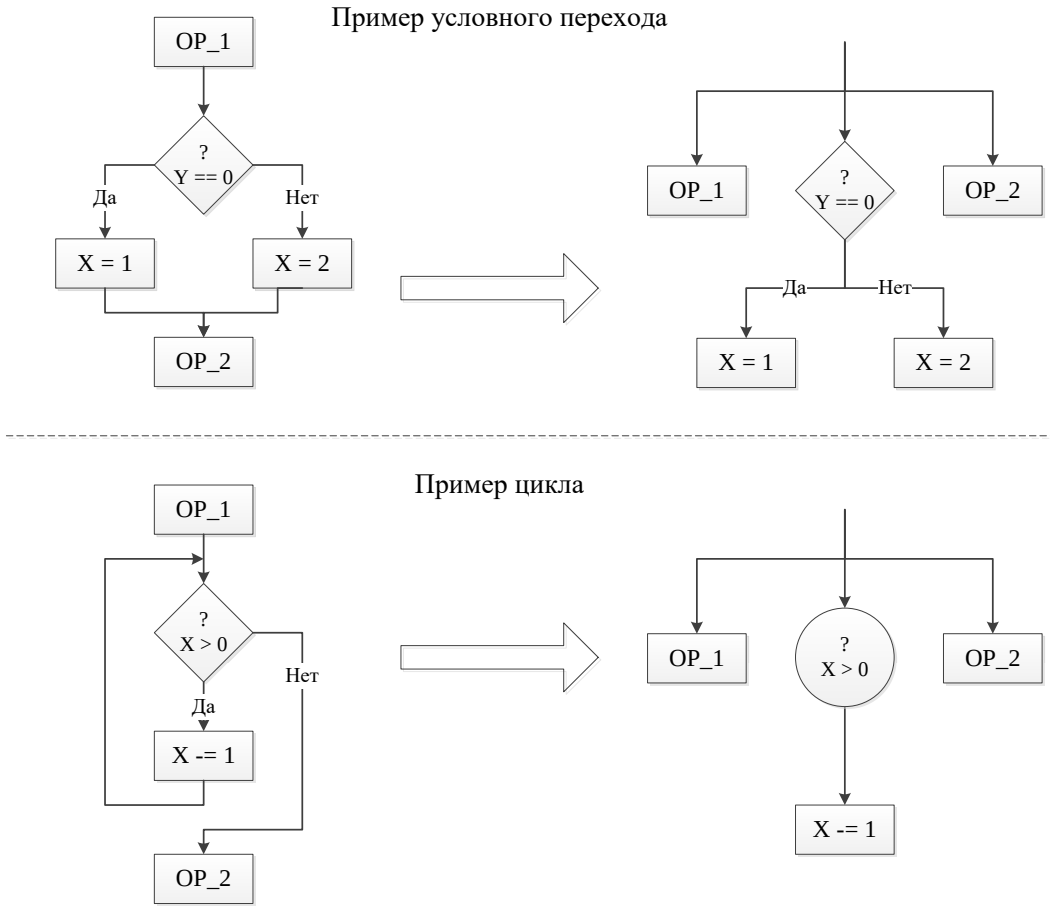

Рисунок Б.3 – Примеры построения диаграмм Насси-Шнейдермана для условного перехода и цикла

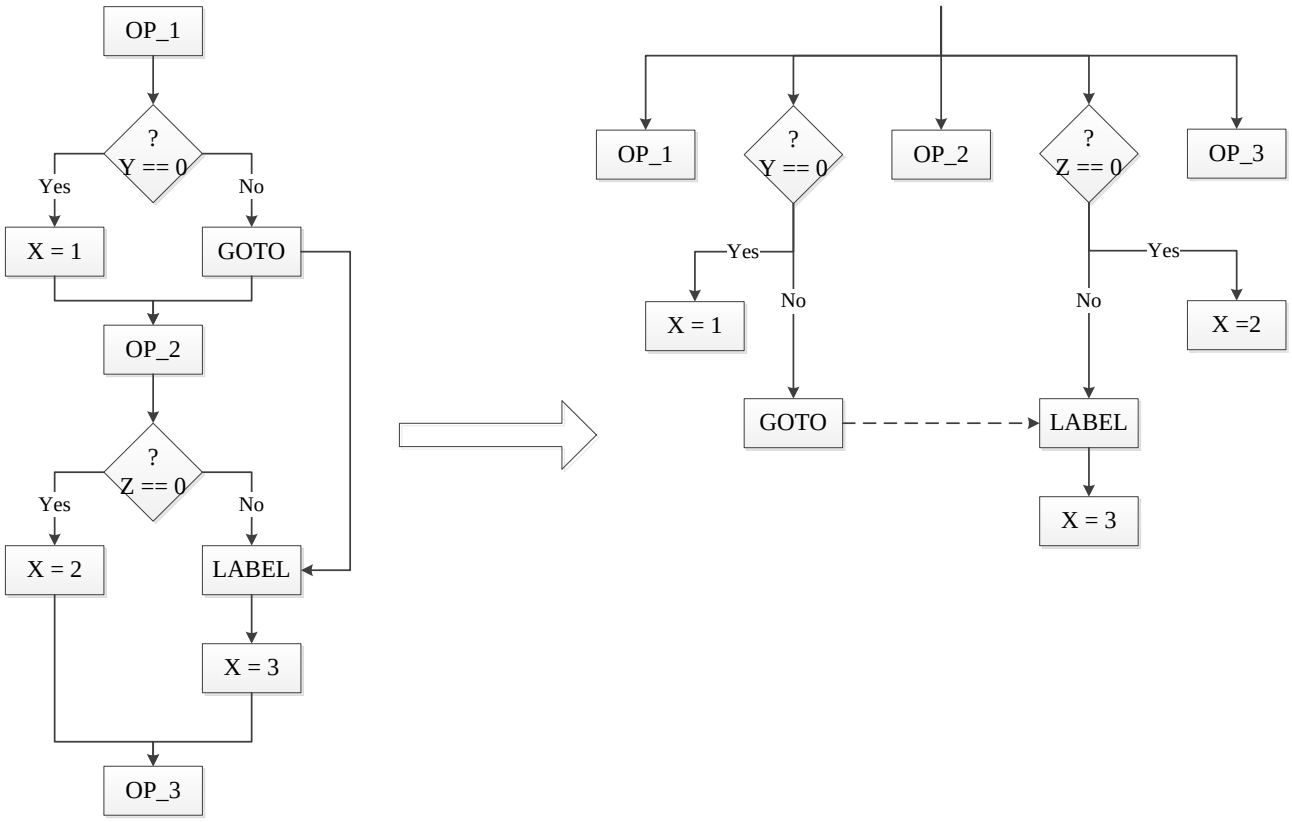

Рисунок Б.4 – Пример с безусловным переходом, не сводимый к древовидной структуре

Стадия 4

Стадия предназначена для оптимизации представления кода в интересах сокращения его размера и упрощения УС; она состоит из 3-х модулей:

1) Модуля оптимизации СМД, выполняющего следующие действия:

– удаление неиспользуемых меток, а также безусловного перехода на следующую операцию после ветвления, на начало цикла и на первую инструкцию после цикла;

– объединение меток, а также нескольких веток условных переходов и выхода из цикла;

– пересортировка веток условных переходов;

– вынесение выхода из подпрограммы вне цикла.

2) Модуля оптимизации дерева потока управления, выполняющего замену безусловного выхода из цикла на конструкцию BREAK и безусловного начала следующей итерации цикла на конструкцию NEXT;

3) Модуля оптимизации вычислений, выполняющего следующие действия:

– вычисление значений выражений, включая промежуточные;

– замена вычисленных значений выражений на соответствующие константы, а также битового доступа к переменным на специальные конструкции вида VAR.N_BIT;

– упрощение выражений путем подстановки инициализационных значений переменных;

– упрощение булевских операций, операций сравнения, а также операций сложения/вычитания отрицательных чисел;

– удаление операций, не имеющих эффекта, а также операций, эффект которых не изменяет состояние программы.

Пример оптимизации дерева потока управления показан на рисунке Б.5.

В процессе работы стадии также строится граф зависимости вычислений, который отражает выражения в коде и переменные, от которых зависят вычисления. Граф используется только в рамках стадии Модулями оптимизации и обновляется после большинства крупных операций по перестроению дерева потока управления.

Пример оптимизации дерева потока управления

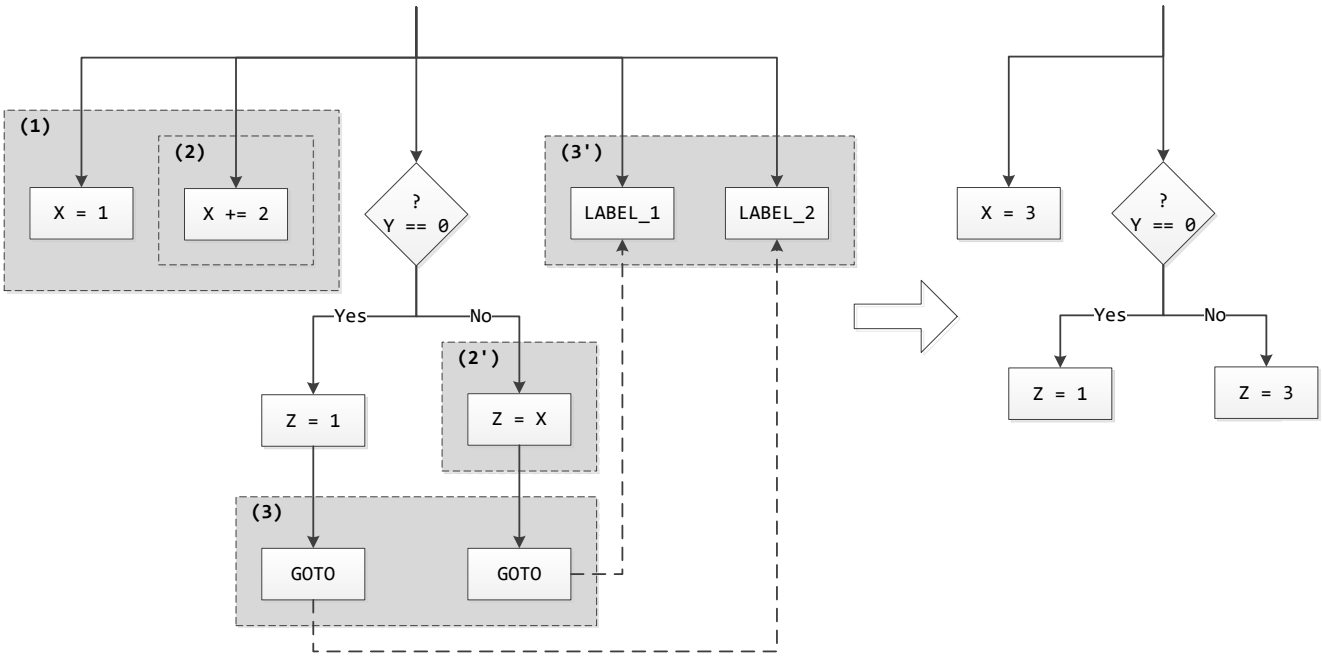

Рисунок Б.5 – Пример оптимизации дерева потока управления

Примечание. На рисунке Б.5 применены следующие оптимизации: (1) – вычисление значений выражений, (2)+(2') – подстановка значений выражений, (3)+(3') – удаление переходов на операцию после ветвления.

Все модули выполняются в несколько проходов, пока ни одна из их оптимизаций не перестанет иметь эффект – т. е. производить какие-либо изменения во внутренних представлениях Утилиты. Зацикливание стадии предотвращает то, что все оптимизации модулей реализованы таким образом, чтобы всегда упрощать или не изменять Представление кода.

Стадия 5

Стадия предназначена для генерации псевдокода алгоритмов в виде соответствующего дерева (включая как подпрограммы, так и глобальные переменные) с последующим приведением его в лаконичный вид, и реализуется следующими модулями.

Модуль генерации псевдокода глобальных переменных производит их добавление в РСТ с указанием адресов размещения (при наличии). Модуль генерации псевдокода подпрограмм практически без изменений переносит внутреннее

представление каждой подпрограммы в РСТ. Затем на данном дереве Модуль лаконизации псевдокода производит его видоизменение в интересах повышения восприятия кода человеком, что обеспечивают следующие действия:

– сортировка и комбинирование коммутативных операций;

– замена абсолютных адресов на соответствующие глобальные переменные, а также вычислительных операций на специальные конструкции – инкрементирование/декрементирование (++, --), операция с присваиванием ([+-*/]=), доступ к элементу массива.

Пример лаконизации РСТ показан на рисунке Б.6.

Рисунок Б.6 – Пример лаконизации дерева псевдокода

Примечание. На рисунке Б.6 применены следующие оптимизации: (1) – замена на специальную конструкцию: инкрементирование, (2)+(2') – подстановка глобальной переменной вместо адреса, (3) – замена на специальную конструкцию: доступ к элементу массива.

Модуль выполняется в несколько проходов, пока ни одно из его действий по лаконизации не перестанет производить изменения.

Модуль генерации псевдокода архитектуры добавляет в РСТ информацию об архитектуре ПрК с помощью деления последнего на модули. Пример архитектуры в результате такого деления показан на рисунке Б.7.

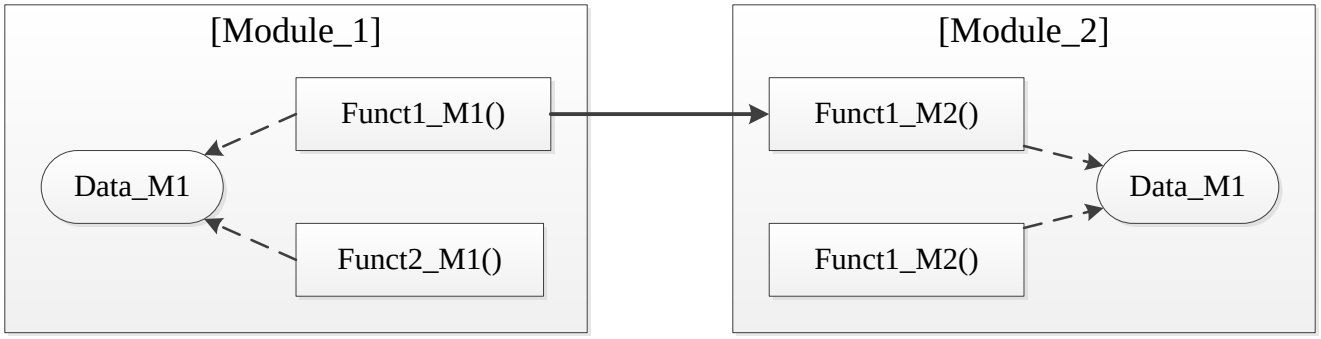

Рисунок Б.7 – Пример архитектуры программного кода

Согласно рисунку, архитектура состоит из двух модулей (Module_1 и Module_2), содержащих по две функции (Funct1_M1() и Funct1_M2() в первом модуле; Funct1_M2() и Funct2_M2() во втором модуле), использующие данные модуля (Data_M1 и Data_M2; их использование показано пунктирными линиями). При этом, функция одного модуля вызывает функцию другого (Funct1_M1() вызывает Funct1_M2(); показано непрерывной линией).

Таким образом, стадия практически не сокращает объем кода, а только делает его более воспринимаемым за счет использования специальных конструкций и выбора других форм представления элементов кода. Стадия не может зацикливаться по причинам, аналогичным Стадии 4; она является заключительной в фазе Middle-End.

Стадия 6

Стадия предназначена для генерации выходного описания алгоритмов кода в фазе Back-End на основании построенных, оптимизированных и лаконизированных внутренних представлений данных. Стадия представляет собой классический пример фазы компилятора и состоит из следующих модулей.

Модуль генерации текстового описания корректировок алгоритмизации добавляет в выходной код корректировки алгоритмизации, внесенные во входной ассемблерный. Это необходимо для того, чтобы полученное алгоритмизированного представления (включая информацию в нем об уязвимостях) считалось обоснованным, поскольку последнее зависит от вносимых корректировок. Пример

корректировок в выходном коде Утилиты, указывающих на переменную с режимом доступа «readonly» (только для чтения) полностью совпадает со входными:

```
user_control {
  secret_for_read: p_readonly;
}
```

Модуль генерации текстового описания архитектуры разбивает глобальные переменные и функции по модулям, указывая их интерфейсы. Так, в случае архитектуры из примера на рисунке Б.7, вывод Утилиты будет иметь следующий вид:

```
module Module_1 {
  var Data_M1;
  () Funct1_M1() {
    ...
    Funct1_M2();
  }
  () Funct2_M1() {
    ...
  }
}
module Module_2 {
  /* Funct1_M2() - Interface (nCall = 1) */;
  var Data_M2;
  () Funct1_M2() {
    ...
  }
  () Funct2_M2() {
    ...
  }
}
```

Также, согласно алгоритмизированному представлению, Утилита определила функцию Funct1_M2(), как интерфейсную для модуля Module_2, поскольку она вызывается из функции Module_1.

Модуль генерации текстового описания глобальных переменных создает описание всех глобальных переменных, восстановленных по АК, в текстовом виде, при возможности указывая их адреса. Пример такого вывода для глобальных переменных global_1 и global_2, последняя из которых размещенная по адресу 0x1234, будет следующим:

```
var global_1;
var global_2 & 0x1234;
```

Модуль генерации текстового описания подпрограмм создает описания всех подпрограмм (включая их сигнатуры и алгоритмы) в текстовом виде. Пример та-

кого вывода для подпрограммы FUNCT(), инкрементирующей и возвращающей значение параметра (через регистр R0), будет следующим:

```
(r0) FUNCT(r0) {
  ++ r0;
  return r0;
}
```

Модуль генерации информации об уязвимостях добавляет пометки о возможных уязвимых местах, используя данные, собранные Модулем поиска уязвимостей. Пример такого вывода для подпрограммы Destructed_Funct(), структура алгоритма которой была разрушена, будет следующим:

```
Destructed_Funct() {
  /* ATTENTION!!! Possible, destruction of the structure. */;
}
```

Модули фазы Middle-End

На всех стадиях фазы Middle-End (Стадии 2-5) функционируют два модуля: учет корректировок алгоритмизации и поиск уязвимостей. Первый, используя информацию о корректировках алгоритмизации, полученную при разборе входного ассемблера, управляет работой всех модулей в этой фазе – например, явно задавая для указанной ассемблерной подпрограммы регистры, используемые в качестве ее аргументов. Таким образом, модуль влияет как на топологию и отдельные характеристики внутренних представлений Утилиты, так и на конечный текстовый вид алгоритмов. Второй же предоставляет различные шаблоны и алгоритмы для выделения информации о потенциальных уязвимостях, которые, в конечном итоге, используются Модулем генерации информации об уязвимостях для их вывода. С точки зрения линейности выполнения стадий, данные модули можно назвать *ортогональными*, поскольку они, так или иначе, могут взаимодействовать со всеми другими.

ПРИЛОЖЕНИЕ В. Внутренние представления Утилиты

Описаны внутренние представления, используемые Утилитой, в виде их назначения, формы и содержание. Приведены конкретные «снимки» внутренних представлений для сквозного примера.

В примерах использован способ описания деревьев в виде текстовых строк (генерируемых в отладочном выводе Утилиты), каждая из которых содержит описание одной вершины дерева, пробельные отступы означают глубину вложенности элемента, а более ранний в тексте элемент с меньшим отступом – родительский узел дерева.

Примеры графов имеют графический вид, созданный с применением программы из пакета GraphViz, визуализирующей графы в формате DOT (также генерируемые Утилитой в отладочном выводе).

Дерево абстрактного синтаксиса (AST)

Для базовой формализации входного АК классически в компиляторах и подобных ПС применяется AST. Внутренние вершины дерева, как правило, сопоставляются с операторами языка, а листья – с операндами и константами. Классическое дерево абстрактного синтаксиса (что также вытекает и из его названия) отражает специфику входного синтаксиса и впоследствии преобразуется к дереву внутреннего представления (от англ. Intermediate Representation или IR), считающемуся полностью независимым от входного языка (как, кстати, и от выходного). Тем не менее, по причине простоты синтаксиса ассемблера и его инструкций, возможно и целесообразно строить дерево абстрактного синтаксиса сразу же независимым от ассемблерных операций и регистров, т. е. переводя их в разряд *обезличенных* операций над переменными; что и было реализовано в Утилите. Дерево абстрактного синтаксиса для текущего примера следующее:

```
IrRoot()
  IrList()   // GlobalDecl
  IrList()   // Functs
    IrName('max2')
      IrList()     // Args
        IrEmpty()
      IrList()     // Rets
```

```
  IrEmpty()
IrEmpty()
IrReg('lr'), id=41, exprInfo=(NULL)
IrList()     // Code
  IrLabel('max2'), name='max2'
  IrOperation('cmpw'), kind='assign'
    IrOperation('cmpw'), kind='bit access'
      IrReg('cr'), id=32, exprInfo=(NULL)
      IrInteger(''), value=0, kind=dec, fSigned=true
    IrCond('cmpw'), kind='?true'
      IrCond('cmpw'), kind='<'
        IrReg('r3'), id=3, exprInfo=(NULL)
        IrReg('r4'), id=4, exprInfo=(NULL)
  IrOperation('cmpw'), kind='assign'
    IrOperation('cmpw'), kind='bit access'
      IrReg('cr'), id=32, exprInfo=(NULL)
      IrInteger(''), value=1, kind=dec, fSigned=true
    IrCond('cmpw'), kind='?true'
      IrCond('cmpw'), kind='>'
        IrReg('r3'), id=3, exprInfo=(NULL)
        IrReg('r4'), id=4, exprInfo=(NULL)
  IrOperation('cmpw'), kind='assign'
    IrOperation('cmpw'), kind='bit access'
      IrReg('cr'), id=32, exprInfo=(NULL)
      IrInteger(''), value=2, kind=dec, fSigned=true
    IrCond('cmpw'), kind='?true'
      IrCond('cmpw'), kind='=='
        IrReg('r3'), id=3, exprInfo=(NULL)
        IrReg('r4'), id=4, exprInfo=(NULL)
  IrBranch('ble')
    IrCond('ble'), kind='?false'
      IrOperation('ble'), kind='bit access'
        IrReg('cr'), id=32, exprInfo=(NULL)
        IrInteger(''), value=1, kind=dec, fSigned=true
    IrLabel('label_1'), name='label_1'
  IrOperation('mr'), kind='assign'
    IrReg('r5'), id=5, exprInfo=(NULL)
    IrReg('r3'), id=3, exprInfo=(NULL)
  IrBranch('b')
    IrEmpty()
    IrLabel('label_2'), name='label_2'
  IrLabel('label_1'), name='label_1'
  IrOperation('mr'), kind='assign'
    IrReg('r5'), id=5, exprInfo=(NULL)
    IrReg('r4'), id=4, exprInfo=(NULL)
  IrLabel('label_2'), name='label_2'
  IrOperation('mr'), kind='assign'
    IrReg('r3'), id=3, exprInfo=(NULL)
    IrReg('r5'), id=5, exprInfo=(NULL)
  IrBranch('blr')
    IrEmpty()
    IrReg('lr'), id=41, exprInfo=(NULL)
```

При детальном рассмотрении дерева хорошо видно, что оно практически полностью повторяет весь АК, но сделав его структуризацию (выделив подпрограмму, операции и операнды) и используя независимые от процессора термины (IrReg могут трактоваться как некие переменные; все IrOperation имеют свойство

kind, задающее обобщенный смысл операции; IrBranch задают любые переходы, как условные, так безусловные и даже выходы из подпрограммы, и т. п.). При этом в дереве отсутствует так называемый «лексический и синтаксический мусор», такой, как пробелы, запятые, неиспользуемые адреса, инструкции и т. п.

