DOI: 10.15514/ISPRAS-2019-31(6)-3

Программный комплекс для выявления недекларированных возможностей в условиях отсутствия исходного кода

[1,2] *А.Б. Бугеря, ORCID: 0000-0002-9698-458X <shurabug@yandex.ru>*
[2] *В.Ю. Ефимов, ORCID: 0000-0003-3433-6787 <real@ispras.ru>*
[2] *И.И. Кулагин, ORCID: 0000-0003-2191-1578 <i.kulagin@ispras.ru>*
[2,3] *В.А. Падарян, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*
[2,3] *М.А. Соловьев, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>*
[4] *А.Ю. Тихонов, ORCID: 0000-0003-1705-5166 <fireboo@ispras.ru>*

[1] *Институт прикладной математики им. М.В. Келдыша РАН, 125047, Россия, г. Москва, Миусская пл., д. 4.*
[2] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Москва, улица Солженицына, д. 25.*
[3] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*
[4] *Московский государственный технический университет имени Н.Э. Баумана, 105005, Москва, 2-я Бауманская ул., д. 5, стр. 1*

Аннотация. Обнаружение недекларированных возможностей программного обеспечения является одной из основных задач анализа безопасности бинарного кода. Автоматизация решения этой задачи затруднена и требует участия эксперта информационной безопасности. Существующие решения ориентированы на ручную работу аналитика, автоматизация его действий не несет в себе системный характер. В случае отсутствия необходимого инструментария аналитик лишается необходимой поддержки и вынужден самостоятельно заниматься разработкой инструментов, что сильно отдаляет его от получения необходимых практических результатов. В данной работе представлен программный комплекс, решающий задачу выявления недекларированных возможностей в целом: от создания контролируемой среды выполнения до подготовки высокоуровневого описания интересующего алгоритма. Представлен пакет инструментов разработчика QEMU QDT, предлагающий поддержку жизненного цикла разработки виртуальных машин, включая вопросы специализированного тестирования и отладки. Представлено высокоуровневое иерархическое представление алгоритма программы на основе блок-схем, а также алгоритм его построения. Предложенное представление основано на гиперграфе и позволяет реализовать ручной анализ потока данных на различных уровнях детализации. В будущем разработанное представление может использоваться для реализации алгоритмов автоматического анализа. Предложен подход к повышению качества полученного представления алгоритма с помощью объединения отдельных потоков данных в один, связывающий логические модули алгоритма. Для оценки результата построения высокоуровневого представления алгоритма разработан набор тестов на основе реальных программ и модельных примеров.

Ключевые слова: анализ бинарного кода; блок-схемы алгоритмов; анализ потока данных; контролируемое выполнение; специализированные среды разработки

Для цитирования: Бугеря А.Б., Ефимов В.Ю., Кулагин И.И., Падарян В.А., Соловьев М.А., А.Ю. Тихонов. Программный комплекс для выявления недекларированных возможностей в условиях отсутствия исходного кода. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 33–64. DOI: 10.15514/ISPRAS–2019–31(6)–3

Благодарности: Работа поддержана грантом РФФИ № 16-29-09632.

A software complex for revealing malicious behavior in untrusted binary code

[1,2] *A.B. Bugerya, ORCID: 0000-0002-9698-458X <shurabug@yandex.ru>*
[2] *V.Yu Efimov, ORCID: 0000-0003-3433-6787 <real@ispras.ru>*
[2] *I.I. Kulagin, ORCID: 0000-0003-2191-1578 <i.kulagin@ispras.ru>*
[2,3] *V.A. Padaryan, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*
[2,3] *M.A. Solovev, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>*
[4] *A.Yu. Tikhonov, ORCID: 0000-0003-1705-5166 <fireboo@ispras.ru>*

[1] *Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences, Miusskaya sq., 4, Moscow, 125047, Russia*
[2] *Ivannikov Institute for System Programming of the RAS Alexander Solzhenitsyn st., 25, Moscow, 109004, Russia*
[3] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[4] *Bauman Moscow State Technical Univarsity, ul. Baumanskaya 2-ya, 5/1, Moscow, 105005, Russia*

Abstract. One of the main problem of a binary code security analysis is a revealing of malicious behavior in an untrusted program. This task is hard to automate, and it requires a participation of a cybersecurity expert. Existing solutions are aimed on the analyst manual work; automation they provide does not demonstrate a system approach. In case where needed analysis tools are absent, the analyst loses the proper support and he is forced to develop tools on one's own. This greatly slows down him from obtaining the practical results. The paper presents a software complex to solve a revealing of malicious behavior problem as a whole: from creating a controlled execution environment to man guided preparing a high-level description of an analyzed algorithm. A QEMU Developer Toolkit (QDT) is introduced, offering support for the domain specific development life cycle. QDT is especially suited for QEMU virtual machine development, including specialized testing and debugging technologies and tools. A high-level hierarchical flowchart-based representation of a program algorithm is presented, as well as an algorithm for its construction. The proposed representation is based on a hypergraph and it allows both automatic and manual data flow analysis at various detail levels. The developed representation is suitable for automatic analysis algorithms implementation. An approach to improve the quality of the resulting representation of the algorithm is proposed. The approach combines individual data streams into the one that links separate logical modules of the algorithm. A test set based on real programs and model examples has been developed to evaluate the result of constructing the proposed high-level algorithm representation.

Keywords: binary code analysis; flowcharts; data flow analysis; controlled execution; domain specific development environment.

For citation*:* Bugerya A.B., Efimov V.Yu. Kulagin I.I., Padaryan V.A., Solovev M.A., Tikhonov A.Yu. A software complex for revealing malicious behavior in untrusted binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019. pp. 33-64 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-3

Acknowledgements. The work is supported by RFBR grant # 16-29-09632.

1. Введение

Развитие информационных технологий во многом определяется запросами рынка, требующего от разработчиков программного обеспечения (ПО) новых функций и применения уже существующих технологий в новых областях. Высокий темп развития в условиях временны*х* и ресурсных ограничений привел к тому, что вопросы безопасности ПО рассматривались как второстепенные, и только последние годы они начали получать должный приоритет у передовых разработчиков.

Угрозы безопасности исходят как от деструктивного функционала, целенаправленно заложенного в программу, так и от случайных программных дефектов, эксплуатация которых приводит к аналогичным последствиям – отказам в обслуживании, порче данных, утечке конфиденциальной информации.

На стороне разработчика уже сформировался технологический инструментарий для разработки безопасного ПО. Он позволяет сокращать число ошибок, своевременно выявляя их на ранних этапах жизненного цикла, помогает прослеживать реализуемые функции на этапах разработки – от формулирования требований до поставки пользователям исполняемого кода (дистрибутива).

Принципиально сложнее положение в области аудита, когда требуется оценить безопасность ПО в отсутствии исходных текстов и документации. Такого рода ситуации возникают, когда программные системы включают в свой состав сторонние библиотеки, доступные только в виде исполняемого бинарного кода, когда анализируются проприетарные приложения, системное ПО или встроенное ПО программно-аппаратных платформ, исходные коды которых и документация не доступны из-за экспортных или каких-то других ограничений. Практика такова, что из-за высокой трудоемкости ручного, плохо автоматизированного анализа бинарного кода аудиторы вынужденно изучают только отдельные фрагменты средних и крупных программ, основываясь в своем выборе на экспертном опыте: знаниях, какие компоненты ПО наиболее критичны для безопасности, истории ранее найденных ошибок и т.п.

Крайне востребованы методы и программные средства, способные качественно изменить работу аудитора в задачах оценки безопасности бинарного кода сложных программных систем: автоматизировать решение типовых задач, позволить формализовано выразить экспертные знания об исследуемой программе для улучшения работы автоматических средств анализа, повторного использования результатов обратной инженерии, обучения новых специалистов.

На данный момент существует ряд программных систем, включая системы промышленного уровня, предлагающих набор готовых средств анализа и возможности по расширению этих средств. К ним относятся итеративный дизассемблер IDA Pro [1], средство обратной инженерии Ghidra [2], платформы анализа бинарного кода BAP [3], angr [4], Radare2 [5], среда анализа бинарного кода ТРАЛ [6] и др. В состав этих расширяемых систем изначально или силами сторонних разработчиков включаются алгоритмы распознавания высокоуровневых конструкций (программных модулей, отдельных функций, операторов и выражений), механизм символьного выполнения, анализ помеченных данных и др.

Тем не менее, применение даже таких развитых систем сталкивается с еще не решенными проблемами, затягивающими получение целевого результата: восстановленного алгоритма, оценки критичности программного дефекта, восстановления структуры программы или форматов обрабатываемых данных. Стоит выделить две причины, растягивающие работу аудитора: отсутствие возможности провести динамический анализ и непосредственно увидеть фактические данные, с которыми работает программа, и этап ручного восстановления интересующего алгоритма в виде высокоуровневой блок-схемы.

В данной работе представлены результаты, позволяющие качественно ускорить проведение обратной инженерии бинарного кода, включая этап подготовки инструментальных средств, необходимых для эффективного анализа. Во разд. 2 описывается программный комплекс, позволяющий эффективно проводить обратную инженерию ПО в условиях отсутствия исходного кода. В разд. 3 описывается инструментарий быстрой разработки контролируемых сред выполнения, которые затем будут использоваться при проведении динамического анализа. В разд. 4 рассматривается задача автоматизированного ручного анализа бинарного кода с целью высокоуровневого

описания свойств заданного алгоритма и последующего экспертного анализа свойств алгоритма, в первую очередь – в части наблюдаемых потоков данных. Формулируется обобщённая постановка задачи поиска недекларированных возможностей (НДВ) и приводится обзор соответствующих результатов мирового уровня. В следующем, пятом, разделе предлагается иерархическое высокоуровневое представление алгоритма, описаны разработанные алгоритмы построения этого представления. Раздел 6 посвящен набору тестовых программ, использовавшихся в качестве репрезентативных примеров в ходе разработки высокоуровневого представления и оценки удобства ручной работы. В седьмом, последнем разделе даются итоговые заключения.

2. Архитектура программного комплекса анализа бинарного кода

Состав комплекса технологий обусловлен различными требованиями: по расширяемости, скорости адаптации под новые программно-аппаратные платформы, по возможностям интеграции с другими средствами анализа, причем расширяемость должна обеспечиваться в двух различных направлениях: (1) возможностей комплекса должно быть достаточно для решения различных практических задач информационной безопасности, (2) должна обеспечиваться применимость комплекса к разным классам ПО (различных процессорных архитектур, различных ОС, различный уровень ПО: встроенное, системное и прикладное).

Упрощенная схема комплекса представлена на Рис. 1.