Граф потока управления (CFG)

Для воспроизводства топологии алгоритма строится ориентированный CFG. Узлы графа соответствуют базовым блокам – прямолинейным участкам кода без операций передачи и точек получения управления, с тем исключением, что каждый базовый блок начинается с метки для получения управления и заканчивается инструкцией передачи управления. Направленные дуги в графе задают переходы между блоками согласно инструкциям. Каждый такой граф для подпрограммы начинается с входного блока и заканчивается выходным. Таким образом, граф описывает все множество путей выполнения программы. Граф потока управления для текущего примера следующий (рисунок В.1).

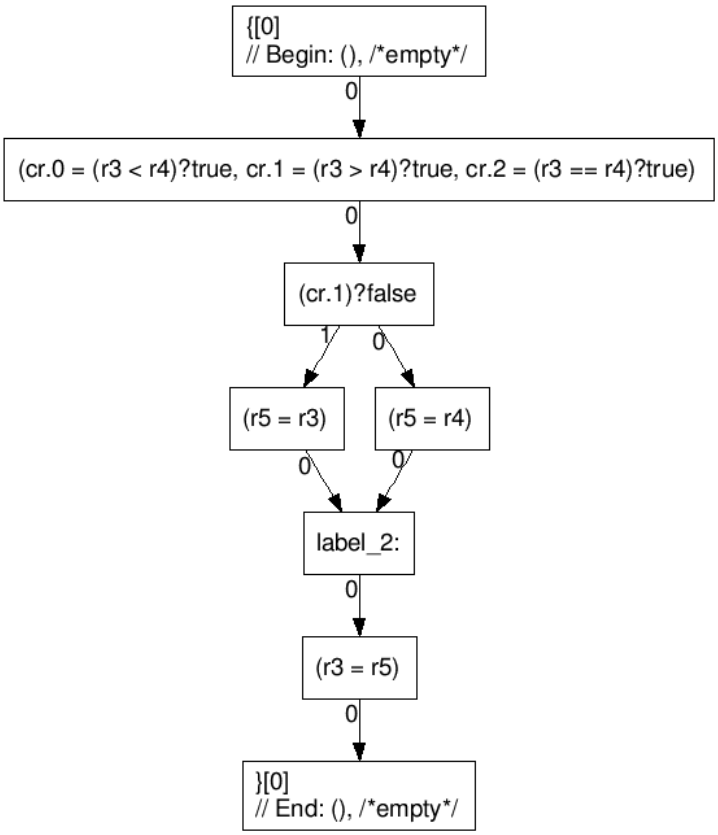

Рисунок В.1 – Пример графа потока управления для функции max2()

Как хорошо видно из рисунка, граф имеет входной блок (с комментарием «// Begin») – соответствующий началу подпрограммы, и выходной блок (с комментарием «// End») – соответствующий завершению подпрограммы, т. е. выполнению инструкции BLR для PowerPC (или операции «return» в языке C). Также, на графе видны два пути выполнения программы в зависимости от значения 1-го бита регистра CR («(cr.1)?false»), определяемого операциями в предыдущем блоке графа по результату сравнения «R3 > R4».

Граф потока управления («жесткий»)

«Жесткий» граф потока управления не имеет применения в практике разработки компиляторов, хотя в последних и используются иные гибридные решения. Основное назначения графа в стандартизации его топологии для упрощения разработки будущих алгоритмов его обработки; для этого в «нежесткий» граф добавляются служебные узлы, являющиеся первыми в каждой из веток условного выполнения, даже если последние не содержат инструкций. Это приводит к тому, что у каждого УС для условного выполнения всегда есть два нижележащих служебных узла, которые не могут быть удалены в дальнейшем оптимизациями, тем самым разрушая структуру. Это позволяет реализовывать однотипные алгоритмы обработки таких конструкций, не заботясь о роде и сохранности нижележащих узлов. «Жесткий» граф потока управления для текущего примера следующий (рисунок В.2). Согласно изображению графа, он отличается от «нежесткого» лишь наличием служебных узлов («[0]» и «[1]»), расположенных в начале каждой ветки условного выполнения.

Граф вызовов подпрограмм (CSG)

Данный ориентированный граф описывает для каждой подпрограммы вызовы всех других с помощью узлов, задающих подпрограммы, а также дуг между ними, определяющих сами вызовы. Для текущего примера граф состоит из одного узла, соответствующего подпрограмме «max2», поскольку она никаких вызовов других подпрограмм не делает.

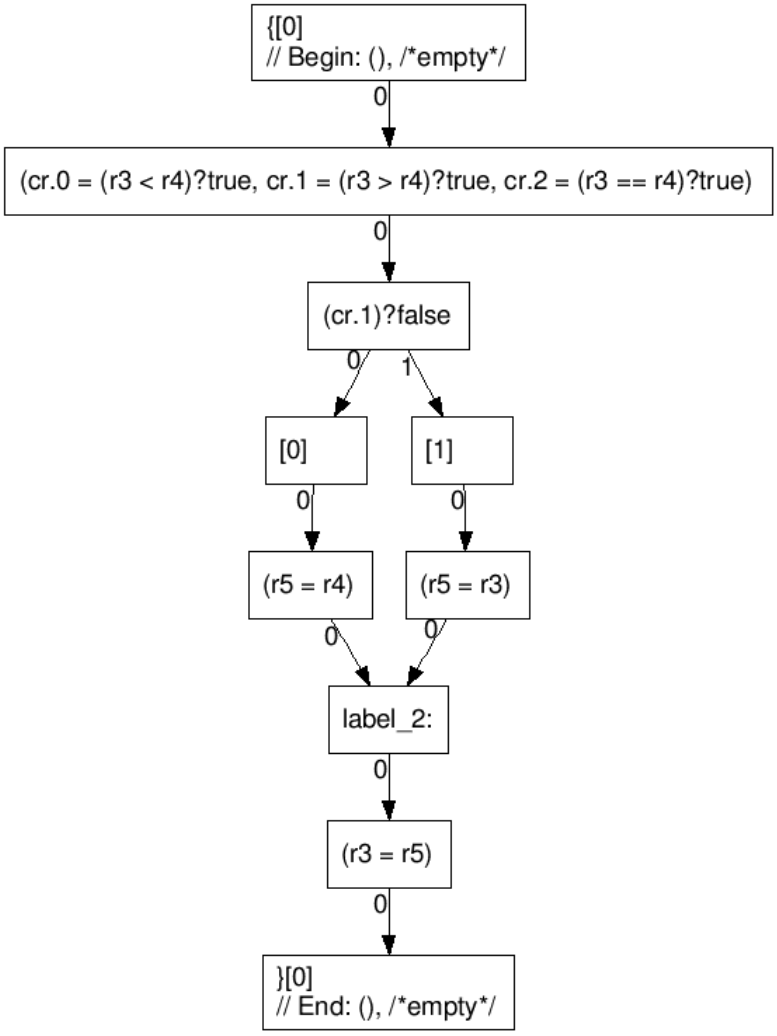

Рисунок В.2 – Пример «жесткого» графа потока управления для функции max2()

Дерево областей переменных и подпрограмм (SCT)

Данное дерево задает программные области, в которых расположены переменные (как глобальные, так и локальные) и подпрограммы. Логично, что глобальные переменные и подпрограммы в дереве расположены на одном уровне, а локальные – на подуровне соответствующей подпрограммы. Отметим, что в отличие от языка C, к локальным переменным относятся как входные/выходные параметры, так и регистр с адресом возврата из подпрограммы (для PowerPC это LR). Шаблон дерева приведен далее (рисунок В.3).

По причине тривиальности дерева областей переменных и подпрограмм, его отладочный вид представляется в текстовом виде и для текущего примера следующий:

```
// Declaration of Global Variables
// Declaration of Subprograms
(...) max2(...) lr: 41{}
```

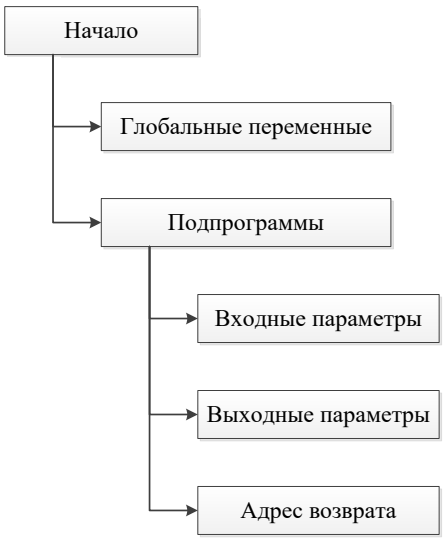

Рисунок В.3 – Шаблон дерева областей переменных и подпрограмм для функции max2()

Согласно описанию дерева, оно состоит из области глобальных переменных и области подпрограмм, локальные переменные которых заданы в их сигнатуре. По сути, текстовое описание дерева аналогично заголовочному файлу, используемому в языке С. Например, такое созданное в результате работы стадии 2 Утилиты дерево описывает подпрограмму «max2» с неизвестным списком входящих/выходящих параметров, у которой регистр LR (тождественный регистру R41) хранит адрес возврата из подпрограммы.

Информация об уязвимостях

Для хранения информации о найденных уязвимостях используются узлы деревьев и графов, которые могут хранить произвольные данные. В частности, существуют специальные служебные узлы, создаваемые в процессе создания и анализа S-модели при обнаружении подозрительных на уязвимости мест. Таким образом, конечное PCT уже в себе хранит все пометки об уязвимостях и их точных местах обнаружения, которые генерируются в текстовое алгоритмизированное Представление. Для указания уязвимостей, не имеющих жесткой привязки к коду, возможно добавление соответствующих узлов в близкие логически-связанные узлы (например, родительский блок) или к верхнему узлу дерева.

Дерево потока управления (CFT)

В процессе выполнения алгоритмизации граф потока управления преобразуется Утилитой в соответствующее дерево, основным отличием которого является *размыкание* зацикленности на графе, что приводит к структуризации Представления программы и ее подобности диаграмме Насси-Шнейдермана. В случае невозможности избавления от безусловных переходов, они хранятся в дереве, как ссылки одних элементов (очевидно, листьев деревьев) на другие. Такое преобразование производится путем выделения в графе СМД, их *умной* (в смысле применения нетривиальных авторских алгоритмов) пересортировки и построения нового дерева на их базе. Наличие признака структурированности в названии дерева подчеркивает соответствующее произведенное действие. Дерево потока управления для текущего примера следующее (рисунок В.4).

Результат визуального сравнения дерева (рисунок В.4) и графов потока управления (рисунки В.1 и В.2) позволяет утверждать, что представление стало более структурированным, хотя некоторые узлы и хранят ссылки (аналоги безусловного перехода) на другие (отмечены пунктирными линиями).

Дерево потока управления (оптимизированное)

На Стадии 4 производятся специальные действия, направленные на «улучшение» структурированного дерева потока управления, приводя его к оптимизированному; используемый для этого функционал приведен в описании Модуля оптимизации. Также оптимизируются и выражения, которые хотя и не влияют на топологию дерева, но анализ упрощенных выражений может потенциально приводить к удалению неисполняемых веток и отработке других оптимизаций, что впоследствии приведет к перестроениям дерева. Оптимизированное дерево потока управления для текущего примера приведено на рисунке В.5.

Как хорошо видно из рисунка, оптимизированное дерево подобно предыдущему – структурному, но выглядит более компактно, не имеет ссылок для безусловных переходов, а также лишних программных меток.

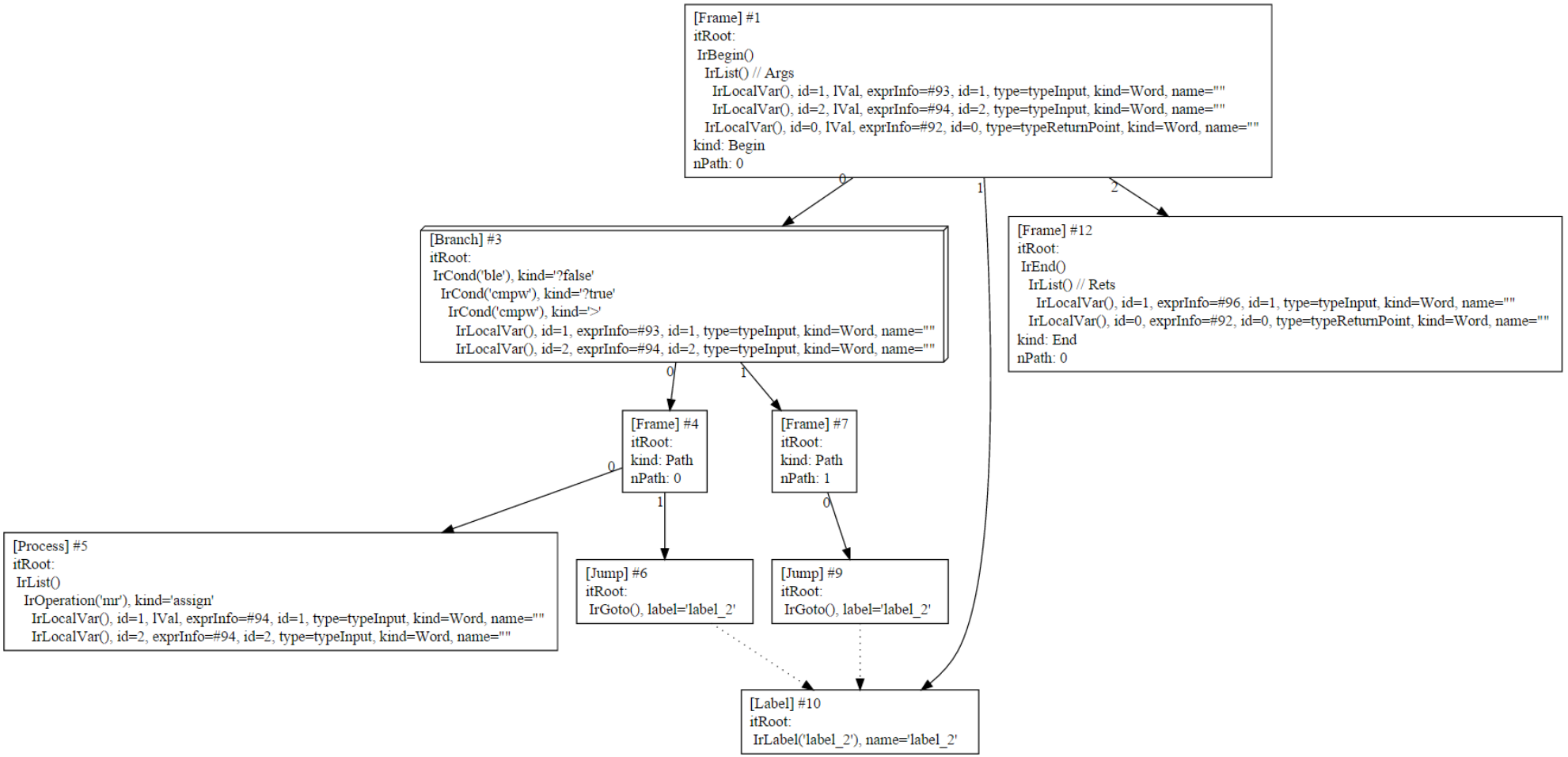

Рисунок В.4 – Пример дерева потока управления для функции max2()

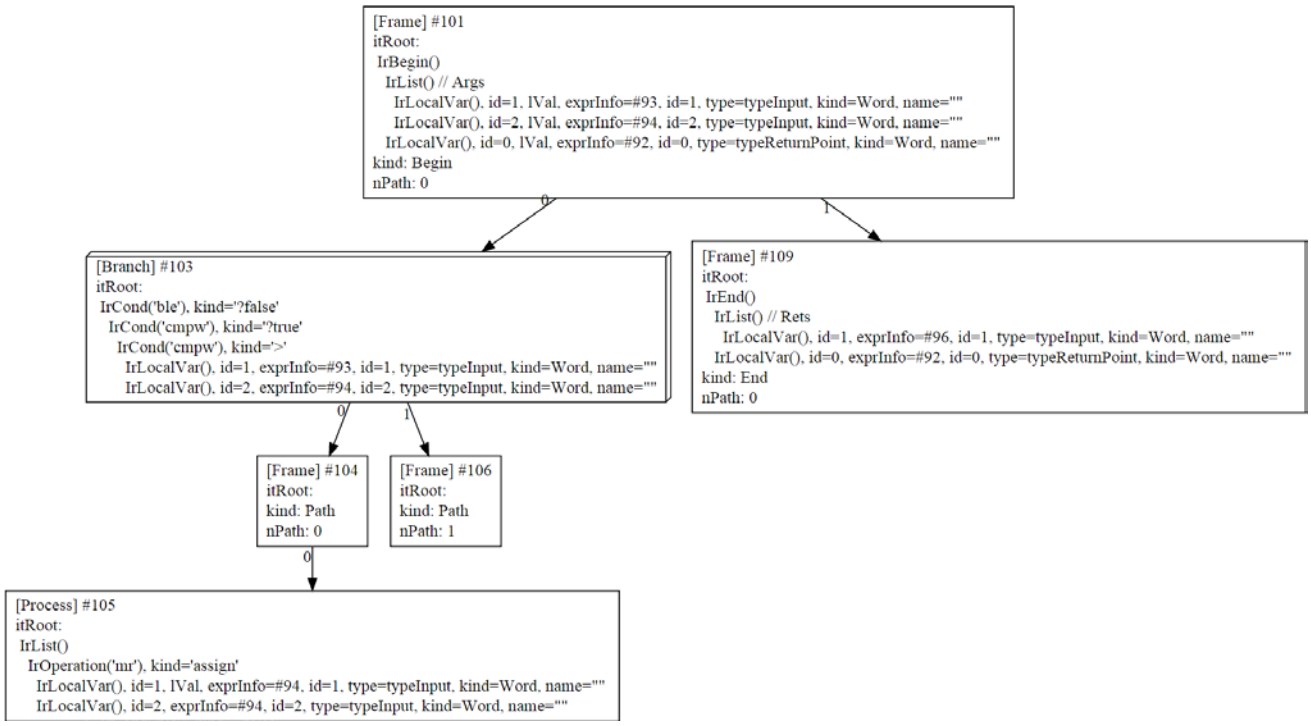

Рисунок В.5 – Пример оптимизированного дерева потока управления для функции max2()

Граф потока данных (DFG)

Данный граф топологически аналогичен потоку управления, но предназначен для хранения информации о качественных значениях переменных – не конкретных чисел или диапазонов, а неких условных идентификаторов их множества и временах их жизни – областях узлов графа, в которых значение переменной инициализируется, хранится или используется. Операции в узлах графа определяют изменения значений переменных, а точки схождения веток – их объединения. Также, первая инициализация переменной в данной ветке графа означает начало ее жизни, а последняя – конец. Верность этой информации определяется успешностью работы Модуля уточнения сигнатуры подпрограммы и внесенными вручную корректировками. Пример оптимизированного графа потока данных для текущего примера приведен на рисунке В.6.

Согласно графическому виду графа потока данных (рисунок В.6), он внешне полностью аналогичен графу потока управления (рисунки В.1 и В.2), однако его узлы содержат иную информацию – исключительно о переменных кода.