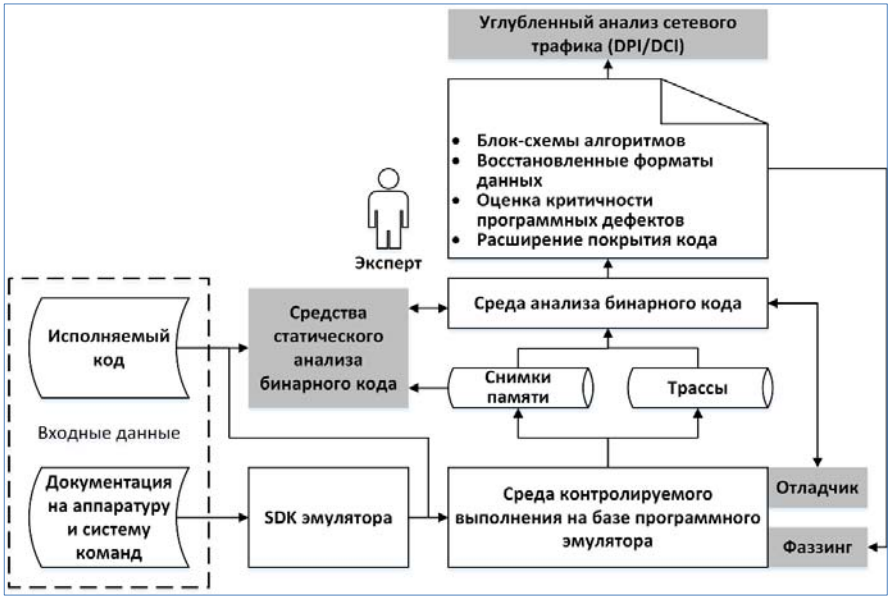

Рис. 1. Состав программного комплекса анализа бинарного кода и его связь со сторонними средствами анализа

Fig. 1. The structure of the binary code analysis software package and its relationship with third-party analysis tools

Комплекс состоит из трех основных компонент: среды анализа бинарного кода, среды контролируемого выполнения на базе программного эмулятора, где собирается информация о работе исследуемой программы, и средств быстрой разработки новых виртуальных машин.

Исследуемое ПО разворачивается в виртуальной машине (Рис. 2), эмулирующей необходимую аппаратуру, что обеспечивает возможности полносистемного анализа ПО для различного уровня. При выполнении ПО собираются трассы и снимки состояния памяти. Помимо post mortem анализа эмулятор позволяет непосредственно контролировать текущее состояние виртуальной машины, как через интерфейс отладчика, так и непосредственно встроившись в эмулятор, как это делают современные фаззеры.

Основными входными данными для среды анализа выступают как трассы выполнения уровня машинных команд, поскольку они отражают всю фактическую информацию о выполнившимся коде, так и снимки физической памяти эмулируемой системы, полученные в различные моменты времени. Снимки используются для дополнения восстанавливаемого по трассам статико-динамического представления в тех местах, где не удалось покрыть трассами код. Помимо того, снимки памяти непосредственно передаются в сторонние средства статического анализа бинарного кода, например, BINSIDE [7].

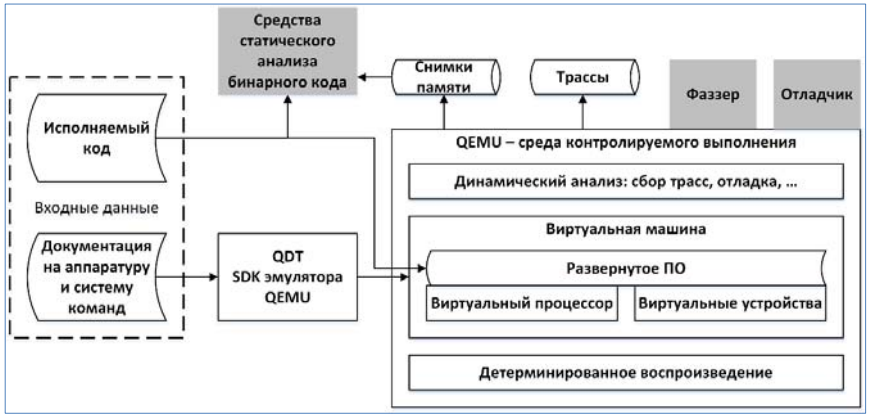

Рис. 2. Обеспечение возможностей динамического анализа средствами программной эмуляции
Fig. 2. Providing dynamic analysis capabilities with software emulation

Возможность провести динамический анализ критична для работы всего комплекса, поскольку трассы выполнения выступают «отправной точкой» для среды анализа. Для работы ОС общего назначения, таких как Windows и Linux, и их приложений на платформе x86 достаточно штатных виртуальных машин, входящих в состав эмулятора QEMU. Для некоторых аппаратных платформ удаётся подобрать либо уже готовую виртуальную машину, либо самостоятельно ее скомпоновать из штатно поддерживаемых процессоров и устройств. В остальных случаях работоспособности исследуемого кода приходится добиваться, разрабатывая необходимую виртуальную аппаратуру.

В настоящее время процессы разработки эмулятора QEMU опираются на классические инструменты, а сама разработка сопряжена с написанием большого объема шаблонного кода и времязатратным исправлением ошибок в виртуальных устройствах. Чтоб ускорить создание виртуальной машины, позволяющей выполниться объекту исследования, был разработан и реализован пакет инструментов разработчика QEMU [8] (QEMU Development Toolkit, QDT) QDT, который меняет существующие процессы разработки и автоматизирует деятельность разработчиков. QDT поддерживает разработку новых виртуальных процессоров, отдельных устройств (контроллеров, шин, периферии и т.п.), интеграцию компонент полноценной виртуальной машины.

Среда анализа ТРАЛ позволяет проводить обратную инженерию бинарного кода, восстанавливая алгоритмы в виде блок-схем, для обрабатываемых этими алгоритмами

данных – спецификацию форматов, оценивает критичность сработавших в трассах выполнения программных дефектов. Более детальное устройство среды ТРАЛ показано на Рис. 3.

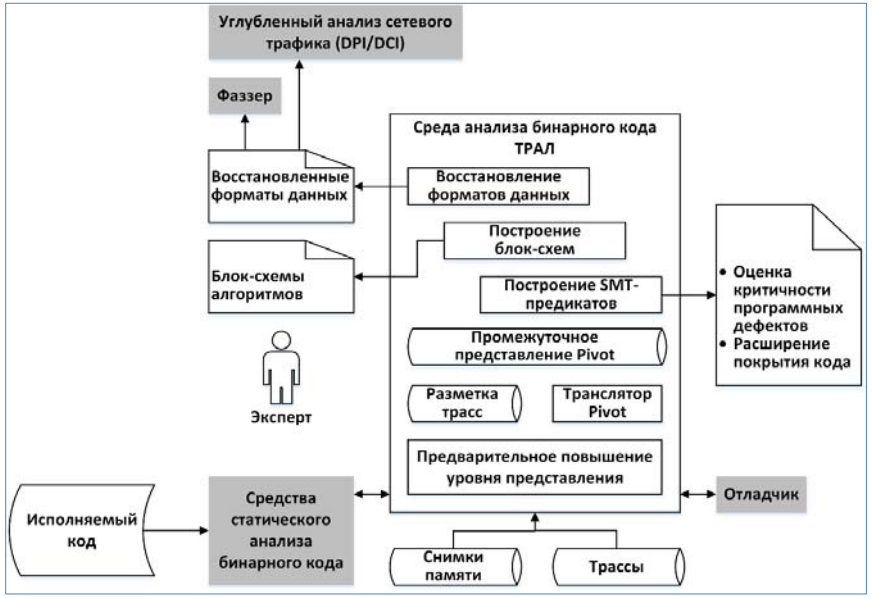

Рис. 3. Анализ трасс выполнения в среде ТРАЛ
Fig. 3. Analysis of execution traces in the TRAL environment

Работа с трассами в среде ТРАЛ начинается с повышения уровня представления, которое проводится автоматически. В определённой последовательности трассы просматривают алгоритмы анализа, строящие разметку и статико-динамическое представление программы. В трассах размечаются участки, относящиеся к различным процессам и тредам, выделяются обработчики прерываний, вызовы функций. По командам, покрытым трассами, формируется последовательность карт памяти, учитывающая модификацию кода по ходу выполнения. Восстанавливаются функции, графы потоков данных и управления, граф вызовов функций. Анализ потоков данных и управления выполняется поверх унифицированного представления декодированных инструкций, которое вырабатывают модули поддержки процессорных архитектур. Такого представления достаточно для построения блок-схем алгоритмов и восстановления форматов данных.

Если внутреннее устройство исследуемой программы частично или полностью неизвестно, нет исчерпывающего описания ее интерфейсов и форматов входных/выходных данных, то эффективность фаззинга как метода поиска ошибок падает. Становится затруднительно подготовить конфигурацию фаззинга, обеспечивающую хорошее покрытие. Некоторые интерфейсы, через которые в программу поступают входные данные, могут остаться незамеченными. Более того, для определенных классов ПО, например, встроенного, необходимо определить критерии, позволяющие трактовать некоторое состояние программы как состояние сбоя. Такие критерии предварительно вырабатываются путем ручного анализа отдельных алгоритмов, представленных в виде блок-схем. Восстановленные форматы данных транслируются в вид, пригодный для использования в генерационном фаззинге (PeachPit). Помимо того, на основе восстановленных форматов открывается возможность уточнять разборщики протоколов, в системах анализа трафика, например, ProtoSphere [9].

Во многих практических задачах исследования бинарного кода недостаточно знать только потоки данных и управления, необходимо уметь унифицировано анализировать, как именно работают с данными команды различных процессорных архитектур. Для этого операционная семантика кода транслируется в промежуточное представление Pivot. Наиболее важные сценарии его применения предполагают динамическое символьное выполнение через дальнейшую трансляцию Pivot-кода в SMT-предикаты. Первый сценарий – поиск входных данных, позволяющий активировать ни разу не сработавший в рассматриваемых трассах условный переход, второй – оценка критичности программного дефекта.

В первом сценарии ищутся входные данные, позволяющие инвертировать в заданной точке трассы условный переход. Для этого формируется предикат пути, в котором входные данные выступают в качестве свободных символьных переменных, операции над ними записываются в виде символьных выражений, а каждый сработавший условный переход добавляет в общий предикат пути свое условие. Условие перехода в заданной точке трассы записывается в общую систему ограничений в инвертированном виде. Собранный предикат пути передается в SMT-решатель, в случае нахождения его решения открывается возможность расширить покрытие кода.

Второй сценарий предполагает совмещение предиката пути с предикатом безопасности, который описывает реализацию уязвимости некоторого типа.

Возможности среды анализа не исчерпываются описанными выше сценариями, среда содержит несколько десятков модулей с различными алгоритмами анализа. Имеется API, позволяющее пользователям среды независимо разрабатывать свои алгоритмы и оформлять их в виде подключаемых модулей.

3 Процесс разработки модели вычислительной машины для эмулятора QEMU

Виртуальная машина (ВМ) в эмуляторе QEMU состоит из относительно самостоятельных программных компонентов, процесс ее разработки можно разделить на подпроцессы, схожие между собой и с объемлющим процессом.