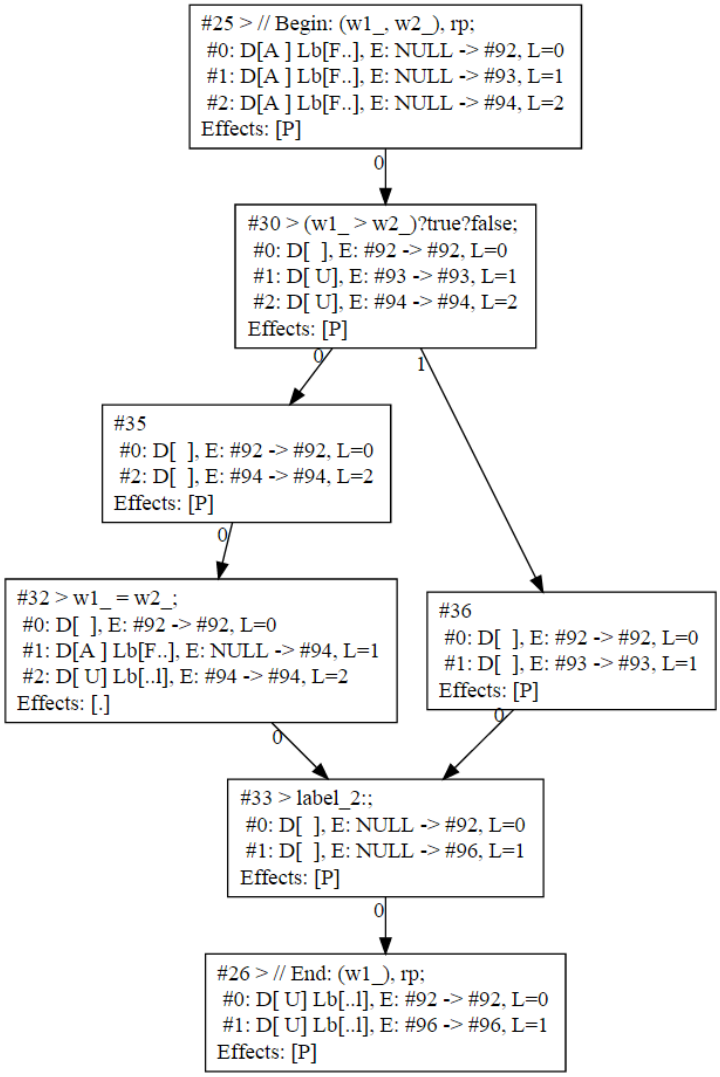

Рисунок В.6 – Пример графа потока данных для функции max2()

Такая информация закодирована с помощью специальных структур и флагов, основные из которых на изображении графа имеют следующие обозначения: #N – идентификатор переменной, D[] – массив флагов операций над переменной (A или Assign – присваивание, U или Usage – использование), Lb – массив флагов времени жизни переменной (F – начало жизни, l – конец жизни), E: #N1 -> #N2 – идентификатор значения переменной с присвоением его другому, L = N – идентификатор времени жизни переменной, Effects – массив флагов эффектов узлов (P или Permanent – узел служебный и не может быть удален).

Граф зависимости вычислений

Граф предназначен для хранения информации о связи между вычисляемыми значениями переменных, соответствующих присваиваемым выражениям. Такая

информация используется для распределения значений, хранящихся на множестве регистров начального ассемблера, на более компактное множество переменных конечного восстановленного алгоритма, а также для упрощения математических выражений – т. е. используется в процессе оптимизации. Узлы графа хранят идентификаторы выражений, а дуги – связи между их вычислениями. Пример графа зависимостей вычислений для текущего примера приведен на рисунке В.7.

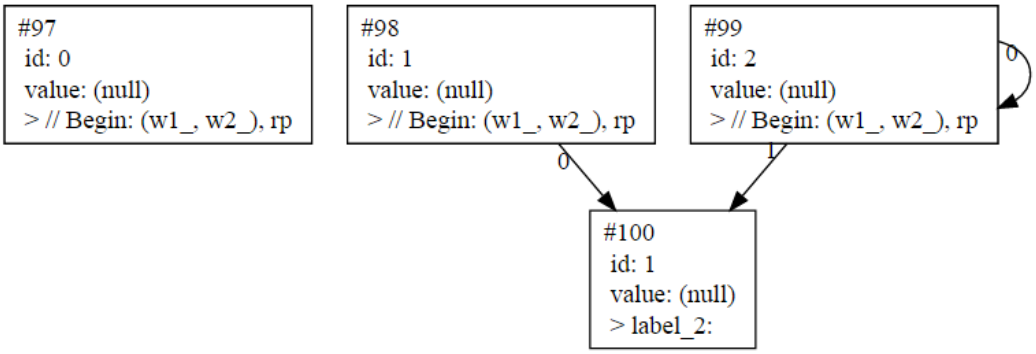

Рисунок В.7 – Пример графа зависимостей вычислений для функции max2()

Необходимо отметить, что идентификаторы значений переменных на приведенных графах потока данных и зависимостей вычислений не совпадают, поскольку последний в процессе оптимизаций неоднократно перестраивается с использованием новых значений идентификаторов. Тем не менее, можно определить следующее соответствие между идентификаторами выражений этих двух графов: #92 = #97, #93 = #98, #94 = #99, #96 = #100.

Дерево псевдокода (PCT)

Дерево описывает восстановленные алгоритмы в виде псевдокода, сгенерированного по оптимизированному дереву потока управления и прошедшему лаконизацию. Представление дерева можно считать конечным, полностью готовым к генерации текстового описания (или, например, графического). PCT для текущего примера следующее.

```
IrList()     // Root
  IrFunction()
    IrIdent('max2')
    IrList() // Args
      IrLocalVar(), id=1, lVal, exprInfo=#93, id=1, type=typeInput, kind=Word, name=""
      IrLocalVar(), id=2, lVal, exprInfo=#94, id=2, type=typeInput, kind=Word, name=""
    IrList() // Rets
```

```
  IrReg(), id=3, exprInfo=(NULL)
IrLocalVar(), id=0, lVal, exprInfo=#92, id=0, type=typeReturnPoint, kind=Word, name=""
IrBlock()
  IrIfElse()
    IrCond('ble'), kind='?true'
      IrCond('cmpw'), kind='<='
        IrLocalVar(), id=1, exprInfo=#93, id=1, type=typeInput, kind=Word, name=""
        IrLocalVar(), id=2, exprInfo=#94, id=2, type=typeInput, kind=Word, name=""
    IrBlock()
      IrOperation('mr'), kind='assign'
       IrLocalVar(), id=1, lVal, exprInfo=#94, id=1, type=typeInput, kind=Word, name=""
        IrLocalVar(), id=2, exprInfo=#94, id=2, type=typeInput, kind=Word, name=""
    IrBlock()
  IrReturn()
    IrList()   // Rets
      IrLocalVar(), id=1, exprInfo=#96, id=1, type=typeInput, kind=Word, name=""
    IrLocalVar(), id=0, exprInfo=#92, id=0, type=typeReturnPoint, kind=Word, name=""
```

В дереве присутствуют новые типы узлов, зависимые от конечного языка представления алгоритмов и не встречающиеся в АК (как и в его дереве абстрактного синтаксиса), такие как: IrIfElse – конструкция веток условного перехода «IF () THEN {} ELSE {}», IrReturn – возврат из подпрограммы, IrLocalVar – локальная переменная (с типами: typeInput – для входных параметров и typeReturnPoint – для адреса возврата подпрограммы). Также очевидно, что данное дерево стало значительно меньше дерева абстрактного синтаксиса (24 строки против 59), притом с сокращением используемых переменных (2 против 3-х). Таким образом, псевдокод однозначно можно считать более структурированным и компактным, чем ассемблерный.

Вспомогательные данные

В Утилите используются также вспомогательные данные, обеспечивающие построение и обработку внутренних представлений. Во-первых, это хранилище глобальных настроек, определяющих визуальный стиль восстановленных алгоритмов, генерацию комментариев и информации об используемых процессорных регистрах, выбор внутренних представлений для отладочного вывода. Во-вторых, это значения цветов раскраски, используемые как для самой работы алгоритмов, так и для визуального отображения их результатов на графах и деревьях. В-третьих, это таблица регистров процессора входного ассемблера, содержащая их имена со свойствами, используемая в процессе синтаксического анализа. И, в-четвертых, это объект с текущей версией Утилиты и историей предыдущих изменений.

ПРИЛОЖЕНИЕ Г. Алгоритмы модулей Прототипа

Описаны принцип и блок-схема алгоритмов модулей прототипа ПС алгоритмизации. Используется нумерация алгоритмов модулей в следующем формате: M_X_Y, где X – номер стадии, а Y – номер модуля в стадии; для сквозных модулей номер стадии обозначается, как C.

Алгоритм M_1_1. Лексический анализатор

Принцип работы модуля основан на разбиении входного текста ассемблерного кода в виде потока символов на лексемы согласно заданным регулярным выражениям. Блок-схема алгоритма представлена на рисунке Г.1.

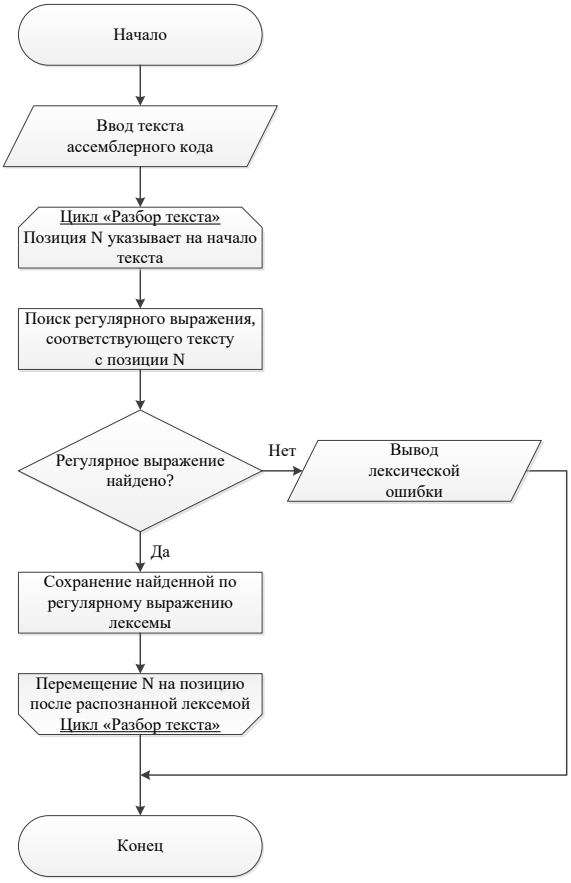

Рисунок Г.1 – Блок-схема алгоритма лексического анализатора

Загруженный входной АК посимвольно обрабатывается – производится разбор текста, и проверяется на соответствие каждого блока текста регулярным выражениям, с каждым из которых сопоставлена определенная лексема. В случае успешной проверки, лексема сохраняется, блок текста для последующей обработ-

ки устанавливается на следующий символ, цикл повторяется. Если входной блок текста не соответствует ни одному из регулярных выражений, то возвращается лексическая ошибка и работа ПС прекращается.

Алгоритм M_1_2. Синтаксический анализатор

Принцип работы модуля основан на комбинировании потока лексем согласно заданным правилам – так называемая *свертка*. Схема работы модуля соответствует типичному конечному автомату, переходящему между состояниями согласно получаемым лексемам. Блок-схема алгоритма представлена на рисунке Г.2.

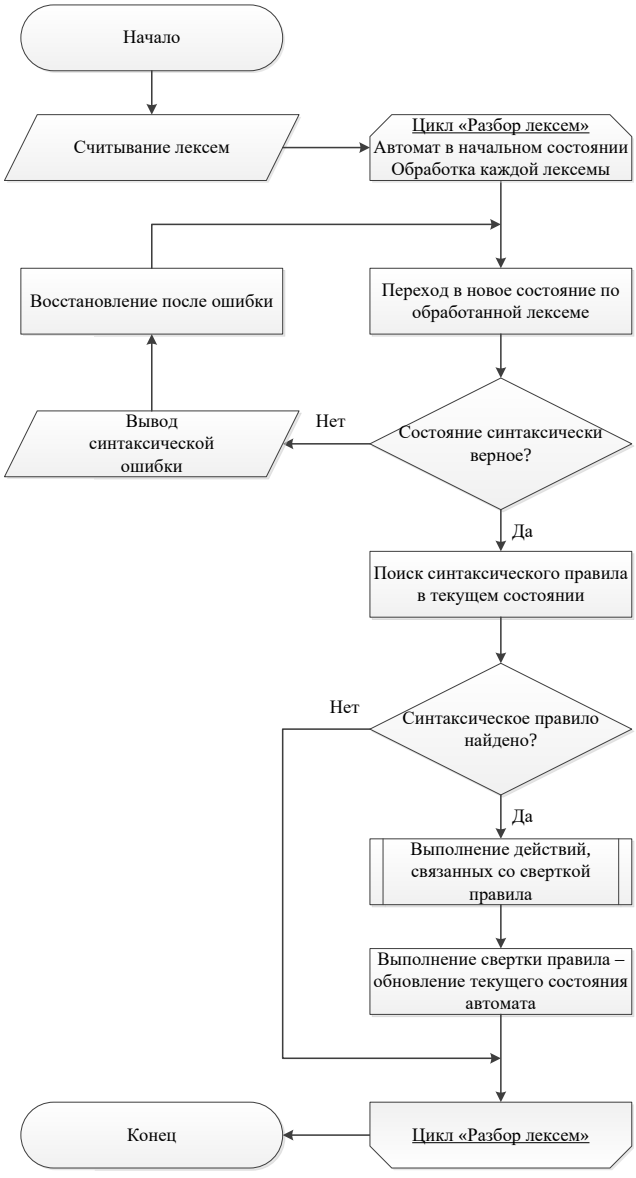

Рисунок Г.2 – Блок-схема алгоритма синтаксического анализатора

Производится считывание лексем, полученных лексическим анализатором. Конечный автомат на основании потока лексем переходит в различные состояния (сгенерированные на основании синтаксиса входного языка). В случае, если состояние (а, следовательно, и предшествующий поток лексем) соответствует определенному правилу, происходит его свертка и выполнение пользовательского кода. Если поток лексем не соответствует ни одному из синтаксических правил (конечный автомат перешел в состояние ошибки), то выдается синтаксическая ошибка и осуществляется попытка восстановления после нее для последующей обработки – например, путем опускания символов до конца строки.

Алгоритм M_1_3. Семантический анализатор

Принцип работы модуля основан на построении AST входного кода согласно произведенным сверткам и синтаксическому значению правил. В случае свертки правила корректировки алгоритмизации (заданной через расширение синтаксиса входного ассемблерного кода) соответствующая информация сохраняется и используется последующими модулями. Блок-схема алгоритма представлена на рисунке Г.3.

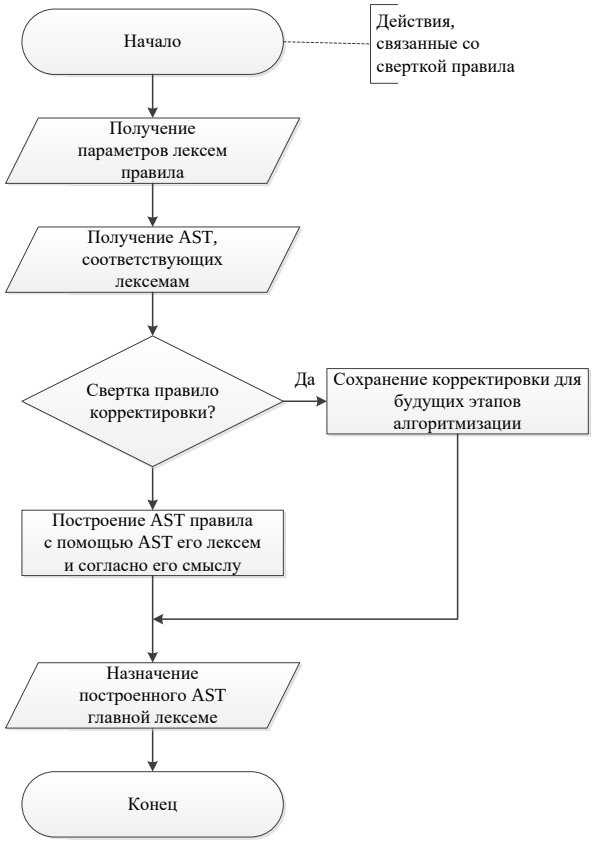

Рисунок Г.3 – Блок-схема алгоритма семантического анализатора

При свертке правил для АК синтаксическим анализатором, пользовательский код производит построение соответствующей части AST, используя узлы дерева, ассоциированные с разобранными лексемами. Корневой узел построенного дерева ассоциируется с текущей лексемой и используется в сборках остальных правил. Если разобранное правило является корректировками алгоритмизации, то информация о последних добавляется во внутренние структуры ПС и используется следующими модулями.

Алгоритм M_2_1. Модуль выделение подпрограмм

Принцип работы модуля основан на анализе AST и выделении в нем поддеревьев, относящихся к отдельным подпрограммам. Также создаются глобальные переменные, указанные явно в ассемблерном коде. Блок-схема алгоритма представлена на рисунке Г.4.

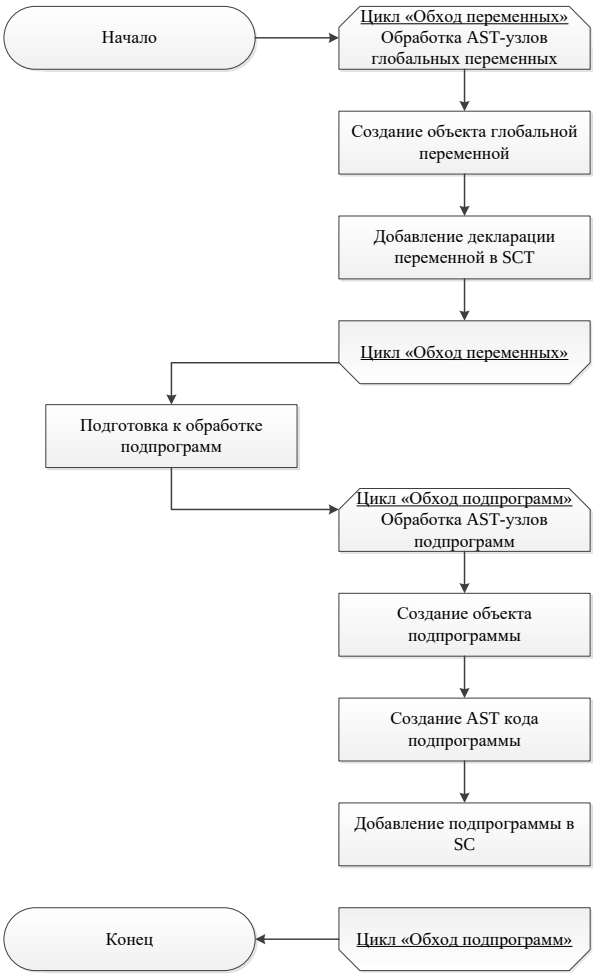

Рисунок Г.4 – Блок-схема модуля выделения подпрограмм

Осуществляется обход веток общего AST, соответствующих указанным в АК глобальным переменным и подпрограммам. В первом случае, создаются внутренние объекты глобальных переменных, и информация о них добавляется в SC. Во-втором случае создаются внутренние объекты подпрограмм с ассоциированным AST их кода, и информация о них также добавляется в SC.

Алгоритм M_2_2. Модуль построения графа потока управления

Принцип работы модуля основан на анализе AST каждой из выделенных подпрограмм и построении графа потока управления. Узлы графа содержат собственные AST, соответствующие элементарным выполняемым операциям. Также осуществляется выделение входных и выходных аргументов, заданных корректировками алгоритмизации. Параллельно строится граф вызовов подпрограмм по соответствующим операциям. Блок-схема алгоритма представлена на рисунке Г.5.

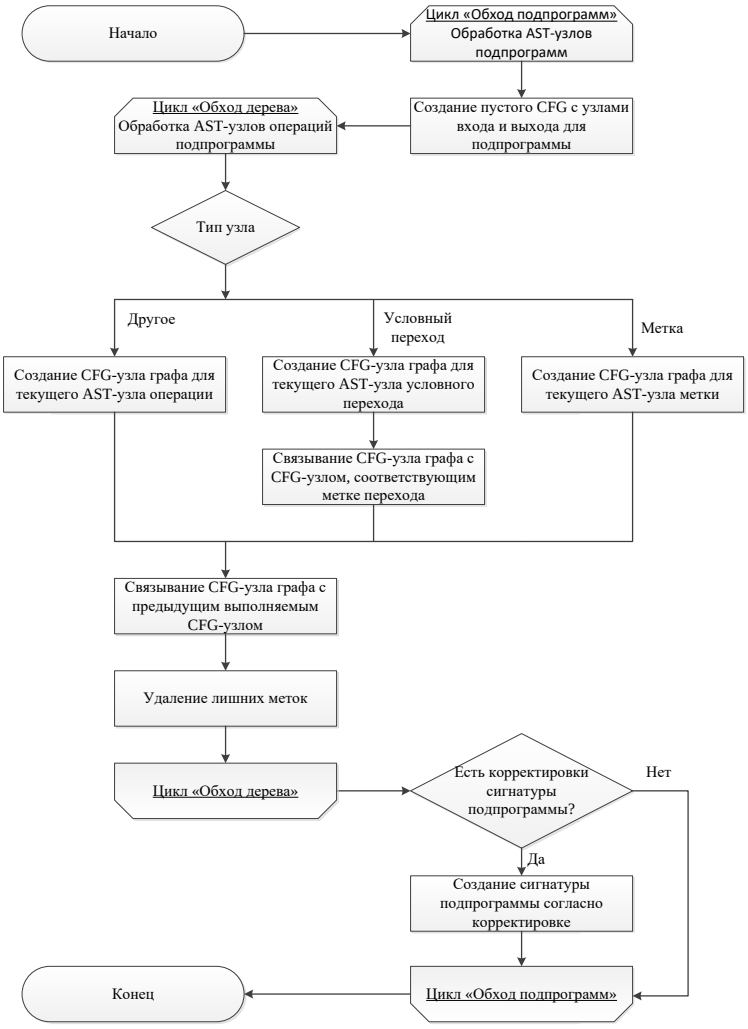

Рисунок Г.5 – Блок-схема модуля построения CFG

Производится обход всех подпрограмм и создание соответствующих им CFG-узлов. Затем используя типы AST-узлов кода подпрограммы строится CFG следующим образом: условный переход приводит к разветвлению графа, метка используется для связывания переходов, а остальные операции добавляются в предыдущий узел графа. Лишние элементы, такие, как неиспользуемые или парные метки, удаляются. Также, уточняется сигнатуры подпрограммы на основании корректировок.