Состав ВМ можно условно разделить на модуль поддержки системы команд процессора (транслятор команд целевой процессорной архитектуры) и комплект модулей, реализующих шины, контроллеры, устройства ввода-вывода. На практике разработка как транслятора, так и сложных устройств оказывается весьма трудозатратной, требует высокой квалификации разработчика, отладка ошибок плохо автоматизируется и, как следствие, длится неприемлемо долго.

Каждый программный компонент ВМ взаимодействует с эмулятором через внутренний API. Процесс разработки любого компонента начинается с реализации базовой версии работоспособного модуля, имеющего минимальный функционал. Затем начинается цикл итеративной разработки, состоящий из внесения необходимого функционала, тестирования, отладки и доработки. На практике цикл приостанавливается, когда все существенные ошибки исправлены, а модель обеспечивает достаточную точность эмуляции для работы исследуемого ПО.

В теории все «вложенные» подпроцессы разработки модулей могут выполняться параллельно и независимо. Но, например, тестировать реализацию системы команд или отдельное устройство значительно легче, если все компоненты ВМ собраны воедино. Т.е. часть этапов разных процессов может выполняться параллельно, а часть выгодней выполнять совместно и последовательно.

Целью работ, проводимых в ИСП РАН, являлось выделение в процессе разработки этапов, поддающихся автоматизации, и непосредственно сама автоматизация. Для разработки

всех обозначенных выше компонентов были созданы специализированные программные инструменты. Большинство из них входит в QDT.

Ниже рассматриваются процессы классической ручной разработки компонентов ВМ и возникающие при этом трудности. Приводятся способы автоматизации процессов, в конце раздела описан предлагаемый процесс разработки ВМ с использованием инструментов автоматизации.

3.1. Система команд

Работа с системой команд целевого процессора ведется фронтендом динамического двоичного транслятора TCG [10]. Фронтенд принимает на вход данные, являющиеся машинными командами, и возвращает код на промежуточном представлении TCG. Этот код изменяет состояние виртуальной машины так, как изменилось бы состояние настоящей вычислительной машины, если бы на ней был выполнен соответствующий машинный код. Поскольку для QEMU приоритетна скорость эмуляции, то некоторые, допустимые особенности работы команд не эмулируются.

Представление семантики машинной команды – программа на промежуточном представлении TCG, выполняющаяся вместо команды и совершающая над некоторой моделью вычислительной машины действия, эквивалентные действиям, выполняемым командой при выполнении на настоящей машине.

Разработчик фронтенда должен:

- реализовать парсер (декодер) машинного кода;
- выделить важные подробности работы команд;
- разработать на языке TCG семантику каждой команды;
- реализовать на языке Си программу, которая вернёт разработанную семантику;
- протестировать работу всего переднего плана.

Наконец, для того, чтобы поддержку системы команд можно было использовать, нужно создать модель процессора.

Декодер машинного кода совмещает в себе лексический и синтаксический анализ, где токенами являются последовательности битов (коды операций, операнды и т.д.), из которых собираются машинные команды. Ручное кодирование разбора двоичного представления – задача длительная, рутинная и чревата неочевидными опечатками, поиск которых может потребовать много времени.

Однако по совокупности описаний кодировок машинных команд можно автоматически сгенерировать анализатор [11]. Задать кодировки машинных команд значительно проще, чем вручную разработать анализатор. На практике эта задача сводится к переписыванию таблиц из документации.

Сгенерированный анализатор состоит из вложенных блоков `switch-case`, где выбор каждого следующего блока case происходит путём выделения подпоследовательности битов в машинном слове. В конце выполняется вызов функции, соответствующей распознанной машинной команде. В течение вызова этой функции должна быть создана программа на промежуточном представлении TCG, выполняющая действия, подразумеваемые соответствующей командой, – семантика команды. После работы генератора требуемые функции создаются с пустыми телами, куда разработчику следует вписать код, выражающий семантику команд.

Вместе с анализатором по специальному описанию генерируется модель процессора. Описание процессора включает перечень регистров, длину адреса, порядок байтов в машинном слове и др. Эти параметры также учитываются при генерации анализатора.

Описание семантики команд предоставляется производителем процессора на естественном языке, единственное исключение – формальная спецификация ISA ARM v8-

A и v8-M, разработанная компанией ARM [12, 13] При этом не существует единого формата описания, которого бы придерживались различные производители. Кроме того, в описаниях часто бывают неточности и ошибки, выявление которых требует значительных усилий человека.

Составление программы на промежуточном представлении TCG выполняется путём вызова специальных функций из API QEMU. Другими словами, разработчик пишет не непосредственно на промежуточном представлении, а на языке Си, составляя программу, которая вернёт требуемую программу.

Промежуточное представление является низкоуровневым, напоминающим язык ассемблера, поэтому даже простая программа, например, реализация команды сложения с выставлением флагов в регистре состояния процессора, получается длинной и трудночитаемой.

Для упрощения написания семантики был разработан инструмент I3S [14] – Instruction Set Semantics Specification (Описание семантики системы команд). I3S принимает описание семантики машинных команд на одноимённом языке, основывающемся на языке Си, и генерирует программу на языке Си, которая возвращает программу в промежуточном представлении TCG.

Наиболее надёжным способом тестирования транслятора является сравнение с эталоном – настоящим процессором, однако доступ к такому эталону не всегда возможен. В таких случаях вместо эталона может быть использован произвольный процессор [15], в частности – процессор на рабочем месте разработчика. В основе такой замены лежит тот факт, что избегая неопределенного и реализационно-зависимого поведения, можно составить тестовую программу на языке Си, которая будет одинаково выполняться на процессорах различной архитектуры. Выбор относительно низкоуровневого языка Си обусловлен тем, что если для некоторой, малораспространённой процессорной архитектуры доступны средства разработки, то они гарантированно будут содержать компилятор языка Си. Поиск несоответствий выполнения эталона и транслированного кода происходит на уровне терминов языка Си: значение переменной, номер строки в пути выполнения. При обнаружении несоответствия разработчик самостоятельно разбирается, чем вызвана проблема: ошибкой в машинной команде или различиями в системах команд. Поскольку тесты изначально разрабатываются с учётом возможных различий, несоответствия второго вида на практике не встречаются. Хотя у данного подхода есть множество ограничений, он позволяет покрыть значительную (более 60%) часть реализации. Кроме того, тесты обладают высокой универсальностью, и, как следствие, могут быть повторно использованы при разработке других ВМ без изменений. Подход реализован в инструменте c2t из QDT.

3.2. Периферийные устройства

Периферийные устройства в эмуляторе QEMU задаются в событийно-ориентированном стиле. Виртуальное устройство – набор функций на языке Си (обработчиков событий) и набор структур данных, описывающих состояние времени выполнения.

В число типовых событий входят:

- создание (инициализация) устройства без привязки к контексту объемлющей ВМ;
- создание экземпляра устройства (*realization*) в составе ВМ;
- сброс в начальное состояние;
- чтение и запись в регистр.

Кроме того, устройства могут издавать и принимать запросы прерываний.

Многие устройства поддерживают настройку через интерфейс командной строки. Например, можно указать файл образ для ПЗУ, реальный сетевой интерфейс для виртуального сетевого адаптера и др.

Разработчик виртуального устройства должен:

- описать состояние устройства в терминах языка Си;
- реализовать настройку устройства пользователем;
- выбрать события, нуждающиеся в обработке;
- реализовать на языке Си реакцию виртуального устройства на каждое событие;
- протестировать работу устройства.

Положение с документацией устройств аналогично схоже с описанием систем команд – если документация доступна, она подготовлена на естественном языке и не поддается автоматической формализации. Наибольшие затруднение вызывает ситуация, когда отсутствует какое-либо описание, имеется только доступ к исполняемому коду драйвера устройства в составе образа исследуемого ПО. В таких случаях в цикле итеративной разработки появляется задача обратной инженерии драйвера [16]. На каждой итерации разработчик виртуального устройства пытается продвинуться дальше по ходу выполнения драйвера до следующего сбоя в его работе. Обнаружение настоящей причины сбоя – нетривиальная, во многом творческая, задача. Затем происходит исследование причины и корректировка поведения виртуального устройства.

Реализация виртуального устройства сводится к написанию функций и структур данных на языке Си. API QEMU накладывает множество формальных требований к тому, как должен быть написан этот код. Эти требования обусловлены как языком Си, так и стилем кодирования, внутренними интерфейсами эмулятора. Например, в API определены структуры данных, куда должны быть занесены указатели на функции-обработчики событий. И эти функции должны обладать конкретной сигнатурой. Другими словами, определённая часть кода виртуального устройства изначально определена интерфейсом прикладного программирования (API) QEMU. Эту часть будем называть интерфейсной. Остальную же часть кода, реализующую индивидуальные особенности работы конкретного устройства, будем называть индивидуальной.

Создание интерфейсной части кода хорошо поддаётся автоматизации. Достаточно составить список событий и сформировать высокоуровневые данные об устройстве, например, имя устройства, чтобы стало возможно сгенерировать начальную заготовку устройства.

Заготовка устройства состоит из различных конструкций языка Си, многие из которых взаимосвязаны. Например, обработчик записи в банк регистров регистрируется в структуре данных, описывающих этот регистр. Эта структура данных, в свою очередь, регистрируется функцией инициализации устройства в служебной структуре, сообщая эмулятору, что у данного устройства есть такой-то банк регистров. Эту цепочку можно продолжать и дальше.

В общем случае зависимости между конструкциями образуют ациклический граф. Чтобы код устройства был синтаксически корректным, нужно правильно расставить идентификаторы, расположить конструкции языка в синтаксически корректном порядке, подключить нужные заголовки (причём желательно минимизировать их количество, учитывая включения одних другими) и т.д. Следует соблюсти требования к стилю написания кода, принятые в сообществе разработчиков QEMU, и учесть многие другие особенности.

При классической разработке это всё приходилось учитывать разработчику вручную. Существующая в сообществе разработчиков QEMU практика такова, что новые устройства компонуются из похожих по функционалу кусков кода готовых виртуальных

устройств. В QDT реализован инструмент, позволяющий по описанию устройства сгенерировать заготовку для устройства с соблюдением всех требований и зарегистрировать её в инфраструктуре эмулятора. Описание формализуется с помощью API QDT на языке Python, оно как правило на порядок меньше, чем получаемая из него заготовка.

Наибольшая часть индивидуальной части устройства находится в обработчиках чтения и записи в банк регистров. Принцип разбиения банков регистров на отрезки, выровненные по границе байта, т.е. регистры с индивидуальным назначением, был использован при автоматизированной генерации заготовок. Кроме как в коде обработчика доступа, регистр присутствует в состоянии времени выполнения устройства, а также упоминается в других частях кода модели. Это создаёт определённые трудности для разработчика – необходимо согласованно редактировать код в разных местах.