Алгоритм M_2_3. Модуль выделение глобальных переменных

Принцип работы модуля основан на анализе AST каждой из выделенных подпрограмм и определении глобальных переменных по идентификаторам, параллельно строя SCT. Блок-схема алгоритма представлена на рисунке Г.6.

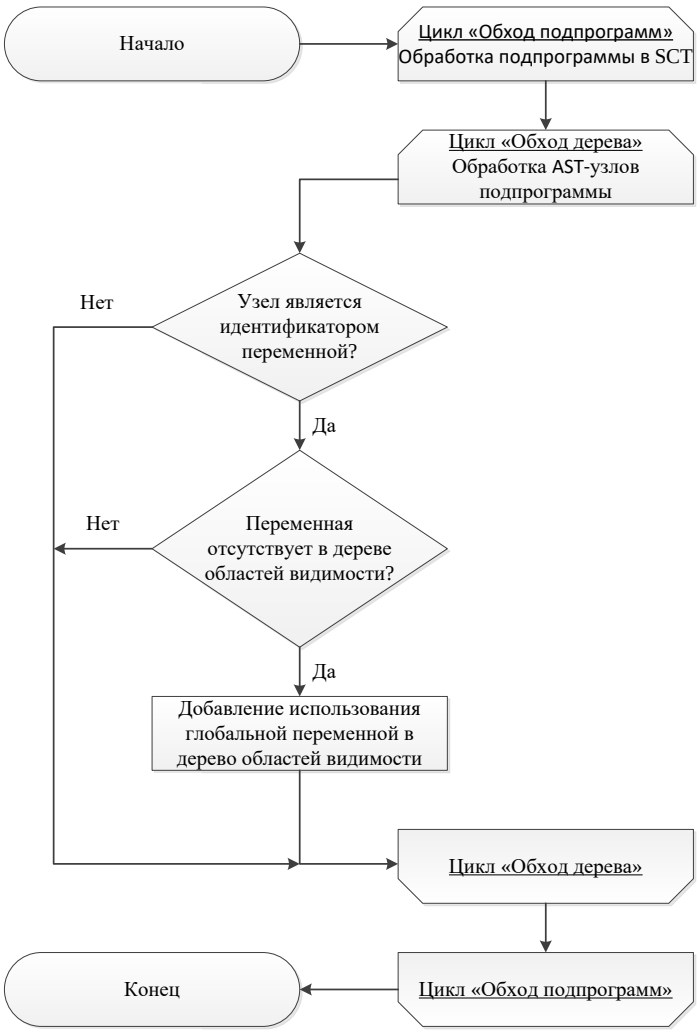

Рисунок Г.6 – Блок-схема модуля выделения глобальных переменных

Производится обход всех подпрограмм и узлов их AST кода, поиск используемых идентификаторов глобальных переменных и добавление необходимых в SCT.

Алгоритм M_3_1. Модуль построения графа потока данных

Принцип работы модуля основан на анализе использовании переменных в каждом узле CFG и сборе следующей информации: первое и последнее использование, хранимое значение, его связь со значениями других переменных. Это позволяет, как получить информацию о временах жизни переменных, так и построить граф поток данных DFG – по которому определяема зависимость между вычислениями. Блок-схема алгоритма представлена на рисунке Г.7.

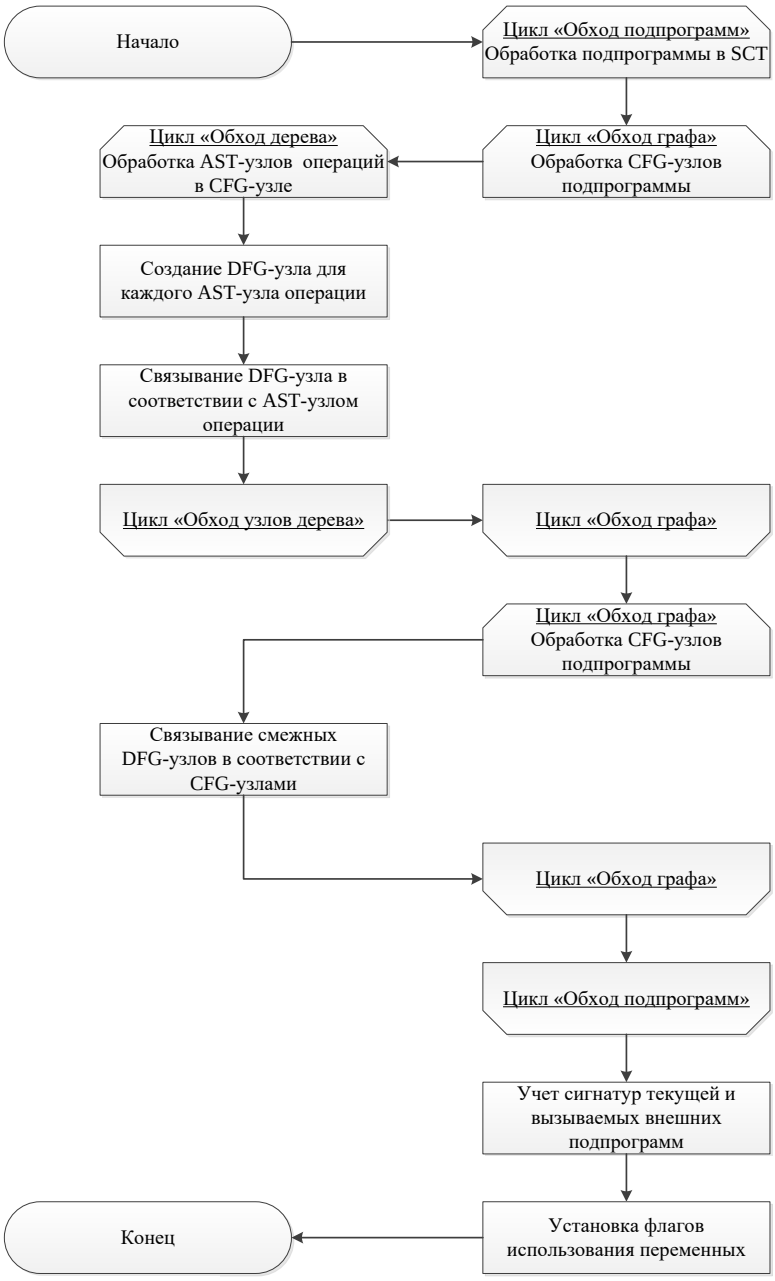

Рисунок Г.7 – Блок-схема модуля построения DFG

Производится обход всех подпрограмм в SCT, их CFG-узлов и AST-узлов для операция последних. Для каждого AST-узла создается соответствующий DFG-узел, хранящий информацию обо всех используемых переменных и их свойствах. Затем повторным обходом CFG-узлов производится связывание DFG-узлов. Учитываются вызовы подпрограмм и их сигнатуры, дополнительно проставляются флаги использования переменных.

Алгоритм M_3_2. Модуль уточнения сигнатуры подпрограмм

Принцип работы модуля основан на учете информации о временах жизни переменных, а именно первому и последнему чтению их значения. Используемые без инициализации переменных считаются входными аргументами подпрограммы, а не используемые после инициализации – выходными. Блок-схема алгоритма представлена на рисунке Г.8.

Производится обход всех подпрограмм в SCT и DFG-узлов их кода. Каждая переменной, используемая в начале кода без инициализации, помечается как входной параметр. Каждая переменной, используемая в конце кода, помечается как выходной параметр. На основании помеченных переменных обновляется сигнатура текущей подпрограммы.

Алгоритм M_3_3. Модуль выделения структурных метаданных

Метод является наиболее существенным с точки зрения эффективности алгоритмизации, поскольку результаты его работы отражаются на конечной структурированности алгоритмизированного представления.

Принцип работы модуля основан на раскраске CFG и определении ветвлений (условных переходов) и циклов в нем; их замена на отдельные управляющие структуры «размыкает» цикличности и преобразовывает граф в CFT – аналогичное представлению Насси-Шнейдермана. Блок-схема алгоритма представлена на рисунке Г.9.

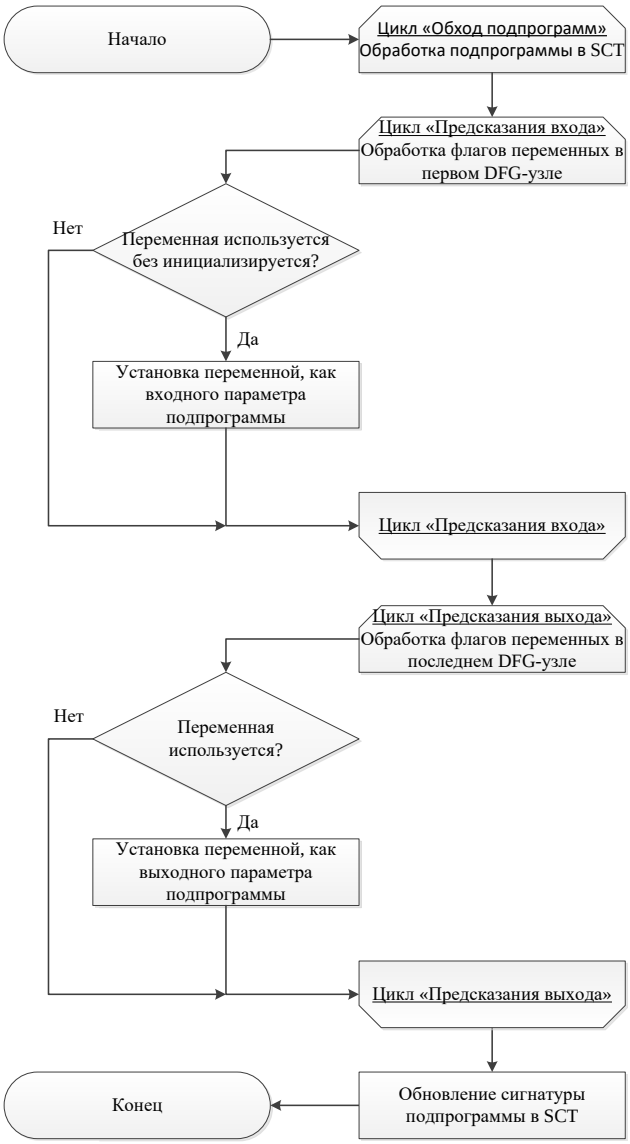

Рисунок Г.8 – Блок-схема модуля уточнения сигнатуры подпрограмм

Производится обход всех подпрограмм в SCT и рекурсивная обработка CFG-узлов с первого и согласно их типам. В процессе этого строится CFT. Если текущий узел является условным переходом, то создается соответствующий CFT-узел; затем к его дочерним узлам прикрепляются CFT-узлы, полученные на основании рекурсивной обработки каждой ветки условного перехода. Также, конец одной ветки отсоединяется от точки входа во вторую – осуществляется структуризация. Если текущий узел является меткой, в которую при этом есть переход (помимо перехода из предыдущей операции), то последующая часть CFG-узлов расценивается, как цикл с созданием соответствующего CFT-узла; CGT-тело цикла получается рекурсивной обработкой CFG-узлов, аналогично веткам условного перехода.

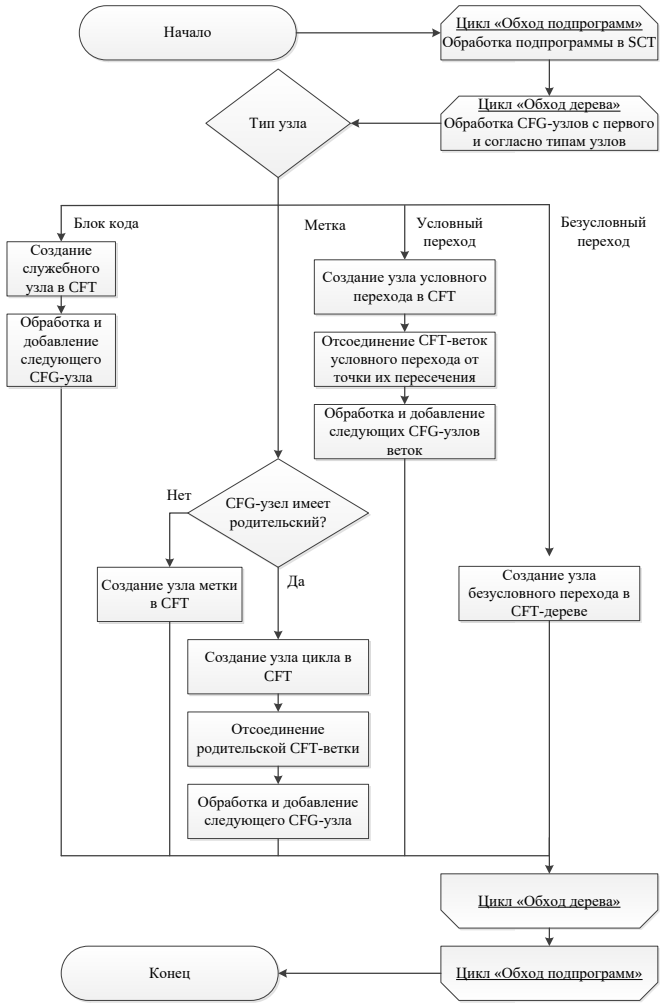

Рисунок Г.9 – Блок-схема модуля выделения структурных метаданных

Также, все выходы из цикла отсоединяются от точек входа во внешние ветки – осуществляется структуризация Текущие узлы одиночных меток, безусловных переходов и блоки кода добавляются в CFT как есть; для последних создаются служебные узлы для их выделения. Таким образом, алгоритм рекурсивно преобразует граф в дерево, структурируя таким образом представление кода; в случае невозможности такого в конечном дереве применяются безусловные переходы на метки между различными ветками дерева.

Алгоритм M_4_1. Модуль оптимизации структурных метаданных

Принцип работы модуля основан на выполнении специализированных оптимизационных действий. Последние производят сопоставление топологии CFT

заданным шаблонам и его перестроение по связанным правилам. Блок-схема алгоритма представлена на рисунке Г.10.

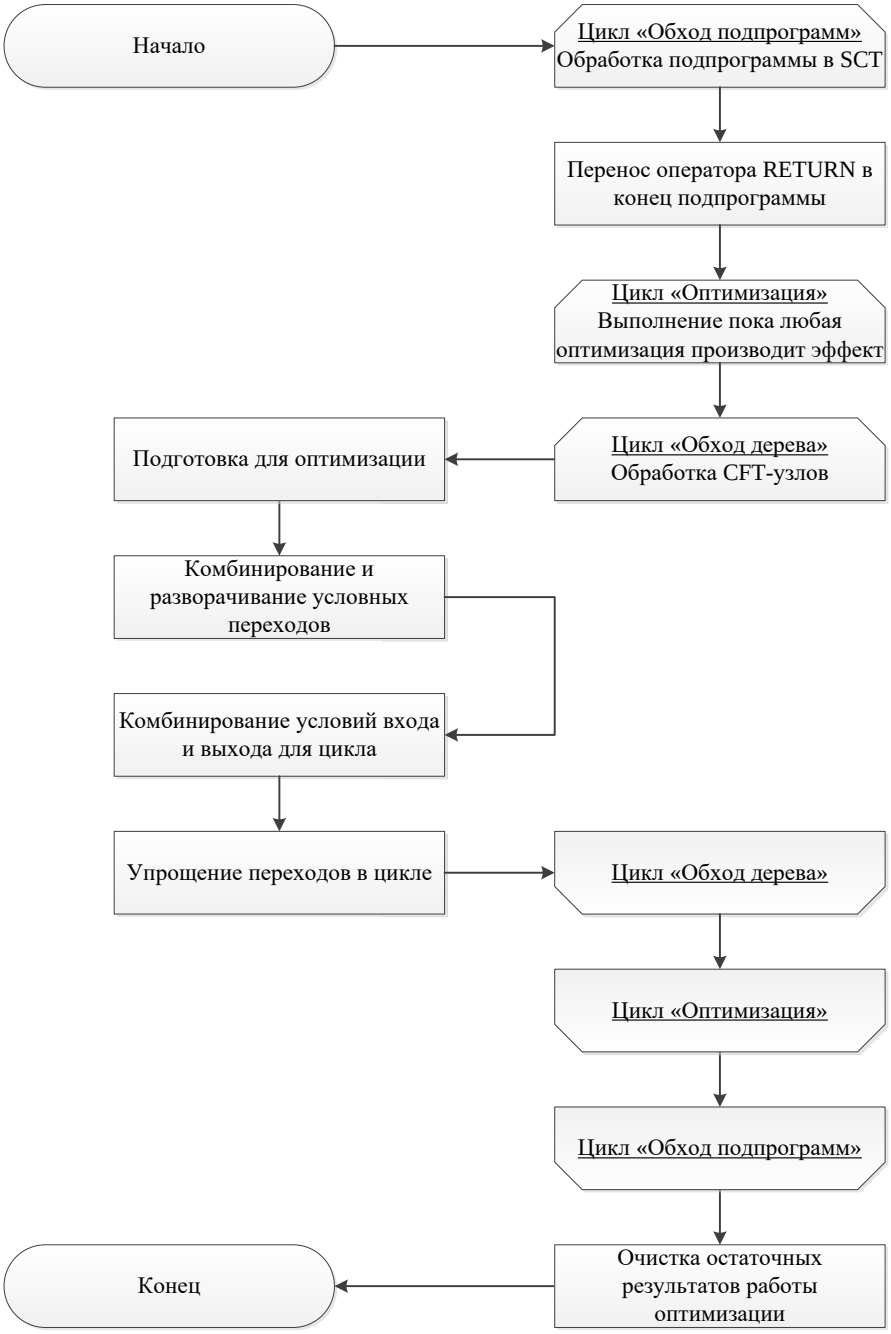

Рисунок Г.10 – Блок-схема модуля оптимизации структурных метаданных

Производится обход всех подпрограмм в SCT и циклическое выполнение оптимизационных действий, пока суммарный эффект от каждых не будет равен нулю. Для этого, каждая оптимизация возвращает количество внесенных во внутреннее представление кода изменений. Выполняемые действия соответствуют функциональным возможностям соответствующего модуля на Стадии 4.

Алгоритм M_4_2. Модуль оптимизации дерева потока управления

Принцип работы модуля основан на выполнении специализированных оптимизационных действий. Последние производят сопоставление областей CFT заданным шаблонам и их замене на более короткие. Блок-схема алгоритма представлена на рисунке Г.11.

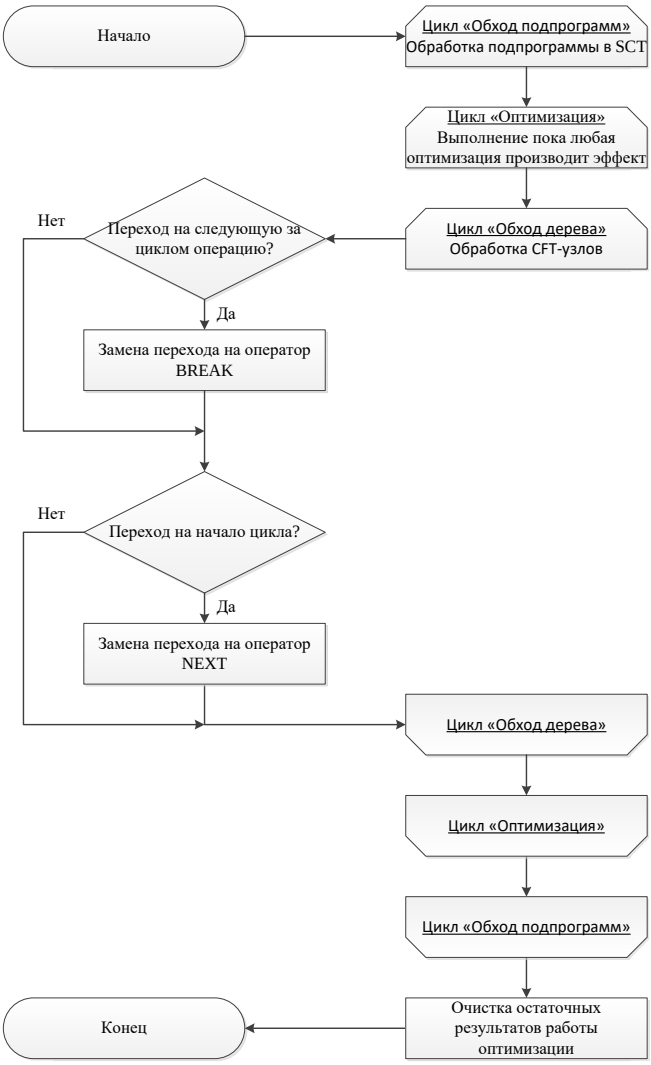

Рисунок Г.11 – Блок-схема модуля оптимизации CFT

Производится обход всех подпрограмм в SCT и циклическое выполнение оптимизационных действий, пока суммарный эффект от каждых не будет равен нулю. Для этого, каждая оптимизация возвращает количество внесенных во внутреннее представление кода изменений. Выполняемые действия соответствуют функциональным возможностям соответствующего модуля на Стадии 4.

Алгоритм M_4_3. Модуль оптимизации вычислений

Принцип работы модуля основан на выполнении специализированных оптимизационных действий. Последние производят анализ операций кода в узлах CFT, вычисление значений переменных, оценку эффекта от выполнения операции с последующим изменением операций. Блок-схема алгоритма представлена на рисунке Г.12.