Регистры могут быть описаны в одном месте файла конфигурации для генератора заготовок с использованием специально разработанного расширения для API QDT [17].

В течение разработки разнообразных устройств удалось выделить некоторые часто используемые виды регистров устройств:

- регистры, доступные только для чтения;
- регистры для чтения и записи;
- регистры-пустышки;
- регистры, доступные только для записи.

Аналогичным образом может быть описан каждый бит в регистре. Указание вида регистра и его битов позволяет сгенерировать в заготовке устройства дополнительный код.

Использование формального описания регистров сокращает объём кода, написанного вручную, примерно в два раза. Однако основное преимущество в удобстве, т.к. на практике описание регистров сводится к переписыванию таблицы из документации, без необходимости редактировать код в разных местах модели устройства.

3.3. Виртуальная машина

Виртуальная машина – совокупность процессора, включая фронтенд TCG, и устройств. Внутреннее устройство ВМ в значительной степени представлено своей функцией инициализации на языке Си, которая создаёт все компоненты и связывает их между собой.

Если ВМ достаточно большая, то функция инициализации может содержать сотни строк, а при расстановке связей между устройствами вручную легко допустить ошибку. Чтобы упростить создание ВМ, в QDT был разработан графический интерфейс пользователя (ГИП), отображающий ВМ в виде схемы из связанных блоков. ГИП реализован с использованием модели ВМ на языке Python. Эта модель ВМ аналогична модели ВМ из API QEMU, но из-за языковых различий между Си и Python работать с ней проще. Вместе с ней разработан генератор, позволяющий создавать заготовку кода машины с необходимой функцией инициализации. Полученная заготовка в большинстве случаев требует минимальной доработки в отличие от заготовок для устройств, с генерации которых разработка только начинается.

3.4. Автоматизированный процесс разработки ВМ

Созданные средства автоматизации встраиваются в процесс разработки ВМ, ускоряя и/или облегчая некоторые его этапы. Рассмотрим основные этапы предложенного автоматизированного процесса и отличия от классического процесса разработки.

1. Изучение документации на систему команд и модель процессора.
2. Составление описаний кодировок команд (без семантики) и свойств процессора.

3. Автоматизированная генерация анализатора машинных команд, заготовки виртуального процессора и пустых обработчиков машинных команд.
4. Написание на языке I3S семантики машинных команд в обработчиках.
5. Автоматизированная трансляция описания семантики с языка I3S в программу на языке Си, генерирующую промежуточное представление TCG.
6. Доработка заготовки виртуального процессора.
7. Итеративная разработка:
   a) автоматизированное тестирование с использованием c2t;
   b) отладка;
   c) доработка.

В цикле итеративной разработки тестирование используется для приблизительного поиска ошибки, а отладка – для её локализации.

Доработку можно производить классическим способом, но рекомендуется вносить правки в описания из п. 2 и/или 4 с последующей повторной генерацией кода согласно п. 3 и 5, соответственно. Чтобы не затереть правки, внесённые на промежуточных шагах, рекомендуется использовать систему контроля версий. В QDT реализован вспомогательный инструмент для Git, который автоматически переносит правки на новую версию, используя механизм «rebase». Вмешательство разработчика требуется только в случае конфликтов изменений.

При классической разработке вместо этапов со 2 по 5 происходило ручное написание на языке Си как декодера команд, так и транслятора TCG; код виртуального процессора приходилось писать с нуля. Тестирование зачастую выполнялось вручную с помощью отладчика, поскольку в большинстве случаев системы тестирования, имеющиеся в открытом доступе, не были рассчитаны на требуемую систему команд. Тесты писались на языке ассемблера.

Автоматизированная разработка устройств состоит из следующих этапов.

1. Изучение документации на устройство.
2. Составление перечня событий и элементов API QEMU, которые потребуются для работы модели. Опционально подготавливается описание регистров.
3. Автоматизированная генерация заготовки устройства.
4. Реализация на языке Си индивидуальной части кода устройства.
5. Итеративная разработка: тестирование, отладка, доработка.

В случае ошибки в формальном описании устройства (п. 2) можно вносить правки в заготовку, как это происходит всегда при классическом процессе. Однако часто бывает быстрее внести изменения в описание и повторить генерацию (п. 3). При повторной генерации текущая реализация устройства затирается, поэтому важно использовать систему контроля версий. В состав QDT включен инструмент для работы с Git, позволяющий автоматизированно сгенерировать новую версию и перенести наработки (п. 4 и ниже) на новую версию. Задача разработчика в этом случае – только разрешение конфликтов.

Методика тестирования устройства определяется видом этого устройства. Например, для тестирования сетевого адаптера можно подключить его к виртуальному адаптеру основной машины и попытаться обмениваться пакетами с гостевой ОС. Наиболее универсальным способом является загрузка гостевого ПО. Во время загрузки ОС драйверы обычно проводят проверку и инициализацию своих устройств. По сообщениям об ошибках от драйверов можно судить о наличии ошибки в модели устройства. Хорошей практикой является работа двух разработчиков: один реализует виртуальное устройство, другой разрабатывает гостевое ПО, которое должно тестировать работу устройства.

Участие второго разработчика крайне желательно, т.к. это минимизирует риск тиражирования ошибки, вызванной неправильной трактовкой текста документации.

Процесс автоматизированной разработки ВМ включает следующие этапы.

1. Изучение документации с целью определения системы команд и версии процессора, перечня устройств и взаимосвязей между ними.
2. Составление формального описания состава ВМ на языке Python вручную и/или с использованием GUI.
3. Автоматизированная генерация заготовки ВМ.
4. Реализация на языке Си не формализованных особенностей работы.
5. Итеративная разработка: тестирование, отладка, доработка.

Формальное описание состава ВМ изначально разрабатывалось для ручного написания. Позже появился ГИП с возможностью отображения состава ВМ в виде схемы. Он позволяет изменять состав и сохранять описание в Python с использованием удобного для ручной доработки форматирования. Возможности ГИП отстают от возможностей ручного способа. Поэтому при составлении описания рекомендует попеременно использовать оба способа, т.к. некоторые манипуляции удобнее производить над текстовым описанием, а некоторые – через ГИП.

После генерации ВМ (п. 3) возможно сразу получить работоспособную ВМ. Однако часто разработка ВМ начинается одновременно с разработкой виртуальных устройств. Это вызвано тем, что тестировать устройства в составе ВМ легче. В результате к моменту завершения разработки последнего устройства, ВМ начинает работать сама собой. В некоторых случаях ВМ может требовать реализации нестандартного поведения, согласно документации (п. 4).

Доработка ВМ производится по тому же принципу, что и доработка устройств.

4. Автоматизированное восстановление алгоритма в виде блок-схемы с целью анализа его прикладных свойств и характеристик

В условиях отсутствия исходных текстов и документации выявление НДВ в бинарном коде программ вынужденно рассматривается как проверка выполнения базовых положений информационной безопасности в восстановленном представлении алгоритмов. При отсутствии «декларации», к нарушениям того, что декларируется следует относить: (1) любые программные дефекты, приводящие к сбоям, порче и утечке данных, (2) особенности реализации алгоритмов, приводящие к выдаче в открытый канал связи чувствительных данных в исходном виде либо после таких преобразований, когда эти данные могут быть полностью или частично восстановлены, иными словами – приводящие к утечке чувствительных данных. Программные дефекты в бинарном коде выявляют либо ручным анализом, либо применяя различные средства статического и динамического анализа [7, 18, 19]. В данной работе рассматривается второй вид нарушений, требующий активного участия эксперта. В этом случае человеком проводится проверка отдельных алгоритмов программы, т.к. тотальный ручной анализ современного ПО невозможен из-за его размера и сложности. Ставится задача автоматизации анализа свойств отдельного алгоритма экспертом.

В общем случае ручная проверка разбивается на следующие этапы: 1) локализация алгоритма в коде программы; 2) представление алгоритма в удобной для анализа форме; 3) исследование свойств алгоритма. К сожалению, полная автоматизация этого процесса невозможна, так как первый и третий этап требуют выполнения нетривиальных действий от эксперта-аналитика. Облегчить труд аналитика на этапе локализации алгоритма позволяет построение статико-динамического представления программы [20], выявление вызовов известных библиотечных функций и событий уровня операционной системы,

которые служат ориентирами в общем коде. Трудозатраты аналитика на выполнение третьего этапа, а также качество полученного результата во многом определяются способом представления анализируемого алгоритма. Получение такого представления алгоритма осложнено из-за отсутствия исходного текста программы, а следовательно – ее высокоуровневой семантики.

Разработка выразительного промежуточного представления (*Intermediate Representation – IR*) программы была и остается актуальной задачей. Существует большое число различных промежуточных представлений, имеющих свои преимущества и недостатки. Каждое из них разрабатывалось для конкретной области применения. Далее рассматриваются промежуточные представления, успешно решающие задачи анализа и преобразования кода программы в областях оптимизирующей компиляции, декомпиляции и высокоуровневого представления алгоритма программы.

4.1. Промежуточные представления программ для задач компиляции

Для задач компиляции наиболее известными являются IR, используемые в компиляторах, негласно признанных промышленным стандартом. А именно, промежуточные представления GENERIC, GIMPLE и RTL, реализованные в наборе компиляторов GCC, и LLVM IR, используемый в LLVM. Существуют и другие известные IR, например, основанное на графе представление программы PDG – program dependence graph (граф зависимостей программы) [21, 22], которое объединяет в себе информацию о потоке данных и потоке управления. Вершинами в этом графе являются команды, а ребра разделены на два типа: 1) выражающие зависимость по данным и 2) выражающие зависимость по управлению. Несмотря на то, что подобные IR хорошо зарекомендовали себя в задачах компиляции (анализ кода для выполнения оптимизирующих преобразований программы), они не пригодны для эффективного анализа бинарного кода и построения высокоуровневого представления алгоритма, так как анализируемый код лишен информации о высокоуровневой семантике программы.

4.2. Промежуточные представления программ для моделирования процессорных команд и анализа бинарного кода

Для задач анализа бинарного кода в общем и задач декомпиляции в частности развиваются специализированные промежуточные представления. Среди множества существующих можно выделить следующие: Pivot/Pivot 2 [23, 24], B2R2 IR [25], REIL [26], MAIL [27], BAP (BIL) [3, 28], BitBlaze [29], ESIL [5].

Промежуточный язык REIL – Reverse Engineering Intermediate Language [26] позволяет выполнять статический анализ кода программы в контексте задач обратной инженерии. Для моделирования машинных команд целевой архитектуры (или описания операционной семантики машинных команд целевой архитектуры) этот язык имеет в своем составе 17 команд, реализующих абстрактную машину с бесконечной памятью и бесконечным числом регистров. В последствии на основе языка REIL появился RREIL [30].