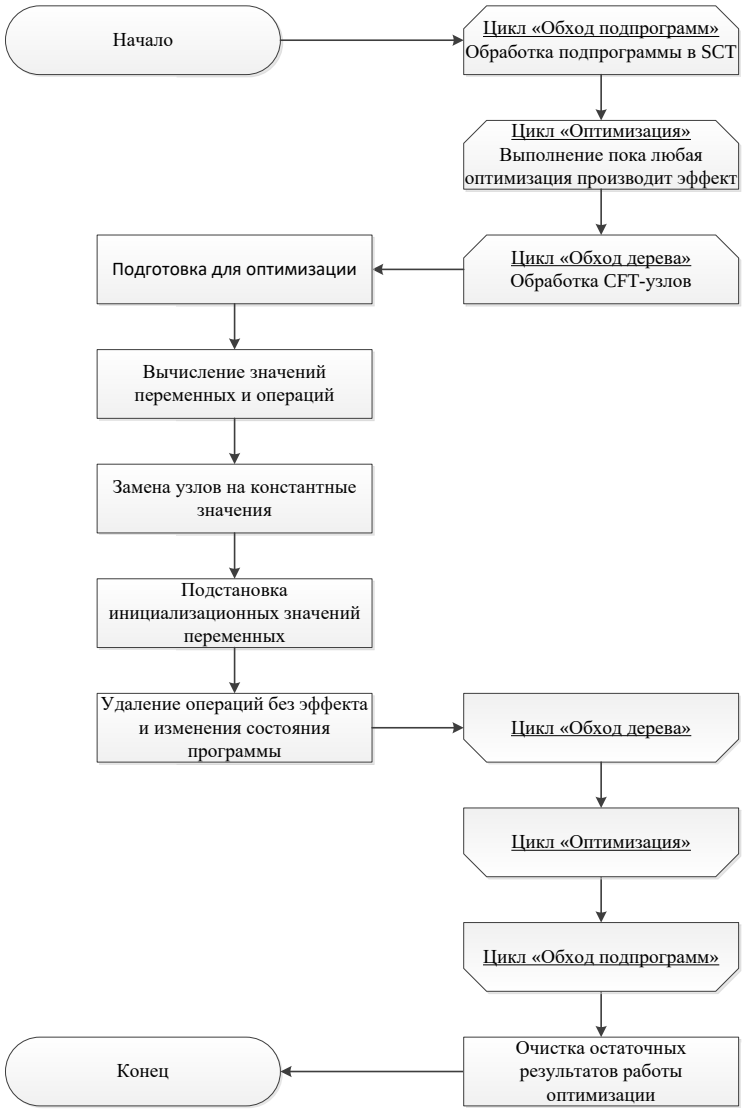

Рисунок Г.12 – Блок-схема модуля оптимизации вычислений

Производится обход всех подпрограмм в SCT и циклическое выполнение оптимизационных действий, пока суммарный эффект от каждых не будет равен нулю. Для этого, каждая оптимизация возвращает количество внесенных во внут-

реннее представление кода изменений. Выполняемые действия соответствуют функциональным возможностям соответствующего модуля на Стадии 4.

Алгоритм M_5_1. Модуль генерации псевдокода глобальных переменных

Принцип работы модуля основан на обходе ветки SCT, содержащей глобальные переменные, и создании соответствующих узлов в PCT. Блок-схема алгоритма представлена на рисунке Г.13.

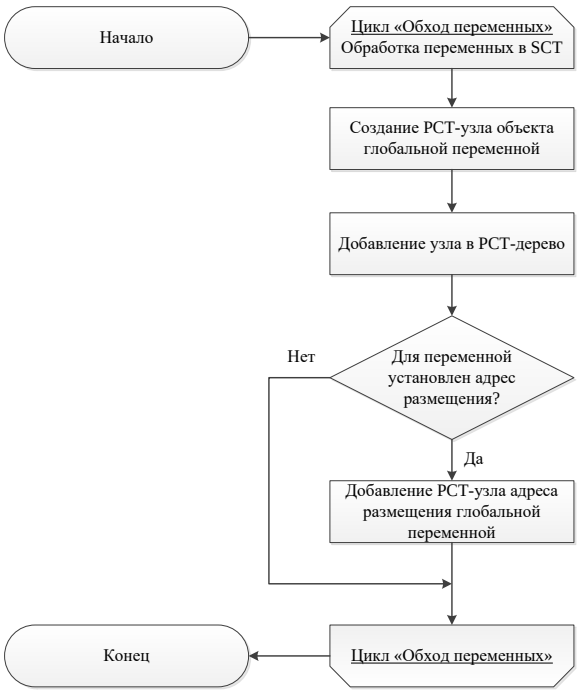

Рисунок Г.13 – Блок-схема модуля генерации псевдокода глобальных переменных

Производится обход всех переменных в SCT и создание соответствующих узлов в PCT. Если для переменной определен адрес размещения, то он также добавляется в PCT.

Алгоритм M_5_2. Модуль генерации псевдокода подпрограмм

Принцип работы модуля основан на обходе ветки SCT, содержащей подпрограммы, и создании соответствующих узлов в PCT. Блок-схема алгоритма представлена на рисунке Г.14.

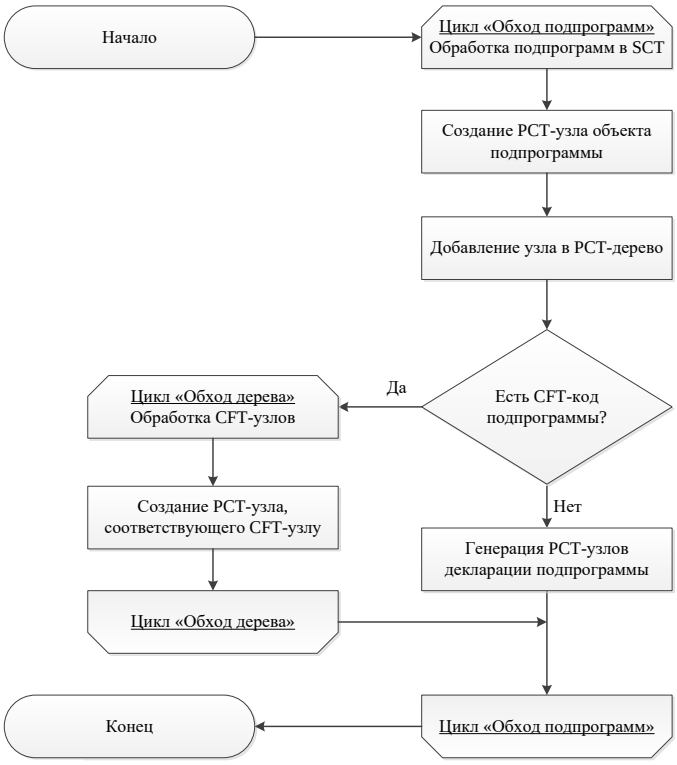

Рисунок Г.14 – Блок-схема модуля генерации псевдокода подпрограмм

Производится обход всех подпрограмм в SCT и создание соответствующих узлов в PCT. Если для подпрограммы присутствует код, то создается его представление и добавляется в PCT. Подпрограммы без кода имеют вид декларации.

Алгоритм M_5_3. Модуль лаконизации псевдокода

Принцип работы модуля основан на выполнении специализированных оптимизационных действий в интересах повышения восприятия кода человеком. Модуль обрабатывает операции в PCT подпрограмм, сопоставляет их с шаблонами и заменяет узлы на более информативные (с позиции человека). Блок-схема алгоритма представлена на рисунке Г.15.

Производится обход всех подпрограмм в SCT и циклическое выполнение действий по лаконизации, пока суммарный эффект от каждых не будет равен нулю (в соответствии с функциональными возможностями соответствующего модуля на Стадии 5). Так, производится поиск PCT-поддеревьев по шаблонам и их перепись по заданным правилам.

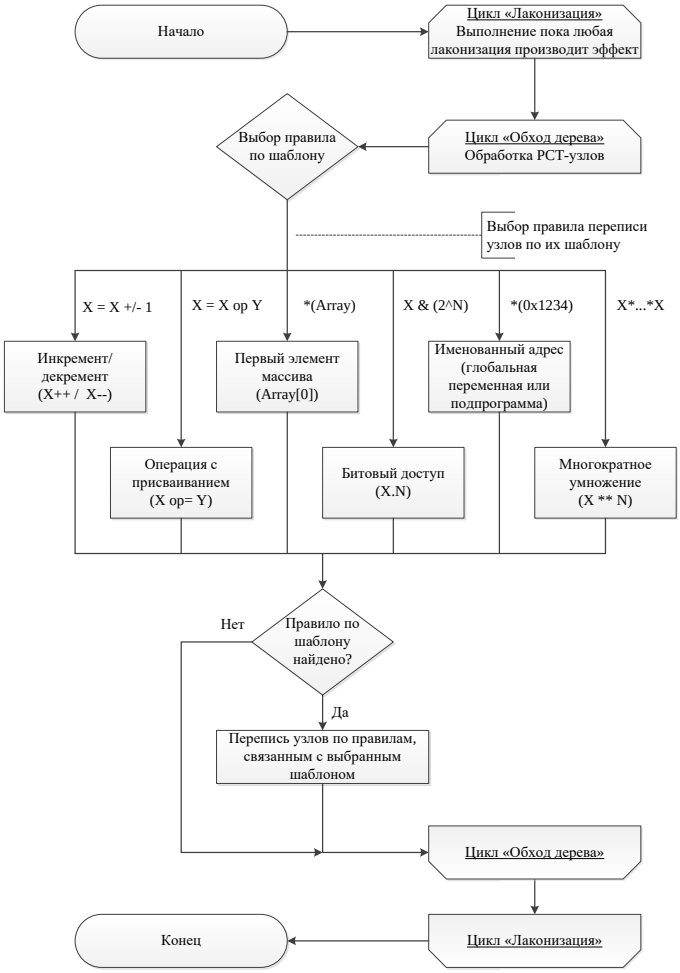

Рисунок Г.15 – Блок-схема модуля лаконизации псевдокода

Алгоритм M_5_4. Модуль генерации архитектуры

Принцип работы модуля основан на учете взаимосвязи глобальные переменных, используемых различными подпрограммами. Для этого по PCT строится промежуточный граф, отражающий все такие взаимосвязи, и производится его *разноцветная* раскраска так, чтобы только смежные узлы имели одинаковый цвет. Если несколько подпрограмм используют одну переменную, то считается, что они объединены в один модуль – на графе их цвет будет одинаковым. Информация о модулях добавляется в общее PCT. Вызовы подпрограмм одним модулем из другого осуществляются посредством интерфейсов последнего. Блок-схема алгоритма представлена на рисунке Г.16.

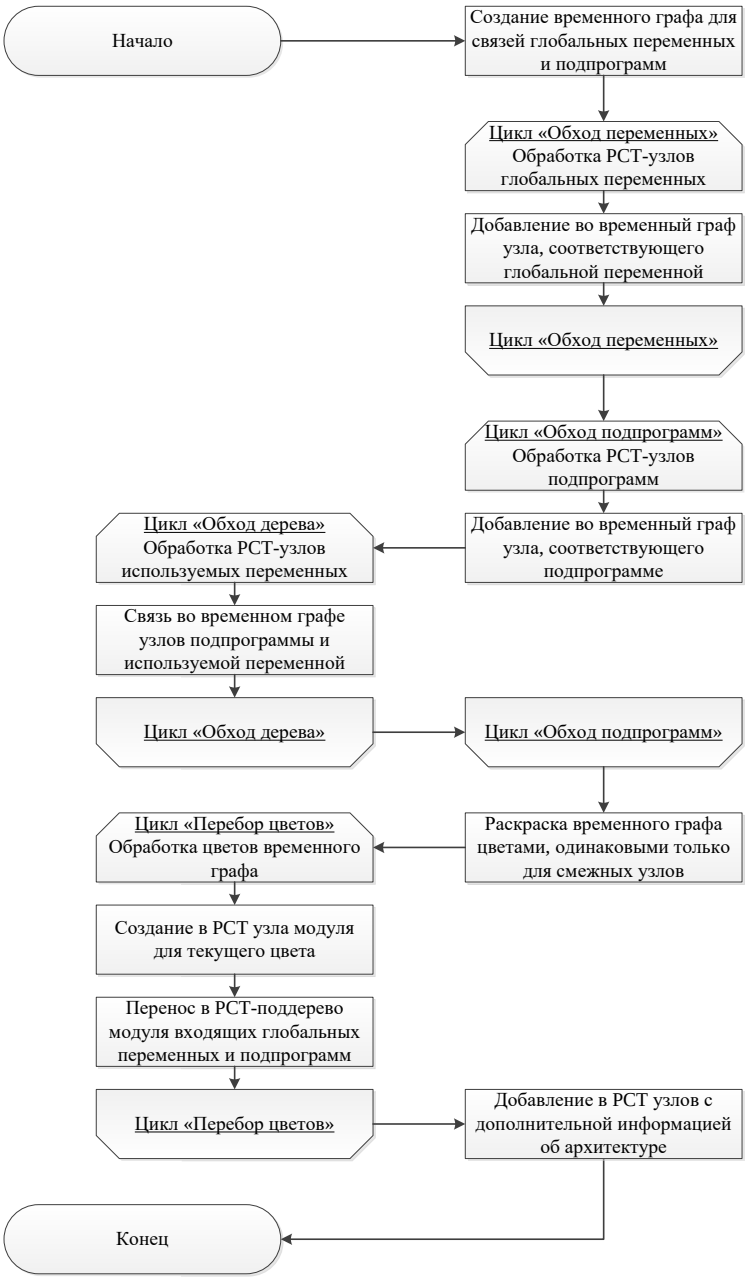

Рисунок Г.16 – Блок-схема модуля генерации архитектуры

Производится обход всех глобальных переменных и подпрограмм в SCT, а также построение временного графа, связывающего их взаимное использование. Затем, каждому узлу графа назначается определенный цвет. Для этого берется любой нераскрашенный узел, задается уникальный цвет (некий порядковый номер); затем берутся ближайшие узлы (подпрограммы или переменные) и им назначается такой же цвет. Когда текущим цветом раскрашены все узлы начиная с первого выбранного, выбирается следующий нераскрашенный узел и операция повторяется. Затем перебираются все созданные цвета, а узлы этих цветов объ-

единяются в один модуль; для этого в РСТ добавляется узел модуля, дочерними узлами которого делаются одноцветные узлы глобальных переменных и подпрограмм. Также, в РСТ-узлы добавляется дополнительная информация об архитектуре, такая, как количество вызовов подпрограммы одного модуля из другой.

Алгоритм М_6_1. Модуль генерации текстового описания корректировок алгоритмизации

Принцип работы модуля основан на генерации корректировок алгоритмизации, введенных пользователем в ассемблерном коде. Генерируемый формат идентичен понимаемому Утилитой на входе. Блок-схема алгоритма представлена на рисунке Г.17.

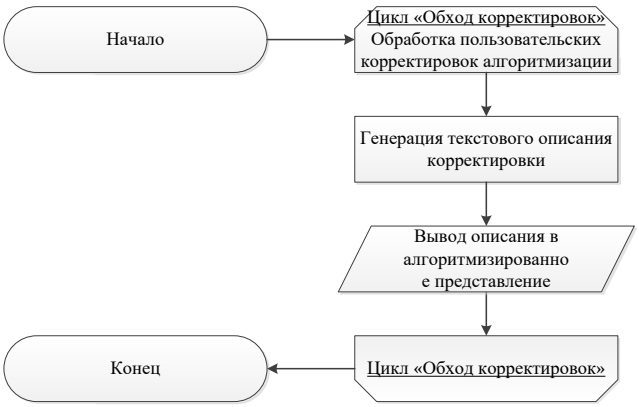

Рисунок Г.17 – Блок-схема модуля генерации текстового описания корректировок алгоритмизации

Производится обход всех корректировок, хранимых в ПС, генерация их текстового описания и вывод последнего в алгоритмизированное представление.

Алгоритм М_6_2. Модуль генерации текстового описания архитектуры

Принцип работы модуля основан на обходе поддерева псевдокода элементов архитектуры и генерации для каждого узла его текстового описание в алгоритмизированном. Блок-схема алгоритма представлена на рисунке Г.18.

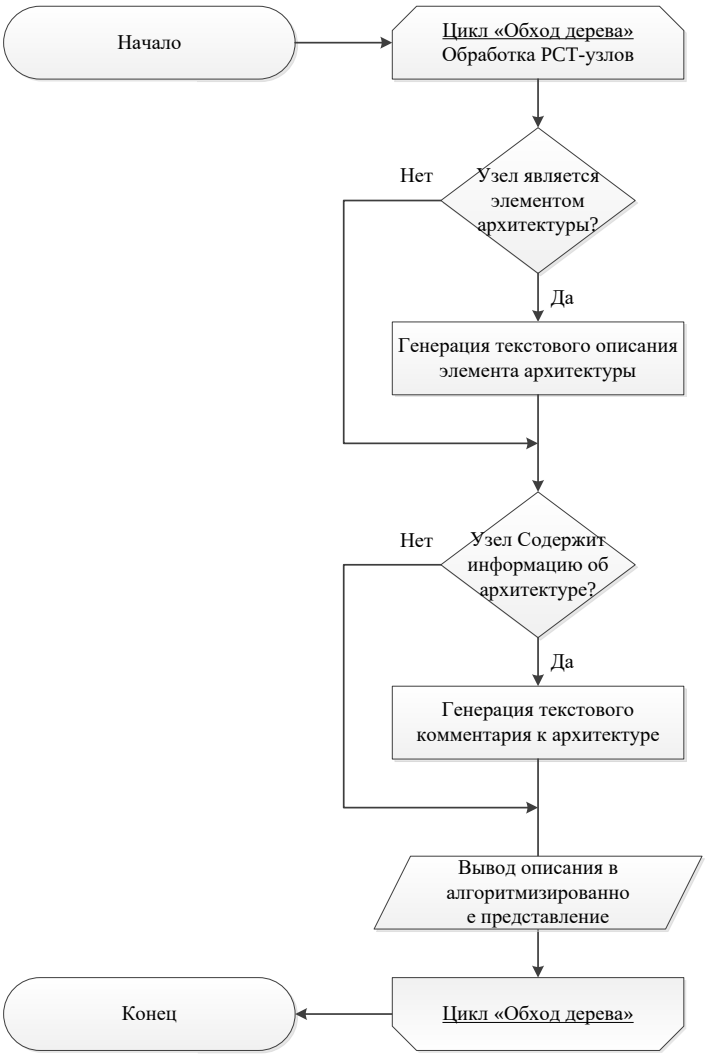

Рисунок Г.18 – Блок-схема модуля генерации текстового описания архитектуры

Производится обход всех РСТ-узлов, генерация текстового описания архитектуры в соответствии с их типами (узлы модулей) и дополнительными комментариями (интерфейсы модулей), вывод последнего в алгоритмизированное представление.

Алгоритм М_6_3. Модуль генерации текстового описания глобальных переменных

Принцип работы модуля основан на обходе поддерева псевдокода глобальных переменных, объединенных в элементы архитектуры, и генерации для каждого узла его текстового описание в алгоритмизированном. Блок-схема алгоритма представлена на рисунке Г.19.

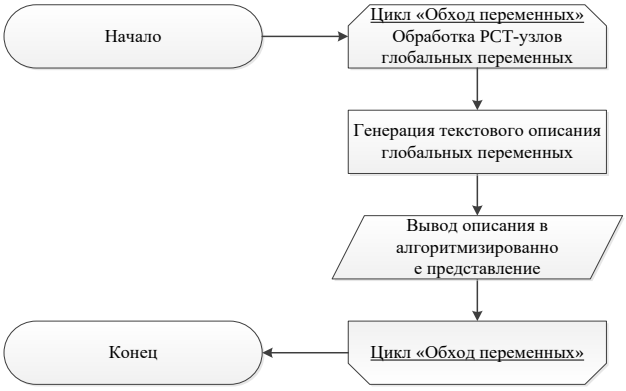

Рисунок Г.19 – Блок-схема модуля генерации текстового описания глобальных переменных

Производится обход всех PCT-узлов глобальных переменных, генерация их текстового описания и вывод последнего в алгоритмизированное представление.

Алгоритм М_6_4. Модуль генерации текстового описания подпрограмм

Принцип работы модуля основан на обходе поддерева псевдокода подпрограмм, объединенных в группирующие элементы архитектуры, и генерации для каждого узла ее текстового описание в алгоритмизированном. Подпрограмма описывается сигнатурой, блоком операций и опциональными комментариями. Блок-схема алгоритма представлена на рисунке Г.20.

Производится обход всех PCT-узлов подпрограмм, генерация текстового описания сигнатуры подпрограмм. Затем, производится рекурсивный обход PCT-узлов кода подпрограмм и генерация текстового описания согласно их типам. Для возврата из подпрограммы генерируется операция return. Для блока кода генерируются границы блока «{…}» и текстовое описание его операций. Для условного перехода генерируется выбор условия «if (…) {…} else {…}» и каждая из веток обрабатывается отдельно. Для цикла генерируется выбор post- и pre- условий «loop (…) {…} (…)», а тело цикла обрабатывается отдельно. Другие узлы (например, комментарии) генерируются согласно их представлению по умолчанию. Затем описание переносится в алгоритмизированное представление.