В платформе BAP [3] анализа бинарного кода для описания семантики команд целевой архитектуры используется один уровень промежуточного представления, реализуемый языком BIL [28]. В отличие от BAP аналогичная ей система BitBlaze [29] поддерживает два уровня промежуточного представления: VEX IL [31], реализованное в средстве динамической инструментации Valgrind, и Vine IL [29]. Первое позволяет описать максимально точно семантику машинных команд с учетом побочных эффектов, а второе IR позволяет сократить избыточную для выполнения анализа детализацию. Точно так же, как и REIL, язык Vine IL содержит команды для абстрактной машины с неограниченным объемом памяти и числом виртуальных регистров.

Еще одним схожим по назначению с вышеупомянутыми языками является язык ESIL – Evaluable String Intermediate Language [5], который используется для описания семантики машинных команд набором инструментов для обратной инженерии radare2. Данный язык реализует абстрактную машину (как и в предыдущих случаях) и позволяет детально описать побочные эффекты машинных команд. Выражения на данном языке записываются в обратной польской нотации.

Также к вышеупомянутому множеству языков можно отнести специализированный промежуточный язык MAIL – Malware Analysis Intermediate Language [27], который содержит команды для описания структуры и поведения бинарной программы. Этот язык позволяет выполнять статический анализ программ, ориентированный на обнаружение полиморфного вредоносного кода.

Стоит отметить, что перечисленные выше промежуточные языки, используемые для анализа бинарного кода, в первую очередь ориентированы на детальное описание операционной семантики команд целевой архитектуры. Поэтому они не подходят для высокоуровневого представления алгоритма программы. Однако, они могут быть использованы в качестве начального IR, из которого итерационно будет удаляться избыточная детализация с последующим повышением уровня представления.

4.3. Высокоуровневые представления алгоритма программ

Если задачу построения высокоуровневого представления алгоритма программы свести к задаче декомпиляции, то результатом будет восстановленный из бинарного кода исходный код на высокоуровневом языке. Для решения данной задачи известные методы декомпиляции [1, 32, 33, 34] выполняют анализ потока управления программы для восстановления ее структуры. Основная идея анализа заключается в сопоставлении сегментов графа потока управления программы с шаблонами из заранее подготовленного экспертами множества, для которых определена соответствующая высокоуровневая языковая конструкция (например, `if-then-else`, `switch-case` и др.). В результате выполненных компилятором оптимизирующих преобразований в коде программы могут появиться несводимые регионы, для которых отсутствует соответствующая языковая конструкция, и тогда используется оператор безусловного перехода для повторения потока управления из бинарного кода. На практике такое случается часто. Кроме того, нерешенной остается задача восстановления структур данных. Это приводит к тому, что результирующий код фактически является копией машинного на высокоуровневом языке. Такой код не пригоден не только для высокоуровневого представления алгоритма, но и для ручного анализа экспертом. В некоторых случаях он может быть полезен для автоматического анализа либо для перекомпиляции программы под другую целевую архитектуру.

Существуют подходы к декомпиляции бинарного кода программы с привлечением методов машинного обучения [35, 36]. Такие подходы сосредоточены на двух задачах: 1) автоматизировать трудоемкий процесс создания шаблонных фрагментов графа потока управления с известной языковой структурой высокого уровня при помощи возможностей аппарата машинного обучения и 2) использовать возможности машинного обучения для установки соответствия операндов машинного кода и переменных высокоуровневых языковых конструкций. На сегодняшний день такие инструменты представляют только академический интерес, и говорить об их применении для решения практических задач еще рано.

Существуют промежуточные представления более высокого уровня, чем те, которые используются для решения задачи декомпиляции. Например, в работе [37] предложено IR для задач поиска криптографических функций.

В работе [38] предложено гибридное высокоуровневое представление программы, объединяющее в себе граф потока данных и потока управления (HI-CFG – Hybrid Information- and Control-Flow Graph). Граф HI-CFG состоит из двух типов вершин: 1) соответствующих фрагментам кода программы (они могут представлять код модуля или функции) и 2) соответствующих структурам данных программы. Вершины в графе связаны тремя типами ребер:

1) ребра между вершинами-фрагментами соответствуют ребрам графа потока управления и показывают последовательность передачи управления;
2) вершины второго типа соединены между собой ребрами, отражающими зависимость по данным и показывающими направление передачи информации от одной структуры данных к другой;
3) третий, последний, тип ребер соединяет между собой вершины обоих типов в порядке отношения «производитель-потребитель». Данный тип ребер показывает, какие структуры данных являются входными аргументами в фрагмент кода, а какие – выходными.

HI-CFG представляет сложную программу в виде ее декомпозиции на основные логические компоненты, интерфейсы взаимодействия между которыми определяются используемыми структурами данных. Такое высокоуровневое представление алгоритма не только позволяет выполнять анализ программы в автоматическом режиме, но и является удобным для восприятия и ручной обработки аналитиком.

На Рис. 4 показан пример высокоуровневого представления HI-CFG программы, которая выполняет разбор (parser) двух типов команд (decoder1/decoder2), поступающих в качестве входного аргумента (input_buffer), декодирует их, используя таблицу поиска (lookup_tbl), и выполняет соответствующие вычисления (compute1/compute2).

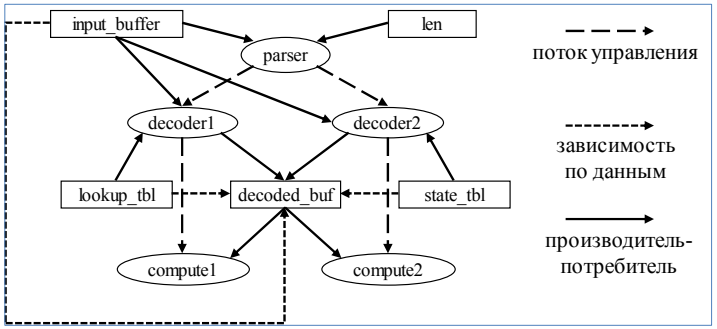

Рис. 4. Пример высокоуровневого представления HI-CFG программы
Fig. 4. An example of a high-level presentation of an HI-CFG program

Восстановление структур данных и определение буферов программы играют особую роль в получении точного и корректного высокоуровневого представления HI-CFG, так как они определяют интерфейсы, с помощью которых связываются логические компоненты программы. Отсутствие отладочной информации в бинарном коде существенно усложняет задачу восстановления структур данных. Для решения этой задачи авторы HI-CFG при построении представления используют известные методы, основанные на динамическом анализе помеченных данных, анализе шаблонов обращения к памяти, распространении типов данных от входных и выходных аргументов хорошо известных библиотечных и системных вызовов.

Более конкретно, формирование буферов выполняется алгоритмом, аналогичным тому, что используется в инструменте Howard [39] для обнаружения массивов. Этот алгоритм динамического анализа бинарного кода основан на эвристическом предположении, что доступ к массивам осуществляется в циклах. Таким образом, алгоритм группирует в массивы используемые области памяти в циклах, доступ к которым осуществлялся с одинаковым смещением относительно предыдущего элемента или относительного базового адреса массива. Стоит отметить, что данный алгоритм в реализации авторов HI-CFG не распознает многомерные массивы.

Для вывода типов используется инструмент REWARD [40], выполняющий динамический анализ потока данных на основе алгоритма Aggregate Structure Identification – ASI (алгоритм идентификации агрегированных структур данных) [41], который, в свою очередь, базируется на алгоритме Хиндли-Милнера [42]. Идея алгоритма заключается в следующем: при выполнении вызова библиотечной функции, системного вызова или выполнении команды, аргументы или операнды которых имеют достоверно известный тип, осуществляется пометка соответствующих аргументам или операндам областей памяти их типом данных. После этого информация о выведенном типе данных распространяется по зависимым операндам в процессе выполнения программы.

Информация о программе, содержащаяся в HI-CFG, может быть по отдельности получена существующими инструментами анализа бинарного кода. Авторы этого высокоуровневого промежуточного представления программы попытались объединить в нем одном результаты работы существующих алгоритмов анализа. Например, наличие ребер трех типов позволяет охарактеризовать модификацию информационного потока в терминах потока данных и потока управления. Представление PDG [21, 22] также имеет ребра, описывающие поток данных и поток управления, однако он не имеет вершин, являющихся структурами данных программы.

Таким образом, среди существующих подходов к промежуточному представлению программы HI-CFG является наиболее подходящим для ручного анализа и высокоуровневого представления алгоритма (на уровне блок-схем). Также, HI-CFG поддерживает различные уровни детализации: вершины, являющиеся фрагментами кода, могут представлять как базовые блоки кода, так и функции, а вершины, соответствующие структурам данных – как отдельные ячейки памяти, так и крупные области (типизированные или нетипизированные). Кроме того, представление HI-CFG содержит большой набор примитивов (ребра и вершины различных типов), что предоставляет возможность для реализации алгоритмов автоматического анализа различных схем потоков данных и выполнения.

5. Иерархическое высокоуровневое представление алгоритма

В контексте обнаружения НДВ, как правило, необходимо выполнить анализ алгоритмов, оперирующих чувствительными данными. Высокоуровневое представление анализируемого алгоритма должно содержать команды, участвующие в формировании результата – *результирующего буфера*, тем самым, позволяя определить, какие входные данные использовались для выполнения алгоритма, и где они размещались, происходили ли утечки чувствительных данных при выполнении шагов алгоритма.

При проектировании промежуточного представления необходимо учесть, что оно будет использоваться по двум основным сценариям: 1) построение компактного аннотированного представления в виде иерархической блок-схемы, пригодной для ручного анализа и 2) изучение свойств алгоритма в автоматическом режиме для определения декларированных и недекларированных потоков данных. Представление должно быть применимо для динамического анализа бинарного кода в автономном режиме, то есть его построение следует реализовать по трассе выполнения программы.

Трасса выполнения программы представляет собой последовательность выполненных процессором команд, а также значения регистров на каждом шаге. Такая трасса может быть получена в результате исполнения программы полносистемным эмулятором, например, QEMU [43], либо в результате выполнения программы под управлением средств динамического инструментирования бинарного кода, например, Pin [44], DynamoRIO [45], Valgrind [31] и др.

Предлагаемое представление алгоритма программы основано на гиперграфе с иерархической организацией и состоит из двух типов вершин: 1) представляющих команды, выполняющиеся на определенном шаге трассы (обозначим их термином *точка*); 2) представляющих ячейку абстрактной памяти (это может быть диапазон адресов в виртуальном адресном пространстве, регистр или его часть) на определенном шаге выполнения трассы (обозначим их термином *буфер*). Ребра в гиперграфе отображают зависимости по данным. Вершины гиперграфа типа точка могут быть объединены в подмножества, называемые *фрагментами*, а фрагменты – в *суперблоки*. Логически связанные между собой вершины типа буфер могут быть объединены в подмножества типа *супербуфер*.

Вершины-фрагменты соответствуют фрагментам кода в трассе, то есть линейной последовательности шагов в трассе, на которых не выполнялись команды вызова функций и возврата из них. Например, если функция A вызывает только одну функцию B, которая не содержит вызовов функций, то в этом случае будет сформировано три фрагмента кода: 1) F_{A1}, содержащий последовательность команд функции A до вызова функции B, 2) F_B – последовательность команд функции B, 3) F_{A2} – последовательность команд функции A, которые выполнялись после возврата из функции B. Суперблоки соответствуют экземплярам вызова функций. Они состоят из фрагментов, принадлежащих данному экземпляру вызова функции, и других суперблоков, которые, в свою очередь соответствуют экземплярам функций, вызванных из данного.

Супербуферы и буферы соответствуют структурам данных программы и определяют интерфейсы взаимодействия между фрагментами и суперблоками. Также они характеризуют значение потока данных на момент входа в фрагменты или суперблоки или выхода из них. Для суперблоков входные и выходные аргументы определяются буферами и супербуферами.

Предложенное высокоуровневое представление позволяет описать алгоритм формирования результирующих буферов в пригодном для ручного и автоматического анализа виде. Ниже представлен алгоритм построения предложенного иерархического представления.

5.1. Алгоритм построения иерархического высокоуровневого представления

В основе построения высокоуровневого представления лежит алгоритм отслеживания потока данных в обратном направлении (обратный слайс трассы). Поэтому представление алгоритма строится только по тем точкам (шагам в трассе), которые участвуют в формировании результирующего буфера, остальные считаются избыточными, не относящимися к искомому алгоритму, и поэтому не рассматриваются. Точки внутри фрагментов и сами фрагменты между собой связываются ребрами, формируя таким образом граф потока данных.

На вход алгоритма поступает *начальный буфер*, являющийся результатом работы интересующего алгоритма, и диапазон шагов в трассе, на которых выполнялись его команды. Алгоритм выполняет 2 основных шага: 1) выделение в трассе точек, относящихся к алгоритму формирования начального буфера, и объединение их в подмножества фрагментов (Рис. 5) и 2) объединение фрагментов в подмножества

суперблоков (Рис. 6). Предполагается, что по трассе выполнения предварительно построено статическое представление программы, содержащее информацию о связях между командами и шагами в трассе и о вызовах функций, восстановлен стек вызовов функций.

```
    Вход: t          – трасса выполнения
                     программы
          C          – информация о вызовах
                     функций
          b:< a_i, l_i  – начальный буфер
          >
    Выход:F          – множество фрагментов
          P          – множество точек

    createPointAndFragments(t, C, b:< a_i, l_i >)
1     F = ∅, f = ∅, P = ∅, p = ∅
2     while (p = getNextPoint(t, b:< a_i, l_i >))
3       P ∪= {p}
4       if (f == ∅)
5         [b, e] = getFragmentBound(C, p)
6         f = newFragment([b, e])
7         addPoint(f, p)
8       else if (p ∈ [b, e])
9         addPoint(f, p)
10      else
11        F ∪= {f}
12        [b, e] = getFragmentBound(C, p)
13        f = newFragment([b, e])
14        addPoint(f, p)
15    if (P ≠ ∅)
16      F ∪= {f}
```

Рис. 5. Псевдокод алгоритма построения множества точек и фрагментов представления
Fig. 5. Pseudocode of the algorithm for constructing a set of points and presentation fragments

На Рис. 5 показан псевдокод алгоритма, реализующего первый шаг построения высокоуровневого представления. Для его выполнения требуется трасса t выполнения программы, информация C о содержащихся в трассе вызовах функций (номер шага, на котором выполнялись команды вызова функций и возврата из них) и начальный буфер b: $< a_i, l_i >$, область в адресном пространстве по адресу a_i и имеющая длину l_i. В результате работы данного шага будут построены множества точек P и фрагментов F. Алгоритм осуществляет восходящий проход по точкам в трассе, относящимся к искомому алгоритму программы (точки из множества, полученного обратным слайсом по начальному буферу). Функция `getNextPoint` возвращает очередную точку p из трассы в порядке ее обратного исполнения. При помощи функции `newFragment` создаются фрагменты. Для их создания необходимо указать номера шагов начала фрагмента и его конца – $[b, e]$. Границы фрагментов получаются при помощи функции `getFragmentBound` и содержатся в информации о вызовах функций, которая формируется на этапе построения статического представления программы. Добавление точки во фрагмент выполняется функцией `addPoint`.

На Рис. 6 приведен псевдокод алгоритма формирования суперблоков иерархического высокоуровневого представления. Алгоритм представляет собой запуск рекурсивной процедуры `superBlockCover` создания многоуровневых, вложенных суперблоков. Уровень вложенности определяется глубиной стека вызовов функций, то есть каждый суперблок является экземпляром функции, а вложенные в него суперблоки –

экземплярами вызываемых функций. На нижнем уровне хранятся фрагменты, наполненные точками. В качестве входных аргументов процедура `createSuperBlocks` принимает множество F фрагментов, полученных в результате работы предыдущего шага, а также стек C вызовов функций трассы, который строится на этапе предварительной обработки трассы. Результатом работы алгоритма является множество S суперблоков. В псевдокоде используются следующие функции:

1) `getCall(`$C, x.b$`)`, используя стек вызовов функций C и номер шага в трассе, возвращает экземпляр функции c, содержащей этот шаг. Экземпляры функций имеют начальные $c.b$ и конечные $c.e$ шаги в трассе;
2) `newSuperBlock(`$[c.b, c.e]$`)` создает новый суперблок s. Фрагменты и суперблоки имеют начальный шаг в трассе (b) и конечный (e);
3) `createEdges(`s, I`)` связывает ребрами фрагменты и суперблоки из множества I и добавляет получившийся граф в суперблок s;
4) `getPrevStackCall(`C, c`)`, используя стек вызовов C, возвращает экземпляр функции предыдущего уровня относительно вызова экземпляра функции c текущего уровня.

```
    Вход: F – множество фрагментов
          C – информация о вызовах функций,
            содержащая стек вызовов функций трассы
    Выход:S – множество суперблоков

    createSuperBlocks(F, C)
1     S = ∅, B = F
2     superBlockCover(B, C)

3   superBlockCover(B, C)
4     foreach (x in B)
5       B \= {x}
6       c = getCall(C, x.b)
7       while (c ≠ ∅)
8         if (x.b ≤ c.b)
9           break
10        s = newSuperBlock([c.b, c.e])
11        S ∪= {s}
12        I = {y_i}: y_i ∈ B ∧ y_i.b ≥ c.b
13        B \= I
14        superBlockCover(I, C)
15        createEdges(s, I)
16        B ∪= {s}
17        c = getPrevStackCall(C, c)
```

Рис. 6. Псевдокод алгоритма построения суперблоков высокоуровневого представления
Fig. 6. Pseudo-code of a high-level representation superblock construction algorithm

Процедура `superBlockCover` выполняет следующие шаги.

1. Из множества B фрагментов и суперблоков извлекается очередной элемент x. По его начальной позиции в трассе $x.b$, используя стек вызовов функций C, получается экземпляр c вызова функции (Рис. 6, строки 4-6), и запускается процесс построения суперблоков для каждого уровня стека вызовов, начиная с вызова функции c (Рис. 6, строка 7). Уровень вызова экземпляра функции c считается текущим.

2. На каждом уровне стека вызовов функций выполняется проверка, находится ли текущий фрагмент или суперблок на уровне не ниже текущего (Рис. 6, строка 8), и если нет, то оставшиеся уровни вызова не рассматриваются. В противном случае создается новый суперблок s при помощи функции `newSuperBlock`, который добавляется в результирующее множество S, и начинается выполнение следующего шага.
3. Из множества суперблоков и фрагментов выбираются те, которые в стеке вызовов функций расположены на уровне не ниже текущего уровня. Выбранные элементы формируют множество I и исключаются из множества B. После чего процедура `superBlockCover` повторяется рекурсивно для вновь сформированного множества I.
4. После возвращения управления из рекурсивного вызова процедуры `superBlockCover` полученные фрагменты и суперблоки связываются между собой ребрами. Затем сформированный граф добавляется в текущий суперблок s, а сам суперблок помещается в результирующее множество B (Рис. 6, строки 15-16).
5. По стеку вызовов получается экземпляр функции предыдущего уровня вызова, который становится текущим. Далее процедура повторяется, начиная с шага 2.

5.2. Пример построения высокоуровневого иерархического представления алгоритма

Ниже рассматривается пример предлагаемого представления с различными уровнями детализации для программы, приведенной на Рис. 7. Программа генерирует ключ `key` при помощи функции `GenerateKey`, получает текст `text` из файла при помощи функции `ReadText` и выполняет его шифрование алгоритмом AES, используя сгенерированный ключ. Результат шифрования хранится в строке `cipher`. Детали реализации функций для демонстрации высокоуровневого представления не важны.

```
key = GenerateKey()
text = ReadText()
cipher = aes(key, text)
```

Рис. 7. Псевдокод программы, выполняющей шифрование текста алгоритмом AES
Fig. 7. Pseudocode for a program encrypting text using AES

Для приведенной программы имеется трасса ее выполнения, а также известен номер шага x, на котором завершена работа алгоритма AES. Трасса представлена на Рис. 8а. Для простоты в ней намеренно опускаются конкретные команды, а указаны только точки $p_0, \dots, p_{27}$ – шаги выполнения. Для наглядности фрагменты кода в трассе выделены серым фоном. Зашифрованный текст хранится в строке `cipher`, которая размещена в области памяти с адресом a_1 и длиной l_1. Эта область памяти формирует отслеживаемый начальный буфер b_1. Стартовой позицией в трассе, с которой начинается построение высокоуровневого представления алгоритма, является шаг x.

Как уже упоминалось выше, высокоуровневое иерархическое представление содержит только те точки, которые относятся к искомому алгоритму, то есть точки, принадлежащие отслеживаемому потоку данных. После выполнения первого этапа алгоритма построения представления, реализованного функцией `createPointAndFragments`, будет сформировано множество точек $P = \{p_1, p_4, p_5, p_7, p_{10}, p_{11}, p_{13}, p_{16}, p_{19}, p_{22}, p_{23}, p_{25}\}$, а также множество фрагментов $F = \{f_0, \dots, f_8\}$. Полученное множество точек соответствует обратному слайсу трассы. На Рис. 8б показаны полученные множества точек и фрагментов на трассе. В построенном высокоуровневом представлении точки и связанные между собой ребрами фрагменты будут формировать граф потока данных. Рис. 8в демонстрирует направления потоков данных для искомого алгоритма, вычисляющего результирующий буфер b_1.