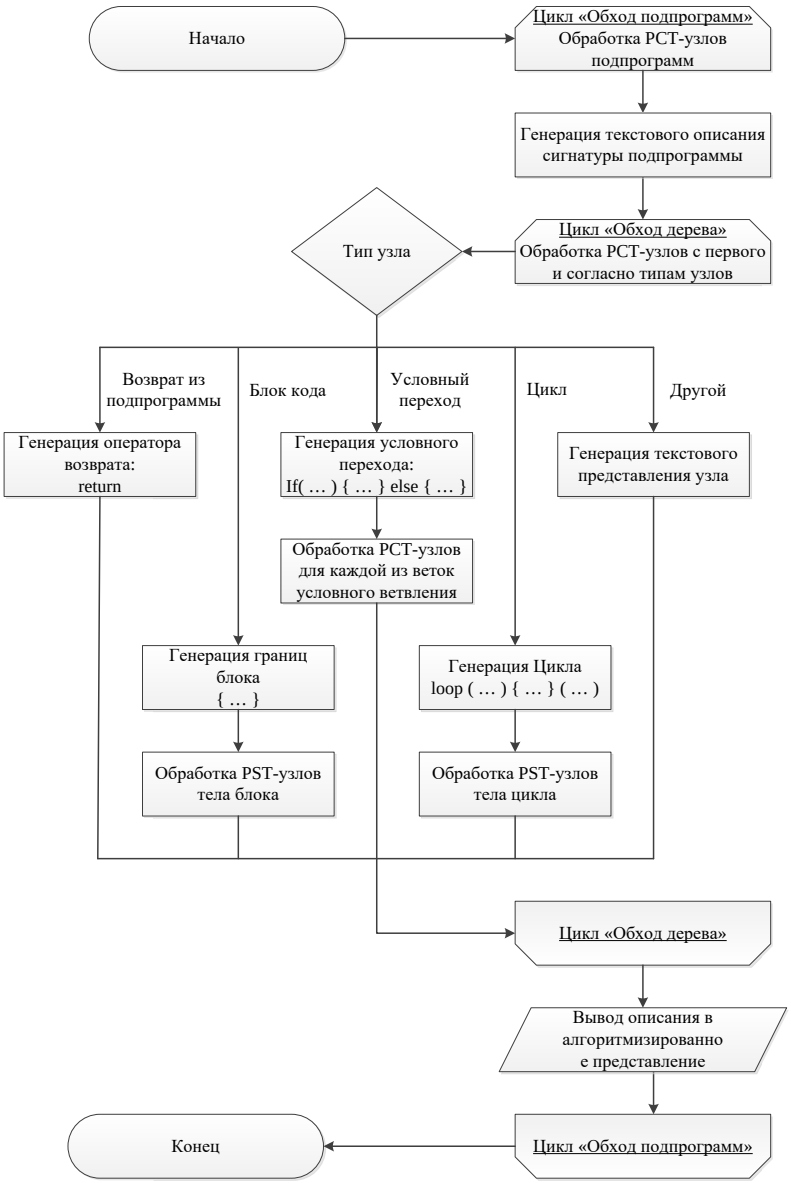

Рисунок Г.20 – Блок-схема модуля генерации текстового описания подпрограмм

Алгоритм M_6_5. Модуль генерации информации об уязвимостях

Модуль работает, как составная часть модуля генерации текстового описания подпрограмм, добавляя в описание информацию об уязвимостях, хранимую в специальных узлах псевдокода. Блок-схема алгоритма представлена на рисунке Г.21.

Производится обход всех PCT-узлов, генерация их текстового описания уязвимостей для соответствующих типов узлов и вывод последнего в алгоритмизированное представление.

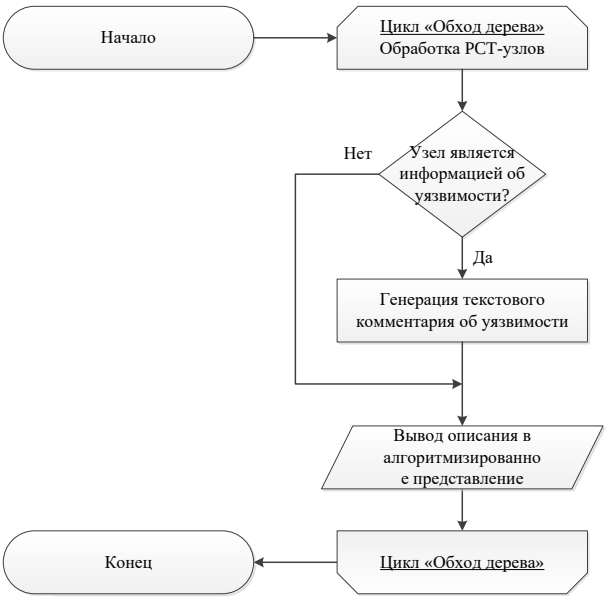

Рисунок Г.21 – Блок-схема модуля генерации информации об уязвимостях

Алгоритм M_C_1. Модуль поиска уязвимостей

Принцип работы модуля основан на эвристическом подходе и состоит из применения обобщенных шаблонов, систем критериев и оценок внутреннего представления Утилиты на предмет поиска потенциальных уязвимостей. Модуль может определить разрушение структуры ПрК и попытку записи в защищенную область памяти. В результате, будут добавлены специальные узлы в деревья и графы различных стадий, сигнализирующие о найденных уязвимостях. Блок-схема алгоритма представлена на рисунке Г.22.

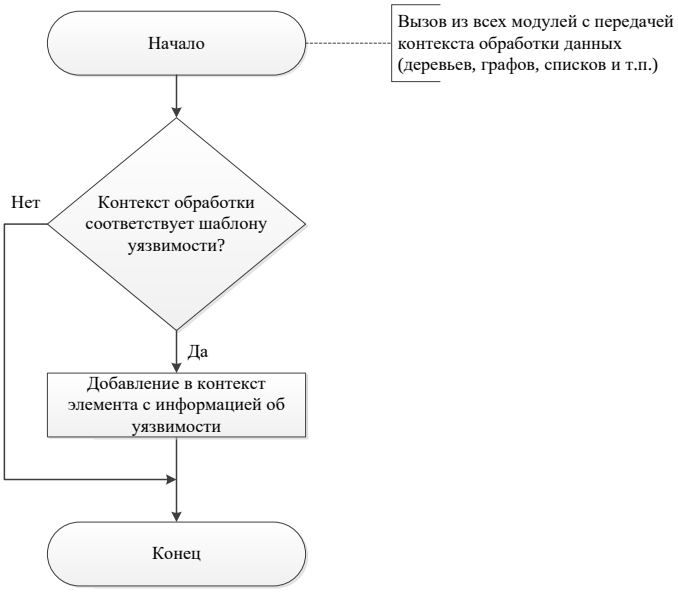

Рисунок Г.22 – Блок-схема модуля поиска уязвимостей

На каждый вызов из модулей ПС производится сравнение переданного контекста (деревьев, графов, списков и т. п.) с шаблонами уязвимостей и в случае соответствия добавление узла с информацией об уязвимости.

Алгоритм M_C_2. Модуль учета корректировок алгоритмизации

Принцип работы модуля основан на внесении поправок в работу всех других модулей на основании корректировок, заданных во входном ассемблерном коде. Модуль может указать другим точную сигнатуру подпрограммы, имя и адрес глобальной переменной и других идентификаторов, защиту от записи в область памяти (применяется при поиске уязвимостей). Блок-схема алгоритма представлена на рисунке Г.23.

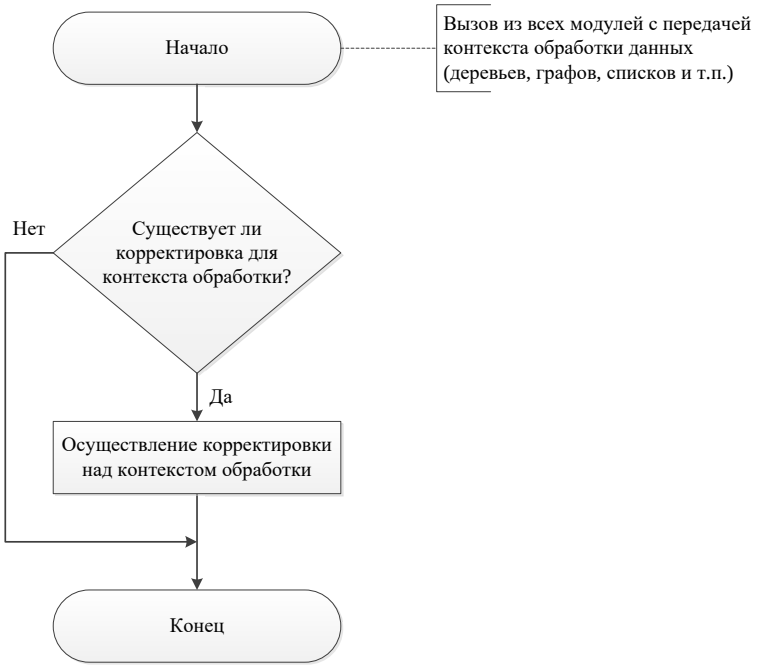

Рисунок Г.23 – Блок-схема модуля учета корректировок алгоритмизации

На каждый вызов из модулей ПС производится проверка наличия корректировки для переданного контекста (деревьев, графов, списков и т. п.) и при наличии такой производятся соответствующие действия над контекстом.

ПРИЛОЖЕНИЕ Д. Исходные данные и результаты тестирования Прототипа

Приведено описание примеров для базового тестирования функционала Прототипа в виде их ИК, АК для процессора PowerPC, блок-схем алгоритмов, командной строки запуска Прототипа и результатов его работы (включая их предварительный анализ), а также вносимых Экспертом-М корректировок.

Пример 1 (простой). Функция нахождения максимального из трех чисел

Типовая функция нахождения максимального из 3-х чисел на языке С имеет вид:

```
int max3(int x, int y, int z) {
    int m, n;
    if (x > y) {
        m = x;
    } else {
        m = y;
    }

    if (m > z) {
        n = m;
    } else {
        n = z;
    }

    return n;
}
```

АК подпрограммы для процессора PowerPC, согласно предложенному расширенному синтаксису, имеет нижеследующий вид (адреса инструкций, как указывалось, Прототипом никак не учитываются; комментарии созданы вручную и описывают аналогичные строки ИК на языке С):

```
max3(){
    0x00000001:  max3:              // { x(r3), y(r4), z(r5), m(r6), n(r7)

    0x00000002:  cmpw r3, r4        //     if (x > y)
    0x00000003:  ble label_1        //     {
    0x00000004:  mr r6, r3          //         m = x;
    0x00000005:  b label_2          //     }

    0x00000006:  label_1:           //     else {
    0x00000007:  mr r6, r4          //         m = y;
    0x00000008:  label_2:           //     }
    0x00000009:  cmpw r6, r5        //     if(m > z)
    0x0000000A:  ble label_3        //     {
```

```
    0x0000000B:  mr r7, r6          //          n = m;
    0x0000000C:  b label_4          //      }

    0x0000000D:  label_3:           //      else {
    0x0000000E:  mr r7, r5          //          n = z;
    0x0000000F:  label_4:           //      }
    0x00000010:  mr r3, r7          //      return n;

    0x00000011:  blr                // }
}
```

Алгоритм работы подпрограммы следующий. В первой конструкции IF-THEN-ELSE производится сравнение первых двух входных параметров: X и У, а максимальное из них присваивается временной переменной M – таким образом, она хранит максимальное из двух чисел. Во второй конструкции IF-THEN-ELSE производится сравнение временной переменной M и третьего входного параметра Z, а максимальное из них присваивается временной переменной N – таким образом, она хранит максимальное из трех чисел. Эта переменная и возвращается подпрограммой. Блок-схема алгоритма приведена на рисунке Д.1.

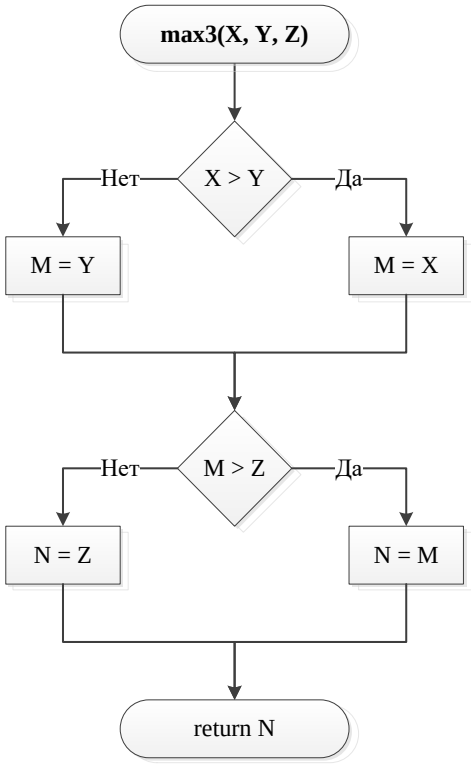

Рисунок Д.1 – Блок-схема алгоритма для функции max3()

Запуск Прототипа осуществляется с помощью командной строки

```
> utility.exe <max3.asm –s,
```

где файл max3.asm содержит приведенный АК.

В результате Прототип выведет в выходной поток восстановленный алгоритм функции «max3», а именно:

```
(r3) max3(w1_, w2_, w3_) {
    if (w1_ <= w2_)?true {
        w1_ = w2_;
    }
    if (w1_ <= w3_)?true {
        w1_ = w3_;
    }
    return (w1_);
}
```

Как хорошо видно, восстановленный алгоритм не только аналогичен исходному, но и имеет более компактный вид (9 строк против 16-ти). Также Прототип абсолютно корректно определил все входные и выходные параметры, как и условия для ветвлений.

Для удобства восприятия, Эксперт-М может внести простейшие корректировки алгоритмизации относительно имен входных параметров функции путем явного задания сигнатуры подпрограммы в АК следующим образом:

```
max3(X:r3, Y:r4, Z:r5){
    0x00000001:  max3:              // { x(r3), y(r4), z(r5), m(r6), n(r7)
    ...
    0x00000011:  blr                // }
}
```

что при повторном запуске Прототипа приведет к генерации следующего восстановленного алгоритма:

```
(r3) max3(X, Y, Z) {
  if (X <= Y)?true {
    X = Y;
  }
  if (X <= Z)?true {
    X = Z;
  }
  return (X);
}
```

Пример 2 (сложный). Функция алгоритма определения старшего бита числа

Типовая функция нахождения старшего значимого (т. е. равного 1) бита на языке С имеет вид:

```
int high_bit_num(int x) {
  int d1 = 0;
  int d2 = 32;
  for (int n = 4; n >= 0; --n) {
    int m = (d1 + d2) / 2;
```

```
    if ((x >> m) == 0)
      d2 = m;
    else
      d1 = m;
  }
  return d1;
}
```

Подпрограмма корректно работает для любого 32-х битного числа, кроме 0, и возвращает номер старшего бита, начиная с 0-го. Так, например, для числа 0x5, соответствующего последовательности бит с ненулевым окончанием b00000101, подпрограмма вернет «high_bit_num(0x5) = 2», указывая на 2-ю позицию старшего бита; для 0x38 (b00111000) вернет «high_bit_num(0x38) = 5».

АК подпрограммы для процессора PowerPC, согласно предложенному расширенному синтаксису, имеет следующий вид (комментарии сгенерированы в IDA Pro и содержат описания инструкций процессора).

```
bit_high(){
    0x00000000: bit_high:

    0x00000000: mr r8, r3              // Move Register
    0x00000001: li r3, 0               // Load Immediate
    0x00000002: li r10, 32             // Load Immediate
    0x00000003: li r11, 4              // Load Immediate

    0x00000004: loc_81CA2A0C:
    0x00000005: add r0, r3, r10         // Add
    0x00000006: srwi r9, r0, 1          // Shift Right Immediate
    0x00000007: srw. r0, r8, r9         // Shift Right Word
    0x00000008: beq loc_81CA2A24        // Branch if equal
    0x00000009: mr r3, r9               // Move Register
    0x00000009: b loc_81CA2A28          // Branch

    0x0000000A: loc_81CA2A24:
    0x0000000A: mr r10, r9              // Move Register

    0x0000000B: loc_81CA2A28:
    0x0000000B: addic. r11, r11, -1     // Add Immediate Carrying
    0x0000000C: bge loc_81CA2A0C        // Branch if greater than or equal

    0x0000000D: blr                     // Branch unconditionally
}
```

Подпрограмма работает следующим образом. Она осуществляет поиск старшего ненулевого бита не полным перебором, а более оптимальным способом – путем деления битовой последовательности числа (в данном примере и для преобладающего большинства случаев ее размер равен 32-битам, что отражено в ин-

струкции по адресу 0x00000002) на равные части и выбора ненулевой последовательности, содержащей максимально старшие биты – классический метод деления пополам. Таким образом, алгоритм последовательно (через ненулевые половинки, заданные диапазоном бит с d1 по d2) приближается к биту в последовательности (или точке на отрезке, следуя аналогии), ненулевому и максимально-старшему. Алгоритм гарантированно находит старший бит числа за 5 итераций, что отражено в инструкции по адресу 0x00000003 и цикле по диапазону адресов 0x00000004-0x0000000C. Блок-схема алгоритма приведена на рисунке Д.2.

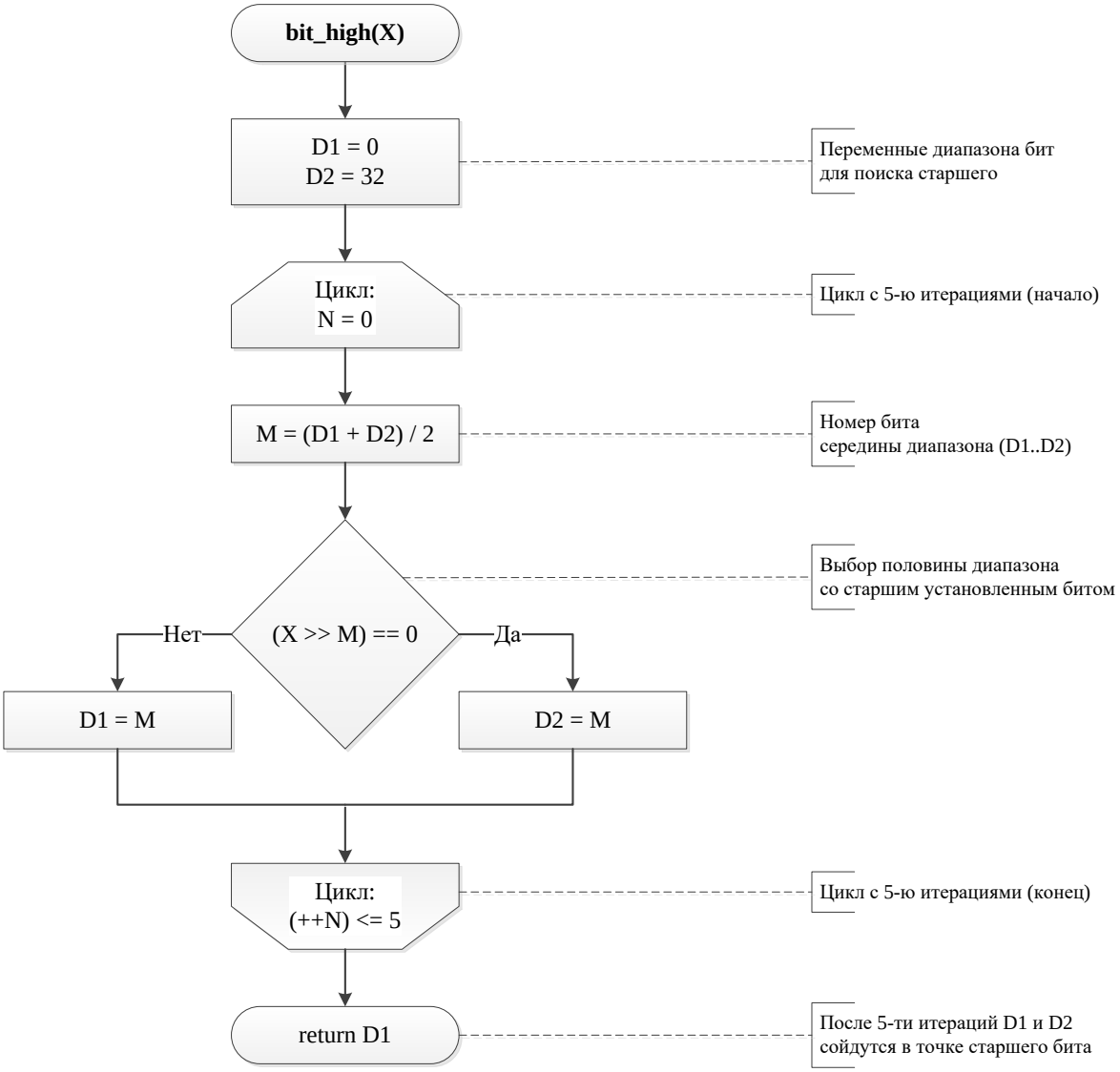

Рисунок Д.2 – Блок-схема алгоритма для функции bit_high()

Отметим тот факт, что как сам алгоритм, так и его АК (для процессора PowerPC) взят из реального образа ТКУ (а именно, Cisco 1760), поэтому успешность применения на нем Прототипа будет являться проверкой работы последнего в «боевых условиях». Также очевидно, что понять назначение такой подпрограммы, имея лишь АК без его алгоритмизации крайне затруднено вследствие сложности машинного Представления алгоритма.

Запуск Прототипа осуществляется с помощью командной строки

```
> utility.exe <bit_high.asm –s,
```

где файл bit_high.asm содержит приведенный АК.

В результате Прототип выведет в выходной поток восстановленный алгоритм функции «bit_high», а именно:

```
(r3, r10) bit_high(w1_) {
    _w2 = 0;
    _w3 = 32;
    @w4 = 4;
    loop{
        @w5 = _w3 + _w2;
        @w5 >>= 1;
        if ((w1_ >> @w5) == 0)?true {
            _w3 = @w5;
        } else {
            _w2 = @w5;
        }
        --@w4;
    }(@w4 >= 0)?true;
    return (_w2, _w3);
}
```

При рассмотрении восстановленного алгоритма и сравнении его с ИК подпрограммы можно отметить следующее. Во-первых, общий вид алгоритма подобен исходному. Во-вторых, Прототип определил подпрограмму, как возвращающую 2 параметра (_W2 в регистре R3 и _W3 в регистре R10) вместо одного; однако это является не ошибкой Прототипа, а неоднозначностью определения возвращаемых параметров – поскольку оба варианта имеют право на существование. В-третьих, Прототип корректно определил временные переменные (@W4 и @W5), несмотря на то, что в АК одна из них размещалась на разных регистрах процессора.

Для удобства восприятия, Эксперт-М может внести корректировки алгоритмизации относительно имени входного параметра функции и количества вы-

ходных путем явного задания сигнатуры подпрограммы в АК следующим образом:

```
(r3) bit_high(X:r3){
    0x00000000: bit_high:
    ...
    0x0000000D: blr                     // Branch unconditionally,
}
```

что при повторном запуске Прототипа приведет к генерации следующего восстановленного алгоритма:

```
(r3) bit_high(X) {
    _w2 = 0;
    @w3 = 32;
    @w4 = 4;
    loop{
        @w5 = @w3 + _w2;
        @w5 >>= 1;
        if ((X >> @w5) == 0)?true {
            @w3 = @w5;
        } else {
            _w2 = @w5;
        }
        --@w4;
    }(@w4 >= 0)?true;
    return (_w2);
}
```

Такой вид алгоритма практически полностью соответствует исходному и хорошо подходит для анализа Экспертом-М.

Пример 3 (специализированный). Функция с внедренной закладкой

Условная функция проверки корректности пароля (передаваемого в первом параметре) путем его сравнения с заданным (передаваемом во втором параметре) имеет вид:

```
#define TRUE 1
#define FALSE 0
int check_password(int pw, int pw_ok) {
    int ok;
    if (pw == pw_ok) {
        ok = TRUE;
    } else {
        ok = FALSE;
    }

    return ok;
}
```

Для простоты считается, что пароли заданы идентификаторами (а, например, не строками).

АК подпрограммы для процессора PowerPC, согласно предложенному расширенному синтаксису, с указанием корректировок относительно полной сигнатуры подпрограммы имеет следующий вид.

```
(r3) check_password(pw:r3, pw_ok:r4)  {
    0x00000001:  check_password: // { pw(r3), pw_ok(r4), ok(r5)

    0x00000002:  cmpw r3, r4     //   if(pw = pw_ok)
    0x00000003:  beq label_1     //   {
    0x00000004:  li r5, 0x1      //     ok = TRUE;
    0x00000005:  b label_2       //   }

    0x00000006:  label_1:        //   else {
    0x00000007:  li r5, 0x0      //     ok  = FALSE;

    0x00000008:  label_2:        //   }
    0x00000009:  mr r3, r5       //   return ok;

    0x0000000A:  blr             // }
}
```

Алгоритм работы подпрограммы состоит из проверки входного тестируемого пароля с входным корректным; в случае совпадения паролей подпрограмма возвращает 0x1 (что соответствует булевому TRUE), иначе 0x0 (булевый FALSE). Блок-схема алгоритма приведена на рисунке Д.3.

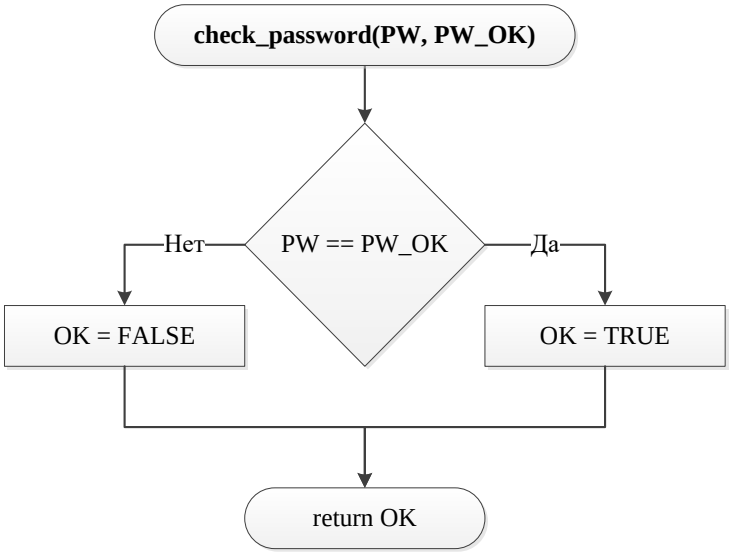

Рисунок Д.3 – Блок-схема алгоритма для функции check_password()

Применение Прототипа для примера восстановит следующий алгоритм:

```
(r3) check_password(pw, pw_ok) {
    if (pw == pw_ok)?true {
        _w3 = 0x0;
    } else {
        _w3 = 0x1;
    }
    return (_w3);
}
```

что является вполне закономерным.

Однако, поскольку подпрограмма проверки пароля относится к разряду критичных с точки зрения безопасности, ее код будет подвержен атакам в первую очередь. Так, злоумышленник может внедрить программную закладку непосредственно в данную функцию. При этом целью закладки будет модификация логики работы подпрограммы таким образом, чтобы она для любых передаваемых паролей выдавала соответствие его заданному – т. е. функция «check_password» всегда будет возвращать 0x1 (TRUE). Таким образом, модифицированный код будет содержать СУ, которая без участия Эксперта-М не может быть гарантированно обнаружена никакими автоматическими средствами поиска уязвимостей. Наиболее простейшим и прямолинейным способом такого внедрения может быть замена первых инструкций подпрограммы:

```
0x00000002:  cmpw r3, r4     //   if(pw = pw_ok)
0x00000003:  beq label_1     //   {
```

на следующие (собственно, и являющиеся кодом программной закладки):

```
0x00000002:  li r3, 0x1      //   r = true;
0x00000003:  blr             //   return r;
```

всегда возвращающие из подпрограммы результат 0x1 (TRUE).

Таким образом, код подпрограммы после модификации злоумышленником (т. е. с программной закладкой) будет следующим:

```
(r3) check_password(pw:r3, pw_ok:r4)  {
    0x00000001:  check_password: // { pw(r3), pw_ok(r4), ok(r5)

//    0x00000002:  cmpw r3, r4     //   if(pw = pw_ok)
//    0x00000003:  beq label_1     //   {
//    Программная закладка по адресам 0x00000002-0x00000003
    0x00000002:  li r3, 0x1      //   r = TRUE;
    0x00000003:  blr           //   return r;

    0x00000004:  li r5, 0x1      //     ok = TRUE;
```

```
    0x00000005:  b label_2       //   }

    0x00000006:  label_1:        //   else {
    0x00000007:  li r5, 0x0      //     ok  = FALSE;

    0x00000008:  label_2:        //   }
    0x00000009:  mr r3, r5       //   return ok;

    0x0000000A:  blr             // }
}
```

Хотя логику работы подпрограммы с закладкой с трудом можно назвать алгоритмом (поскольку последний должен описывать последовательность действий для достижения результата, что нарушается наличием в подпрограмме недостижимого кода по адресам 0x00000004-0x0000000A), тем не менее, отразим факт внедрения чужеродного кода с помощью блок-схемы такого «разрушенного» алгоритма (рисунок Д.4).

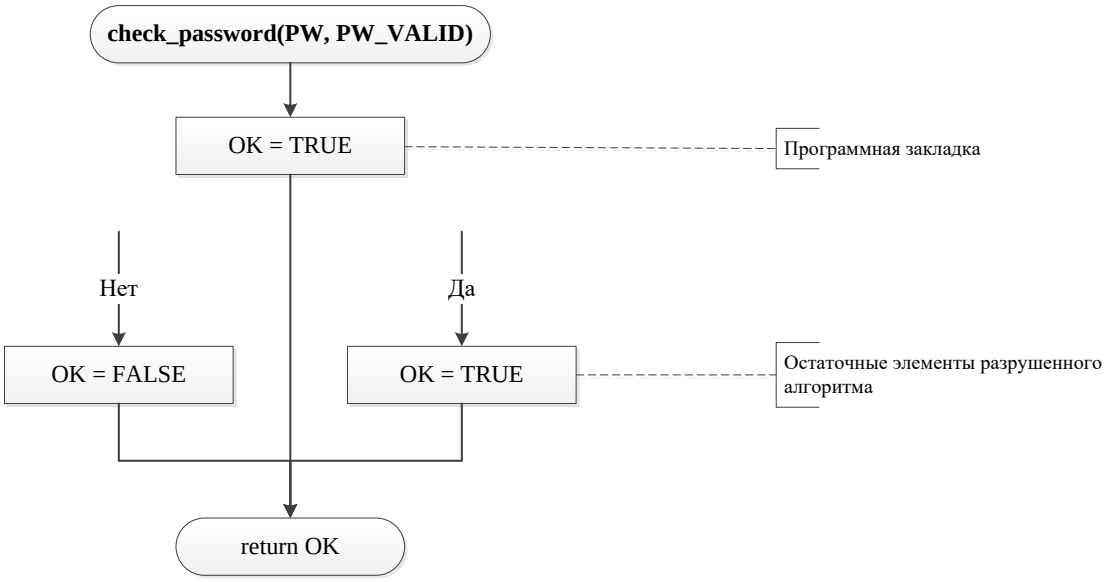

Рисунок Д.4 – Блок-схема алгоритма для функции check_password() после внедрения закладки

Следует отметить, что подобная практика внедрения закладок является крайне распространенной и даже такой «надуманный» код подпрограммы аналогичен реально-существующим, несущим угрозу информационной безопасности.

Запуск Прототипа осуществляется с помощью командной строки

```
> utility.exe <check_password.asm –s,
```

где файл check_password.asm содержит приведенный АК.

В результате Прототип выведет в выходной поток восстановленный алгоритм функции «check_password», а именно:

```
(r3) check_password(pw, pw_ok) {
    /* ATTENTION!!! Possible, destruction of the structure. */;
    return (0x1);
}
```

Также, в процессе выполнения Прототипа будет сгенерирован отладочный граф потока управления, аналогичный блок-схеме на рисунке Д.4, отражающий факт «разрушения» структуры алгоритма программы.

Таким образом, Прототип хотя и восстановил уже модифицированный злоумышленником алгоритм, тем не менее, созданный комментарий в коде «ATTENTION!!! Possible, destruction of the structure» (от рус. «ВНИМАНИЕ! Возможно, разрушение структуры») указывает Эксперту-М, что алгоритм является «разрушенным» и, следовательно, возможно было произведено внедрение закладки.

ПРИЛОЖЕНИЕ Е. База тестов и результаты сравнения Прототипа с IDA Pro

Приведена база тестов в виде ИК и АК (для процессора PowerPC), используемая для оценки функциональности ПС алгоритмизации, а также результаты работы реализации Утилиты в виде Прототипа и плагина Hex-Rays (в продукте IDA Pro). Произведен сравнительный анализ полученных результатов.

Необходимо упомянуть особенности стиля текста алгоритмов, генерируемого Прототипом. Во-первых, используется следующее обозначение имен переменных: префикс «_» указывает на то, что переменная является входным параметром подпрограммы, постфикс «_» что выходным, а префикс «@» применяется для локальных переменных; переменные без этих обозначений являются глобальными. Во-вторых, в сигнатуре перед именем функций в скобках указывается список регистров, через которые функция возвращает значения. И, в-третьих, в декларации глобальной переменной после символа «&» может быть указан абсолютный адрес ее размещения.

Исходные данные и результаты тестирования для наглядности и удобства сравнения представим в табличной форме, где последовательно (слева-направо и сверху-вниз) размещена следующая информация: прообраз ИК теста; аналогичный ему АК, используемый непосредственно для тестирования; результат работы Прототипа; результат работы IDA+Hex-Rays.

Тест 1. Выделение управляющих структур – условных ветвлений

Тест состоит из функции, возвращающей различные значения в соответствии с равенством ее 3-х параметров: 111 если параметры равны, 110 если равны первые два параметра, 011 если равны последние два и 101 если равны первый и последний (см. таблицу Е.1).

Как хорошо видно из таблицы, результат работы IDA+Hex-Rays получился более воспринимаемый человеком за счёт использования множества точек выхода (оператор «return»).

Таблица Е.1 – ИК, АК и алгоритмизированный код для Теста 1

Прообраз ИК	АК для тестирования	Результат работы Прототипа	Результат работы IDA+Hex-Rays

int test1(int x, int y, int z){
  if (x == y && y == z)
  return 111;
  if (x == y)
  return 110;
  if (y == z)
  return 11;
  if (x == z)
  return 101;
  return 0;
}
	test1(){
	0x00000000: test1:
	0x00000000: cmpw r3, r4
	0x00000004: bne loc_18
	0x00000008: cmpw r4, r5
	0x0000000C: bne loc_18
	0x00000010: li r3, 0x6F
	0x00000014: blr
	0x00000018: loc_18:
	0x00000018: cmpw r3, r4
	0x0000001C: bne loc_28
	0x00000020: li r3, 0x6E
	0x00000024: blr
	0x00000028: loc_28:
	0x00000028: cmpw r3, r5
	0x0000002C: bne loc_38
	0x00000030: li r3, 0xB
	0x00000034: blr
	0x00000038: loc_38:
	0x00000038: cmpw r3, r5
	0x0000003C: bne loc_48
	0x00000040: li r3, 0x65
	0x00000044: blr
	0x00000048: loc_48:
	0x00000048: li r3, 0
	0x0000004C: blr
	}
		(r3) test1(w1_, w2_, w3_) {
		  if (w1_ == w2_ && w2_ == w3_) {
		  _w4 = 0x6f;
		  goto $lr;
		  }
		  if (w1_ != w2_) {
		  if (w1_ != w3_) {
		  if (w1_ != w3_) {
		  _w4 = 0;
		  } else {
		  _w4 = 0x65;
		  }
		  } else {
		  _w4 = 0xb;
		  }
		  } else {
		  _w4 = 0x6e;
		  }
		  $lr:;
		  return _w4;
		}
			signed int __fastcall test1(int a1, int a2, int a3)
			{
			  if (a1 == a2 && a2 == a3)
			  return 111;
			  if (a1 == a2)
			  return 110;
			  if (a1 == a3)
			  return 11;
			  if (a1 == a3)
			  return 101;
			  return 0;
			}

Прототип же, на первый взгляд, восстановил алгоритм в более «сложной» форме – с использованием единственной точки выхода, что привело к наличию оператора «goto» и каскаду вложенных условий. Причина такого поведения Прототипа заключается в его стремлении к соблюдению процедурно-структурной парадигмы для получаемых алгоритмов: деление на подпрограммы, отсутствие безусловных переходов, один выход из подпрограммы – как следствие, большая

иерархичность блоков операций. Такая жесткая форма позволяет более успешно, как применять автоматизированные методики анализа алгоритмов, так и осуществлять поиск уязвимостей. Например, обязательное наличие лишь одной точки выхода будет гарантированно означать невозможность завершения работы подпрограммы до конца ее тела в результате логической бомбы, что увеличит шансы на обнаружение последней Экспертом-М.

Впрочем, целесообразно может быть добавление в Прототип дополнительных возможностей управления алгоритмизацией, таких, как поддержка нескольких точек выхода из подпрограммы, менее строгое требование отсутствие «goto», максимальная глубина вложенностей условий и др.

Тест 2. Выделение управляющих структур – циклов

Тест состоит из функции, возвращающей факториал числа, переданного единственным параметром (см. таблицу Е.2).

Таблица Е.2 – ИК, АК и алгоритмизированный код для Теста 2

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

`int test2(int x){`
`  int t = 1;`
`  for(int n = 1; n <= x; ++n)`
`  t *= n;`
`  return t;`
`}`
	`test2(){`
	`0x00000000: test2:`
	`0x00000000: li r4, 1`
	`0x00000004: li r5, 1`
	`0x00000008: loc_8:`
	`0x00000008: cmpw r5, r3`
	`0x0000000C: bgt loc_1C`
	`0x00000010: mullw r4, r4, r5`
	`0x00000014: addi r5, r5, 1`
	`0x00000018: b loc_8`
	`0x0000001C: loc_1C:`
	`0x0000001C: mr r3, r4`
	`0x00000020: blr`
	`}`
		`(r3) test2(w1_) {`
		`  _w2 = 1;`
		`  @w3 = 1;`
		`  loop (@w3 <= w1_)`
		`{`
		`  _w2 *= @w3;`
		`  ++@w3;`
		`  };`
		`  return _w2;`
		`}`
			`signed int __fastcall test2(signed int a1)`
			`{`
			`  signed int v1; // r4@1`
			`  signed int i; // r5@1`
			
			`  v1 = 1;`
			`  for ( i = 1; i <= a1; ++i )`
			`  v1 *= i;`
			`  return v1;`
			`}`

Как хорошо видно из таблицы, восстановленные алгоритмы подобны с единственным исключением: IDA+Hex-Rays восстановила цикл в единой форме –

с инициализацией переменной, проверкой ее значения и инкрементированием («for (i = 1; i <= a1; ++i)»); Прототип же для этого воспользовался 3-мя отдельными операциями («@w3 = 1», «loop (@w3 <= w1_)», «++@w3»). В данном примере возможности Прототипа уступают имеющимся в распоряжении IDA+Hex-Rays; тем не менее, их реализация в базовом варианте является чисто инженерной задачей кодирования.

Тест 3. Определение сигнатуры подпрограмм

Тест состоит из функции, вычисляющей значение многочлена 3-й степени, где переменная задается 1-м параметром, а коэффициенты остальными 4-мя (см. таблицу Е.3).

Таблица Е.3 – ИК, АК и алгоритмизированный код для Теста 3

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

`int test3(int x, int a, int b, int c, int d){`
`  int t1 = a * x * x * x;`
`  int t2 = b * x * x;`
`  int t3 = c * x;`
`  int t4 = d;`
`  return t1 + t2 + t3 + t4;`
`}`
	`test3(){`
	`0x00000000: test3:`
	`0x00000000: mullw r8, r4, r3`
	`0x00000004: mullw r8, r8, r3`
	`0x00000008: mullw r8, r8, r3`
	`0x0000000C: mullw r9, r5, r3`
	`0x00000010: mullw r9, r9, r3`
	`0x00000014: mullw r10, r6, r3`
	`0x00000018: mr r11, r7`
	`0x0000001C: mr r3, r8`
	`0x00000020: add r3, r3, r9`
	`0x00000024: add r3, r3, r10`
	`0x00000028: add r3, r3, r11`
	`0x0000002C: blr`
	`}`
		`(r3) test3(w1_, w2_, w3_, w4_, w5_) {`
		`  return (w2_ * w1_**3) + (w3_ * w1_**2) + (w4_ * w1_) + w5_;`
		`}`
			`int __fastcall test3(int a1, int a2, int a3, int a4, int a5)`
			`{`
			`  return a2 * a1 * a1 * a1 + a3 * a1 * a1 + a4 * a1 + a5;`
			`}`

Оба программных средства определили сигнатуру подпрограммы одинаковым образом и в соответствии с ИК: 5 входных параметров и 1 выходной. Впрочем, необходимо отметить, что эффект от лаконизации кода, применяемой в Про-

тотипе, позволил получить более короткую и воспринимаемую запись алгоритма – с помощью оператора возведения в степень «**» (более подробное описание будет в Тесте 8).

Тест 4. Поддержка сложного типа – структуры

Тест состоит из функции, возвращающей переменную-структуру, поля которой заполнены результатами операций сложения, вычитания, умножения и деления над двумя входными параметрами (см. таблицу Е.4).

Таблица Е.4 – ИК, АК и алгоритмизированный код для Теста 4

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

typedef struct {
int add;
int sub;
int mul;
int div;
} OpRes;
OpRes test4(int x, int y){
OpRes res;
res.add = x + y;
res.sub = x - y;
res.mul = x * y;
res.div = x / y;
return res;
}
	test4(){
	0x00000000: test4:
	0x00000000: add r7, r3, r4
	0x00000004: subf r8, r4, r3
	0x00000008: mullw r5, r3, r4
	0x0000000C: divw r6, r3, r4
	0x00000010: mr r3, r7
	0x00000014: mr r4, r8
	0x00000018: blr
	}
		(r3, r4, r5, r6) test4(w1_, w2_) {
		return (w1_ + w2_, w1_ - w2_, w1_ * w2_, w1_ / w2_);
		}
			int __fastcall test4(int a1, int a2)
			{
			return a1 + a2;
			}

Как хорошо видно из таблицы, Прототип и IDA+Hex-Rays полноценно не поддерживают функции, возвращающие тип структуры – т. е. переменные, хранящиеся более чем в одном регистре. Однако в результате работы IDA+Hex-Rays алгоритм восстановлен абсолютно некорректно, поскольку согласно его логике функция вернет лишь значение первого члена структуры – «OpRes::add» (через регистр R3). При этом ситуация, когда возвращаемая структура расположена на нескольких регистрах, для современных процессоров и компиляторов является более чем реалистичной. Прототип же хоть и не указал явно в сигнатуре и для переменных тип структуры, тем не менее, алгоритм восстановлен правильно – функция возвращает последовательность из 4-х значений, согласно всем членам

структуры OpRes. По сути, тип структуры физически соответствует упорядоченной последовательность переменных и такую ее интерпретацию можно считать допустимой. Также, отсутствие поддержки типов переменных в Прототипе (за небольшим исключением для простых случаев) соответствует одному из принципов Утилиты – избавлению описания алгоритмов от «семантического мусора», излишнего для понимания сути работы алгоритма.

Тест 5. Работа с глобальными переменными

Тест состоит из 3-х глобальных переменных по заданным адресам и функции, возвращающей их попарное перемножение с входными параметрами (см. таблицу Е.5).

Таблица Е.5 – ИК, АК и алгоритмизированный код для Теста 5

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

int x, y, z;
int test5(int a, int b, int c) {
  int t = a*x + b*y + c*z;
  return t;
}
	var x & 0x0000003C;
	var y & 0x00000040;
	var z & 0x00000044;
	test5(){
	0x00000000: test5:
	0x00000000: lis r6, x@h
	0x00000004: lwz r6, x@l(r6)
	0x00000008: mullw r3, r3, r6
	0x0000000C: lis r7, y@h
	0x00000010: lwz r7, y@l(r7)
	0x00000014: mullw r4, r4, r7
	0x00000018: lis r8, z@h
	0x0000001C: lwz r8, z@l(r8)
	0x00000020: mullw r5, r5, r8
	0x00000024: mr r9, r3
	0x00000028: add r9, r9, r4
	0x0000002C: add r9, r9, r5
	0x00000030: mr r3, r9
	0x00000034: blr
	}
		var x & 0x3c;
		var y & 0x40;
		var z & 0x44;
		(r3) test5(w1_, w2_, w3_) {
		  return (w1_ * x) + (w2_ * y) + (w3_ * z);
		}
			int __fastcall test5(int a1, int a2, int a3)
			{
			  return a1 * x + a2 * y + a3 * z;
			}

Как хорошо видно, Прототип, помимо алгоритма функции, сгенерировал перечень глобальных переменных с их абсолютными адресами – это может быть полезно при анализе их назначения, основываясь на расположении в памяти. В остальном восстановленные алгоритмы можно считать идентичными.