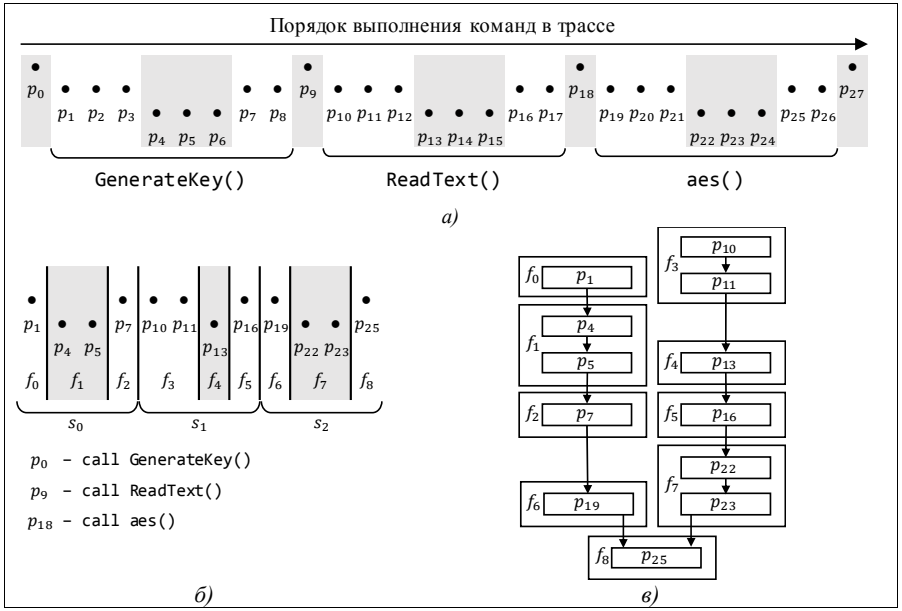

Рис. 8. Трасса выполнения и граф потока данных программы, псевдокод которой представлен на Рис. 7:
а) исходная трасса; б) результат обратного слайса для буфера <a_1, l_1>;
в) направление потоков данных для искомого алгоритма, вычисляющего результирующий буфер b_1
Fig. 8. The execution trace and the graph of the program data flow, whose pseudocode is shown in Fig. 7:
a) source track; б) the result of the inverse slice for the buffer <a_1, l_1>;
в) the direction of the data flows for the desired algorithm that computes the resulting buffer b_1

Полученное множество фрагментов F поступает на вход второго шага алгоритма построения высокоуровневого представления, реализованного процедурой `createSuperBlocks`. Алгоритм восстановления стека вызовов функций C в данной работе не рассматривается. Предполагается, что он уже восстановлен на этапе построения статического представления программы [6, 20, 46]. На втором, завершающем, этапе будет сформировано множество S суперблоков, соединенных между собой ребрами, представляя таким образом *гиперграф*. На Рис. 9 показано результирующее высокоуровневое представление алгоритма с разным уровнем детализации. В результате для примера с Рис. 7 будут сформированы следующие фрагменты:

$$f_0 = \{p_1\}, f_1 = \{p_4, p_5\}, f_2 = \{p_7\}, f_3 = \{p_{10}, p_{11}\}, f_4 = \{p_{13}\},$$
$$f_5 = \{p_{16}\}, f_6 = \{p_{19}\}, f_7 = \{p_{22}, p_{23}\}, f_8 = \{p_{25}\}$$

и суперблоки:

$$s_0 = \{f_0, f_1, f_2\}, s_1 = \{f_3, f_4, f_5\}, s_2 = \{f_6, f_7, f_8\}.$$

Суперблоку s_0 соответствует функция `GenerateKey`, s_1 – `ReadText`, s_2 – `aes`. В случае необходимости, для минимизации представления, суперблоки s_0, s_1, s_2 могут быть свернуты в суперблок s_3. Это позволяет сделать иерархическая организация предложенного представления.

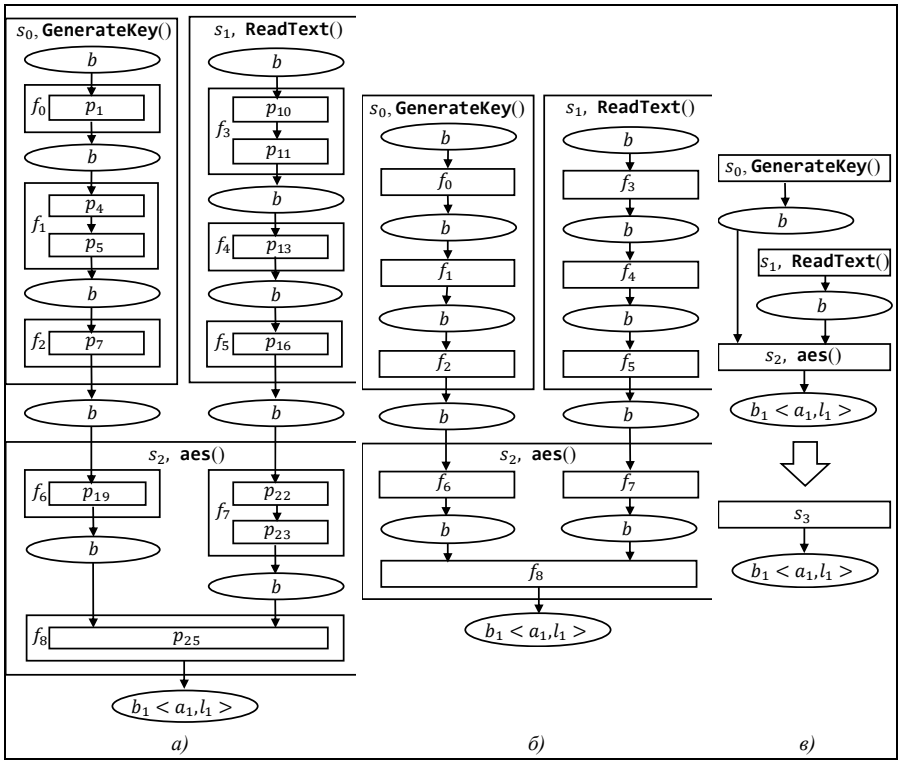

Рис. 9. Высокоуровневое иерархическое представление программы, псевдокод которой представлен на Рис. 7, с различными уровнями детализации: а) уровень точек; б) уровень фрагментов; в) уровень суперблоков
Fig. 9. High-level hierarchical representation of the program, whose pseudocode is shown in fig. 7, with various levels of detail: a) the level of points; б) the level of fragments; в) the level of superblocks

Высокоуровневое представление основано на гиперграфе, содержащем точки (шаги в трассе), на которых выполняются команды искомого алгоритма, формирующие отслеживаемый поток данных. Буферы позволяют получить значение потока данных в различные моменты времени выполнения алгоритма, а фрагменты, суперблоки и супербуферы – выполнить его масштабирование.

Иерархическая организация представления является пригодной как для выполнения анализа алгоритма в ручном режиме, скрыв избыточные подробности алгоритма путем сворачивания некоторых элементов (фрагментов или суперблоков), так и для автоматического анализа.

5.3 Объединение буферов в супербуферы

Используемые структуры данных в программе определяют интерфейсы взаимодействия между модулями программы. Интерфейсы взаимодействия являются одним из ключевых пунктов в вопросе восстановления представления алгоритма, так как позволяют установить логические связи между его компонентами. В контексте анализа бинарного кода установление логической связи затруднено из-за отсутствия в программе информации о высокоуровневой семантике.

В предлагаемом иерархическом высокоуровневом представлении алгоритма программы структуры данных определяются буферами и супербуферами. Ввиду отсутствия информации о высокоуровневой семантике, а именно об используемых типах структур данных, в процессе построения представления искомого алгоритма нельзя однозначно ответить на вопрос, чем является та или иная часть отслеживаемого потока данных на конкретном шаге выполнения: самостоятельной переменной, элементом массива или полем агрегированной структуры данных. Избежать подобной неопределенности позволяют буферы, которые в простейшем случае представляют собой абстрактные ячейки памяти. Однако такой подход затрудняет восприятие представления алгоритма аналитиком, а именно, не позволяет определить интерфейсы взаимодействия между его логическими компонентами. Например, если функция A модифицировала n элементов массива, а после этого функция B использовала их в вычислениях результирующего буфера, то в худшем случае может быть получено представление, в котором будет указано, что эти два модуля связаны между собой по n буферам. В случае больших значений n анализ подобного представления человеком не возможен. Безусловно, если все n элементов образуют смежную непрерывную область, то они будут объединены в один буфер, но такое встречается не всегда. На практике чаще всего шаблоны доступа к памяти не формируют непрерывную область, либо используется нетривиальное размещение структур данных в памяти, что приводит точно к такому же результату.

Избежать избыточную детализацию зависимостей между модулями алгоритма позволяет группировка буферов в супербуферы. В работе предложено две схемы объединения:

1) объединение множества буферов, формирующих непрерывную область, в один буфер;
2) объединение буферов в супербуферы на основе разбиения множества входных и выходных буферов суперблоков на подмножества, содержащие «эквивалентные» буферы.

Первый случай является тривиальным и не нуждается в пояснении в отличие от второго. Понятие «эквивалентность» в данном случае означает, что два различных буфера были выработаны одним общим суперблоком и используются затем в одних и тех же суперблоках, что позволяет их логически объединить. Для пояснения схемы объединения введем следующие операции для суперблоков и буферов:

- $out(s)$ – возвращает множество выходных буферов суперблока s;
- $in(s)$ – возвращает множество входных буферов суперблока s;
- $consumer(b)$ – возвращает множество суперблоков, которые используют буфер b в качестве входного, например, для представления с Рис. 10в $consumer(b_2) = \{s_2, s_3\}$, а $consumer(b_4) = \{s_3\}$;
- $producer(b)$ – возвращает суперблок, который создает буфер b, иначе говоря, $producer(b) = s \mid b \in out(s)$. Например, $producer(b_5) = s_1$ (Рис. 10в).

Таким образом, объединение выходных буферов $out(s)$ во множество супербуферов $B = \{B_i\}$ для суперблока s выполняется так, чтобы каждый супербуфер B_i содержал все буферы b с одинаковым множеством $consumer(b)$.

Ниже приводится пояснение второй схемы объединения буферов в супербуферы для примера с рис. 10. Для демонстрации рассматриваются два варианта представления.

Вариант 1. В результате построения высокоуровневого представления алгоритма было сформировано два суперблока s_1 и s_2 (Рис. 10а), множество выходных буферов $out(s_1) = \{b_1, b_2, b_3\}$ и множество входных буферов $in(s_2) = \{b_1, b_2, b_3\}$. Предполагается, что множества входных и выходных буферов не формируют непрерывную область абстрактной памяти (например, из виртуального адресного

пространства или регистрового файла), поэтому на этапе построения не были объединены в один выходной и один входной буферы. На Рис. 10б изображен результат объединения буферов в супербуферы по второй схеме для данного варианта представления. Так как все множество $out(s_1)$ – выход суперблока s_1 – поэлементно соотносится со множеством $in(s_2)$ – входом суперблока s_2, то в результате будет сформирован супербуфер $B_1 = \{b_1, b_2, b_3\}$.