Тест 6. Восстановление нетривиального алгоритма

Тест состоит из функции нахождения старшего ненулевого бита числа «быстрым» алгоритмом: путем итеративного деления отрезка последовательности бит на части с выбором старшей ненулевой (см. таблицу Е.6).

Таблица Е.6 – ИК, АК и алгоритмизированный код для Теста 6

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

`int test6(int x) {`
`  int d1 = 0;`
`  int d2 = 32;`
`  for (int n = 4; n >= 0; --n) {`
`  int m = (d1 + d2) / 2;`
`  if ((x >> m) == 0)`
`  d2 = m;`
`  else`
`  d1 = m;`
`  }`
`  return d1;`
`}`
	`test6(){`
	`0x00000000: test6:`
	`0x00000000: mr r8, r3`
	`0x00000004: li r3, 0`
	`0x00000008: li r10, 0x20`
	`0x0000000C: li r11, 4`
	`0x00000010: loc_10:`
	`0x00000010: add r0, r3, r10`
	`0x00000014: srwi r9, r0, 1`
	`0x00000018: srw. r0, r8, r9`
	`0x0000001C: beq loc_28`
	`0x00000020: mr r3, r9`
	`0x00000024: b loc_2C`
	`0x00000028: loc_28:`
	`0x00000028: mr r10, r9`
	`0x0000002C: loc_2C:`
	`0x0000002C: addic. 11, r11, -1`
	`0x00000030: bge loc_10`
	`0x00000034: blr`
	`}`
		`(r3, r10) test6(w1_)`
		`{`
		`  _w2 = 0;`
		`  _w3 = 0x20;`
		`  @w4 = 4;`
		`  loop {`
		`  @w5 = (_w2 + _w3) / 2;`
		`  if (w1_ >> @w5 == 0) {`
		`  _w3 = @w5;`
		`  } else {`
		`  _w2 = @w5;`
		`  }`
		`  --@w4;`
		`  } (@w4 >= 0) ;`
		`  return (_w2, _w3);`
		`}`
			`unsigned int __fastcall test6(unsigned int a1)`
			`{`
			`  unsigned int v1; // r8@1`
			`  unsigned int result; // r3@1`
			`  unsigned int v3; // r10@1`
			`  signed int v4; // r11@1`
			
			`  v1 = a1;`
			`  result = 0;`
			`  v3 = 32;`
			`  v4 = 4;`
			`  do`
			`  {`
			`  if ( v1 >> ((result + v3) >> 1) )`
			`  result = (result + v3) >> 1;`
			`  else`
			`  v3 = (result + v3) >> 1;`
			`  --v4;`
			`  }`
			`  while ( v4 >= 0 );`
			`  return result;`
			`}`

Как хорошо видно из таблицы, логика и общее представление восстановленных алгоритмов обоими программными средствами полностью соответствуют заданному в ИК; тем не менее, имеются следующие отличия. Во-первых, функция, восстановленная Прототипом, возвращает значение в двух перемененных (r3 и r10), вместо одной (типа int), как в ИК. Однако это вполне логично, поскольку согласно АК ситуация, когда алгоритм функции возвращает значения в двух указанных регистрах, имеет полное право на существования. Данная ситуация разрешается после анализа алгоритма и назначения функции Экспертом-М и внесения корректировок в ассемблерный код путем явного указания ее сигнатуры («(r3) test6(w1_)»). Во-вторых, алгоритм после Прототипа имеет более оптимизированный вид из-за вынесения общего выражения «(_w2+_w3)/2», в отличие от дублируемого «(result+v3)>>1» в коде после IDA+Hex-Rays. Такая генерация неоптимального вида алгоритма конкурентом Прототипа является, скорее всего, следствием отсутствия необходимых в нем модулей. И, в-третьих, особый вид цикла в опциональной пост-префиксной форме «loop (pre-cond) { … } (post-cond);» (где условия pre-cond и post-cond являются не обязательными) позволяет единым образом создавать все возможные варианты циклов: бесконечный, с предусловием, с постусловием и т.п.. Как следствие, расширяется возможность укороченной записи итерационных частей алгоритма с минимизацией используемых для этого операций.

Тест 7. Восстановление алгоритма кода не структурной парадигмы программирования

Тест состоит из функции, алгоритм которой сознательно сконструирован с нарушением структурной парадигмы программирования – путем использования оператора «goto» для перехода из одной ветки условного перехода в другую на том же уровне иерархии. Суть работы алгоритма заключается в инкрементировании или декрементировании второго параметра на основании знака первого, который впоследствии возвращается функцией. При этом если первый параметр равен 0 или второй параметр оказывается меньше 0 или больше 10, то происходит вызов подпрограммы ошибки – «print_error()» (см. таблицу Е.7).

Таблица Е.7 – ИК, АК и алгоритмизированный код для Теста 7

Прообраз ИК	АК для тестирования	Результат работы Прототипа	Результат работы IDA+Hex-Rays

void error(void);
int test7(int x, int y) {
  if (x > 0){
  y ++;
  if(y > 10)
  goto label;
  }else if(x < 0){
  y --;
  if(y < 0)
  goto label;
  }else{
label:;
  print_error();
  }
  return y;
}
	test7(){
	0x00000000: test7:
	0x00000000: cmpwi r3, 0
	0x00000004: ble loc_18
	0x00000008: addi r4, r4, 1
	0x0000000C: cmpwi r4, 0xA
	0x00000010: bgt loc_30
	0x00000014: b loc_34
	0x00000018: loc_18:
	0x00000018: cmpwi r3, 0
	0x0000001C: bge loc_30
	0x00000020: addi r4, r4, -1
	0x00000024: cmpwi r4, 0
	0x00000028: blt loc_30
	0x0000002C: b loc_34
	0x00000030: loc_30:
	0x00000030: bl print_error
	0x00000034: loc_34:
	0x00000034: mr r3, r4
	0x00000038: blr
	}
		(r3) test7(w1_, w2_)
		{
		  if (w1_ <= 0) {
		  if (w1_ < 0) {
		  --w2_;
		  if (w2_ >= 0)
		{
		  goto loc_34;
		  }
		  }
		  } else {
		  ++w2_;
		  if (w2_ <= 0xa)
		{
		  goto loc_34;
		  }
		  }
		  () =
		print_error();
		  loc_34:;
		  return w2_;
		}
			int __fastcall
			test7(int a1, int a2)
			{
			  int v2; // r4@2
			
			  if (a1 > 0)
			  {
			  v2 = a2 + 1;
			  if (v2 <= 10)
			  return v2;
			  goto LABEL_6;
			  }
			  if (a1 >= 0 || (v2 = a2 - 1, v2 < 0))
			LABEL_6:
			  print_error();
			  return v2;
			}

Как хорошо видно из таблицы, поток управления восстановленных алгоритмов принципиально отличается от заданного с помощью ИК, который имеет симметричный вид с понятной логикой. Такое поведение оцениваемых средств алгоритмизации вполне закономерно, поскольку каждое из них в той или иной степени стремится получить структурированный код (Прототипом в большей, а IDA+Hex-Rays в меньшей степени), что естественно возможно не для всех случаев. Тем не менее, очевидна предрасположенность IDA+Hex-Rays к использованию нескольких точек выхода, что полностью исключается в работе Прототипа. Как указывалось для предыдущих тестов, такая топология алгоритма менее предпочтительна для применения в Методе. Также, восстановленный Прототипом алгоритм более симметричен по сравнению с результатом работы IDA+Hex-Rays.

Тест 8. Лаконизация алгоритма

Тест состоит из глобальной переменной и функции, которая принимает два параметра; первый частично используется для вычисления значений памяти, адрес которой хранится во втором (т. е. второй параметр имеет тип указателя). Под лаконизацией понимается улучшение описания видимой части алгоритма путем использования специальных конструкций без изменения деталей его логики (см. таблицу Е.8).

Таблица Е.8 – ИК, АК и алгоритмизированный код для Теста 8

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

`int g;  // at 0x48`
`int test8(int x,`
`unsigned char *y) {`
`  int t = 0x10;`
`  x ++;`
`  *y = 1;`
`  y[1] += 2;`
`  y[2] = t + 0x38;`
`  y[3] = x * x * x;`
`  y[4] = x & 0x4;`
`  x ++;`
`  return x;`
`}`
	`var g & 0x00000048;`
	`test8(){`
	`0x00000000: test8:`
	`0x00000000: li`
	`r5, 0x10`
	`0x00000004: addi`
	`r3, r3, 1`
	`0x00000008: li`
	`r6, 1`
	`0x0000000C: stb`
	`r6, 0(r4)`
	`0x00000010: lbz`
	`r6, 1(r4)`
	`0x00000014: addi`
	`r6, r6, 2`
	`0x00000018: stb`
	`r6, 1(r4)`
	`0x0000001C: addi`
	`r7, r5, 0x38`
	`0x00000020: stb`
	`r7, 2(r4)`
	`0x00000024: mr`
	`r8, r3`
	`0x00000028: mullw`
	`r8, r8, r3`
	`0x0000002C: mullw`
	`r8, r8, r3`
	`0x00000030: stb`
	`r8, 3(r4)`
	`0x00000034: rlwinm`
	`r9, r3, 0, 29,29`
	`0x00000038: stb`
	`r9, 4(r4)`
	`0x0000003C: addi`
	`r3, r3, 1`
	`0x00000040: blr`
	`}`
		`var g & 0x48;`
		`(r3) test8(w1_, w2_)`
		`{`
		`  ++w1_;`
		`  *w2_ = 0x1;`
		`  w2_[1] += 2;`
		`  w2_[2] = &g;`
		`  w2_[3] = w1_**3;`
		`  w2_[4] = w1_.3;`
		`  return w1_ + 1;`
		`}`
			`int __fastcall`
			`test8(int a1, _BYTE`
			`*a2)`
			`{`
			`  int v2; // r3@1`
			
			`  v2 = a1 + 1;`
			`  *a2 = 1;`
			`  a2[1] += 2;`
			`  a2[2] = 64;`
			`  a2[3] = v2 * v2 *`
			`v2;`
			`  a2[4] = a1 & 0x4;`
			`  return v2 + 1;`
			`}`

Хотя алгоритм функции восстановлен обоими ПС практически одинаково, тем не менее, его описание Прототипом за счет лаконизации можно считать более

компактным и понятным человеку. Лаконизация для данного примера, отсутствующая явно в работе IDA+Hex-Rays, достигается следующими приемами. Во-первых, для увеличения переменной на 1 используется оператор инкремента «++», широко распространенный в языке C/C++. Во-вторых, значение 0x48 заменено на оператор взятия адреса глобальной переменной «&», что может быть полезно при анализе неявных потоков данных. В-третьих, многократное умножение переменной на саму себя заменено на оператор возведения в степень «**», не стандартный для языка C/C++, но применяемый в языках Fortran, Haskell, Python, Ruby и др. И, в-четвертых, доступ к биту числа путем побитовой операции с маской «&» заменен на более логичный и наглядный оператор «.» с указанием номера бита, что аналогично доступу к члену структуры в форме «S.x».

Тест 9. Обнаружение потенциальной уязвимости

Тест состоит из функции, увеличивающей значение глобальной переменной на значение, переданное через ее параметр. Особенность программы заключается в том, что согласно предполагаемой логике работы переменная хранит секретное предустановленное значение, которое может быть только прочтено – т. е. переменная имеет условный атрибут ReadOnly. Таким образом, изменение функцией этого значения считается уязвимостью. И если предотвращение изменения значения объекта на языке C и осуществимо с помощью квалификатора «const», то в МК это практически недостижимо (см. таблицу Е.9).

Таблица Е.9 – ИК, АК и алгоритмизированный код для Теста 9

Прообраз ИК	**АК для тестирования**	**Результат работы Прототипа**	**Результат работы IDA+Hex-Rays**

`int secret_for_read; // Secret value, that is can't be rewritten`
`int test9(int x) {`
`  secret_for_read += x;`
`}`
	`var secret_for_read & 0x0000001C;`
	`test9(){`
	`0x00000000: test9:`
	`0x00000000: lis r4, se-cret_for_read@h`
	`0x00000004: lwz r5, se-cret_for_read@l(r4)`
	`0x00000008: add r3, r3, r5`
	`0x0000000C: lis r4, se-cret_for_read@h`
	`0x00000010: stw r3, se-cret_for_read@l(r4)`
	`0x00000014: blr`
	`}`
		`var secret_for_read & 0x1c;`
		`() test9(w1_) {`
		`  /* ATTENTION!!! Write to variable with readonly ac-cess. */;`
		`  secret_for_read += w1_;`
		`  return ();`
		`}`
			`int __fastcall test9(int a1)`
			`{`
			`  int result; // r3@1`
			
			`  result = a1 + se-cret_for_read;`
			`  secret_for_read = result;`
			`  return result;`
			`}`

Уязвимость в тесте, в силу описанной специфики, является субъективной (как правило, только Эксперт-М может определить данные, к которым должен осуществляться доступ только для чтения). Таким образом, никакие автоматические средства не могут ее обнаружить – для этого требуется участие человека.

IDA+Hex-Rays не имеет каких либо средства для решения такого рода задач, в частности, из-за отсутствия встроенных возможностей по поиску уязвимостей. Применение же встроенных скриптов для подобного анализа может быть не тривиальной задачей.

Прототип же предназначен именно для решения данной задачи и позволяет вносить корректировки в работу своего алгоритма непосредственно в АК (полученный из тестируемого МК).

Так, добавление в АК строк:

```
user_control {
  secret_for_read: p_readonly;
}
```

укажет Прототипу, что объект secret_for_read имеет атрибут ReadOnly и может быть только прочитан. Притом Эксперт-М может сделать такой вывод относительно атрибутов объекта на второй итерации запуска Прототипа после первоначального анализа восстановленного кода алгоритмов. Если в результате анализа МК будет обнаружено, что осуществляется изменение объекта с атрибутом ReadOnly, то Прототип сделает соответствующее предупреждение: «/* ATTENTION!!! Write to variable with readonly access. */».

Как было указано, IDA+Hex-Rays позволяет лишь производить поиск уязвимостей напрямую, поскольку не имеет соответствующих модулей и не выводит вспомогательную информацию, применимую для поиска. Использование встроенных скриптов теоретически может решить данную задачу; однако если требуемая трудоемкость для текущего теста и прогнозируется как удовлетворительная, то для других уязвимостей она может быть крайне высокой. Прототип же, за счёт возможности ручного внесения корректировок к своей работе и итеративного выполнения на второй итерации корректно определил неправомочность доступа к переменной, которая по логике работы не может быть изменена.

ПРИЛОЖЕНИЕ Ж. Свидетельство о регистрации программы для ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2013618433

Утилита восстановления алгоритмов работы машинного кода "AlgorithmRecover"

Правообладатель: ***Федеральное государственное образовательное бюджетное учреждение высшего профессионального образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А.Бонч-Бруевича» (RU)***

Автор: ***Израилов Константин Евгеньевич (RU)***

Заявка № **2013616495**

Дата поступления **23 июля 2013 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***09 сентября 2013 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Б.П. Симонов

ПРИЛОЖЕНИЕ И. Акты внедрения и реализации

ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»
(СПбГУТ)

Юридический адрес: набережная реки Мойки,
д. 61, Санкт-Петербург, 191186

Почтовый адрес: пр. Большевиков, д. 22, корп. 1,
Санкт-Петербург, 193232
Тел.(812) 3263156, Факс: (812) 3263159
E-mail: rector@sut.ru
ИНН 7808004760 КПП 784001001
ОГРН 1027809197635 ОКТМО 40909000

«УТВЕРЖДАЮ»

Первый проректор –
проректор по учебной работе,
доктор технических наук, профессор

Г.М. Машков

«14» марта 2017 г.

АКТ

внедрения научных результатов,
полученных Израиловым Константином Евгеньевичем

Комиссия в составе:

- Дукельского Константина Владимировича, кандидата технических наук, доцента, проректора по научной работе;
- Красова Андрея Владимировича, кандидата технических наук, доцента, заведующего кафедрой защищенных систем связи (ЗСС);
- Аникевич Елены Александровны, кандидата технических наук, начальника отдела организации научно-исследовательской работы и интеллектуальной собственности

составила настоящий акт о том, что научные результаты Израилова К.Е., полученные им в ходе диссертационного исследования на тему «Метод алгоритмизации машинного кода для поиска уязвимостей в телекоммуникационных устройствах», используются на кафедре ЗСС при подготовке и проведении лекционно-практических занятий, а именно:

1) Структурная модель машинного кода с уязвимостями – для направления подготовки бакалавров 10.03.01 – «Информационная безопасность» по дисциплине «Ассемблер в задачах защиты информации» и для направления подготовки магистров 10.04.01 – «Информационная безопасность» по дисциплине «Вредоносное программное обеспечение»;

2) Метод алгоритмизации машинного кода – для направления подготовки бакалавров 10.03.01 – «Информационная безопасность» по дисциплине «Реверс-инжиниринг системного программного обеспечения».

Комиссия считает, что внедрение указанных научных результатов Израилова К.Е. в образовательный процесс СПбГУТ позволило повысить качество подготовки бакалавров и магистров по направлению 10.00.00 – «Информационная безопасность».

Комиссия также установила, что Израилов К.Е. является исполнителем научно-исследовательских работ, проводимых в СПбГУТ, и указанные научные результаты использованы в соответствующих отчетных материалах, а именно:

1) Израилов К.Е. Исследование и моделирование угроз безопасности цифровой телекоммуникационной сети: отчет о НИР шифр «Цифровая угроза-2012» / М.В. Буйневич, К.Е. Израилов.– СПбГУТ, 2012. – рег. № 047-12-054. – С. 143-149, 207-215, 218-219

2) Израилов К.Е. Разработка предложений по организационно-техническому обеспечению устойчивости функционирования сетей связи, защиты сетей связи от несанкционированного доступа к ним и передаваемой посредством их информации, методов проверки и определение перечня нарушений целостности, устойчивости функционирования и безопасности единой сети электросвязи Российской Федерации: отчет о НИР / М.В. Буйневич, А.Г. Владыко, К.Е. Израилов и др.– СПбГУТ, 2012; рег. № 034-12-054.– С. 134-152.

Проректор по научной работе,
канд. техн. наук, доцент К.В. Дукельский

Заведующий кафедрой ЗСС,
канд. техн. наук, доцент А.В. Красов

Начальник отдела организации научно-исследовательской работы
и интеллектуальной собственности,
канд. техн. наук Е.А. Аникевич

АСТРОСОФТ

ООО "Астрософт"
194044, г. Санкт-Петербург,
ул. Гельсингфорсская, дом 3, корпус 11, литера Д
Тел.: (812) 494-90-90, Факс: (812) 494-90-34 (доб. 181)
www.astrosoft.ru

ИНН 7802863195 КПП 780201001

Северо-Западный банк ПАО Сбербанк
р/сч 40702810455080007094
к/сч 30101810500000000653
БИК 044030653

14.03.2017 № АСФ-46/1

на № _____ от _____

УТВЕРЖДАЮ

Генеральный директор
ООО «Астрософт»

ВАСИЛЬЕВ П.В.

АКТ

О РЕАЛИЗАЦИИ РЕЗУЛЬТАТОВ НАУЧНОЙ ДЕЯТЕЛЬНОСТИ ИЗРАИЛОВА КОНСТАНТИНА ЕВГЕНЬЕВИЧА

Комиссия в составе:

- Нанобашвили Вячеслава Арсеновича, директора департамента;
- Бойко Павла Валентиновича, руководителя проекта;
- Бабина Германа Владимировича, ведущего программиста-математика, –

составила настоящий акт в том, что результаты научной деятельности Израилова К.Е. – архитектура программного средства алгоритмизации машинного кода и комплекс научно-методических средств оценки алгоритмизации машинного кода в интересах поиска уязвимостей, – опубликованные им в работах:

- Буйневич М.В., Израилов К.Е. Автоматизированное средство алгоритмизации машинного кода телекоммуникационных устройств // Телекоммуникации. 2013. № 6. С. 2-9;
- Буйневич М.В., Израилов К.Е. Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного кода. Часть 1. Функциональная архитектура // Информационные технологии и телекоммуникации. 2016. Т. 4. № 1. С. 115-130;
- Израилов К.Е. Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного кода. Часть 2. Информационная архитектура // Информационные технологии и телекоммуникации. 2016. Т. 4. № 2. С. 86-104;
- Израилов К.Е., Васильева А.Ю., Рамазанов А.И. Укрупненная методика оценки эффективности автоматизированных средств, восстанавливающих исходный код в целях поиска уязвимостей // Вестник ИНЖЭКОНа. Серия: Технические науки. 2013. № 8 (67). С. 107-109;
- Израилов К.Е. Методика оценки эффективности средств алгоритмизации, используемых для поиска уязвимостей // Информатизация и связь. 2014. № 3. С. 39-42,

реализованы в Обществе с ограниченной ответственностью «Астрософт» – при разработке архитектуры системного ПО (компиляторов C/C++ для встроенных систем) и в методическом обеспечении лаборатории тестирования.

Директор департамента В.А. Нанобашвили

Руководитель проекта П.В. Бойко

Ведущий программист-математик, к.т.н. Г.В. Бабин