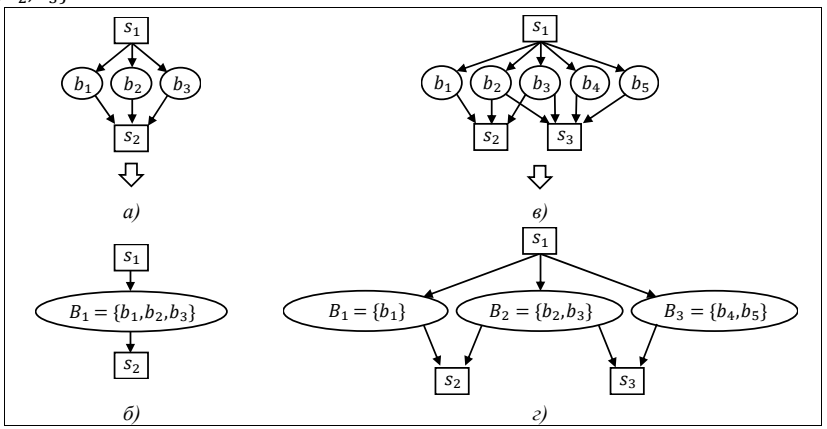

Рис. 10. Пример объединения буферов в супербуферы
Fig. 10. Example of combining buffers into superbuffers

Вариант 2. Фрагмент высокоуровневого представления алгоритма, соответствующий данному варианту, изображен на рис. 10в и содержит три суперблока s_1, s_2, s_3 со множествами $out(s_1) = \{b_1, b_2, b_3, b_4, b_5\}$ выходных буферов из s_1, $in(s_2) = \{b_1, b_2, b_3\}$ и $in(s_3) = \{b_2, b_3, b_4, b_5\}$ входных буферов в суперблоки s_2 и s_3. В этом случае выходные буферы b_2 и b_3 суперблока s_1 используются в качестве входных в суперблоки s_2 и s_3, то есть $consumer(b_2) = consumer(b_3) = \{s_2, s_3\}$. Поэтому выходные буферы суперблока s_1 будут сгруппированы в супербуферы $B_1 = \{b_1\}, B_2 = \{b_2, b_3\}, B_3 = \{b_4, b_5\}$, а не в $B_1 = \{b_1, b_2, b_3\}$ и $B_2 = \{b_3, b_4, b_5\}$, как может показаться на первый взгляд.

6. Набор тестовых программ для оценки результата построения высокоуровневого представления алгоритма

Для оценки возможностей средств восстановления высокоуровневого представления алгоритма по бинарному коду в работе предложен набор программ, содержащий как синтетические тесты, так и реально используемые приложения. При составлении тестового набора учитывался характер задач, которые требуется решить в процессе анализа исследуемых программ. К числу таких задач относится выявление ошибок логики приложения, потенциально приводящих к возникновению утечек чувствительных данных.

Набор тестов, состоящий из реальных приложений и синтетических тестов, должен покрывать значительный класс программ из различных прикладных областей и содержать разнообразные сценарии выполнения, которые либо приводят к возникновению утечек чувствительных данных, либо нуждаются в ручном анализе для доказательства их отсутствия. Для оценки результата построения высокоуровневого представления алгоритма предлагается использовать следующие реальные программы и модельные примеры:

1) модуль установки SSL-соединения браузеров Google Chrome и Mozilla Firefox;

2) программа депонирования криптографических ключей, т.е. реализующая их утечку в открытый канал в виде следующих сценариев:
   a) получение ключа и его утечка происходят в рамках одного потока выполнения;
   b) получение ключа, работа с ним и утечка разнесены по различным потокам выполнения;
3) синтетический тест, содержащий утечку чувствительных данных, оставшихся в стеке после выполнения функции, оперирующей ими;
4) программа, выполняющая «повреждение» секретного ключа перед его использованием по следующей схеме:
   a) перезапись части ключа константными значениями;
   b) перезапись ключа пользовательскими данными в результате срабатывания уязвимости CWE-123 (Write-what-where Condition, запись произвольных данных в произвольную область);
5) программа, реализующая клавиатурный фильтр (сниффер, «клавиатурный шпион»);
6) программа, использующая одинаковый генератор случайных чисел без повторной инициализации для генерации ключа и случайных данных с последующей отправкой в открытый канал текста (зашифрованного при помощи сгенерированного ключа) и сгенерированных случайных данных.

Далее подробно рассматривается тест 6 как наиболее репрезентативный.

Тест 6 представляет собой программу, которая выполняет шифрование текста алгоритмом AES и записывает зашифрованный текст в файл. Для того чтобы расшифровать содержащийся в файле текст, необходим ключ, который в зашифрованном виде также записан в файле. В свою очередь, для того чтобы выполнить расшифровку ключа, необходимо ввести пароль, с помощью которого был зашифрован ключ перед записью в файл.

Блок-схема алгоритма формирования результирующего буфера, который будет записан в файл и который содержит зашифрованные ключ, текст и случайно сгенерированные данные, представлена на рисунке Рис. 11. Алгоритм содержит следующие основные шаги:

1) получение пароля (Password) и входного текста (Input Text);
2) инициализация генератора случайных чисел текущим временем;
3) генерирование ключа для шифрования входного текста с помощью генератора случайных чисел из шага 2;
4) генерирование двух раундовых ключей алгоритмом расширения ключа (Key Expansion) на основе введенного пароля (Password) и сгенерированного на шаге 3 ключа. Первый раундовый ключ будет использоваться для шифрования ключа, сгенерированного на шаге 3, а второй – введенного текста;
5) шифрование алгоритмом AES второго ключа, используя раундовый ключ, полученный по введенному паролю;
6) шифрование алгоритмом AES введенного текста (Input Text), используя раундовый ключ, полученный по случайно сгенерированному на шаге 3 ключу;
7) объединение шифров, полученных на шагах 5 и 6, в одном буфере (Result Buffer);
8) дописывание случайных данных (Random Data) в конец результирующего буфера, используя для этого проинициализированный на шаге 2 генератор случайных чисел.

После этого результирующий буфер сохраняется в файл. Так как для заполнения результирующего буфера случайными данными используется тот же самый генератор случайных чисел, что и для создания ключа шифрования, этот ключ может быть восстановлен, а текст расшифрован.

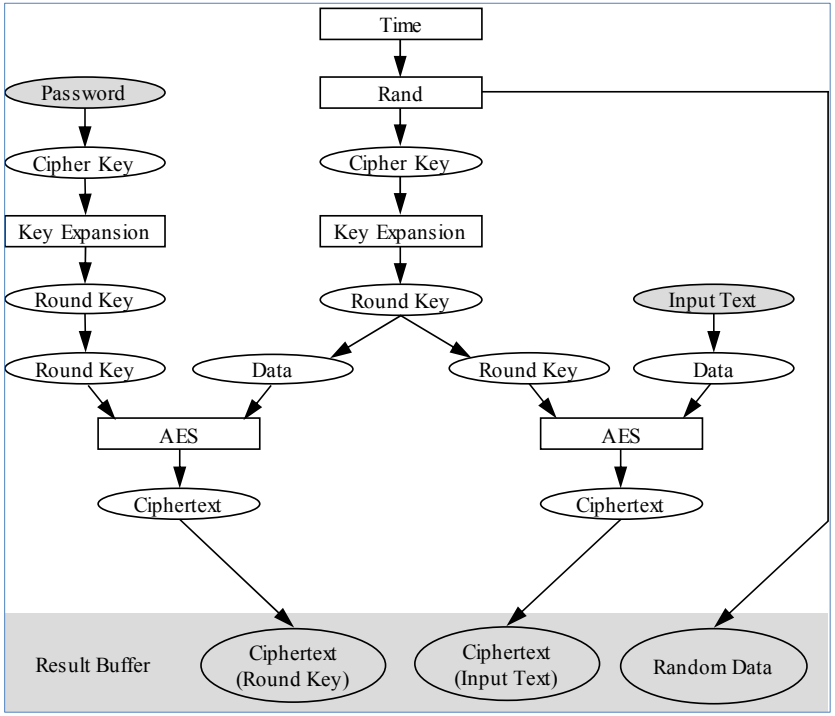

Рис. 11. Блок-схема алгоритма формирования результирующего буфера
Fig. 11. The block diagram of the algorithm for the constructing of the resulting buffer

7. Заключение

В работе представлен программный комплекс для выявления НДВ в условиях отсутствия исходного кода. Комплекс применим для ПО различных классов и различных процессорных архитектур. В его состав входят среда контролируемого выполнения на базе эмулятора QEMU, среда анализа бинарного кода TPAL, пакет инструментов разработчика QEMU QDT. В совокупности эти средства обеспечивают ускоренное получение практических результатов анализа бинарного кода даже в тех ситуациях, когда изначально отсутствует возможность проведения динамического анализа.

Набор инструментов автоматизации QDT позволяет ускорить процесс разработки ВМ с помощью автоматической генерации анализатора машинных команд, заготовок моделей процессора, устройств и ВМ. Язык I3S и одноимённый инструмент облегчают написание семантики машинных команд. Инструмент i2c позволяет автоматически протестировать наибольшую часть семантики и оценить покрытие. Графический интерфейс пользователя облегчает восприятие и редактирование связей между компонентами большой ВМ.

Для ускорения этапа ручного анализа разработано высокоуровневое иерархическое представление алгоритма программы на основе блок-схем и алгоритм его построения. Разработанное представление основывается на гиперграфе и позволяет выполнять анализ потока данных в автоматическом и ручном режимах. Его иерархическая организация делает возможным исследование свойств алгоритма программы на различных уровнях детализации и решение задач обнаружения НДВ, например, утечек конфиденциальных данных. Также предложен подход к повышению качества полученного представления

алгоритма с помощью объединения отдельных потоков данных в один, связывающий логические модули алгоритма.

Основным направлением развития представленного комплекса станет его интеграция с другими средствами анализа, такими как фаззер, статический анализатор бинарного кода, анализатор сетевого трафика и др. Такая интеграция позволит в перспективе выстраивать сложные технологические цепочки из различных средств анализа, адаптируясь под требования процессов разработки, сертификации, подтверждения доверия во время эксплуатации ПО.

Список литературы / References

[1]. The IDA Pro disassembler and debugger. URL http://www.hex-rays.com/idapro/, accessed 20.11.2019.

[2]. NSA, Ghidra is a software reverse engineering (SRE) framework. NSA. URL https://github.com/NationalSecurityAgency/ghidra, accessed 20.11.2019.

[3]. D. Brumley, I. Jager, T. Avgerinos, E. J. Schwartz. BAP: A Binary Analysis Platform. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 463-469.

[4]. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel,G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. IEEE Symposium on Security and Privacy, 2016, pp. 138-157.

[5]. ESIL: Radare2 book. URL https://radare.gitbooks.io/radare2book/content/disassembling/esil.html, accessed 20.11.2019.

[6]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 251-276 / V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Methods and software tools for combined binary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[7]. Aslanyan H.K. Plarform for interprocedural static analysis of binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018. pp. 89-100. DOI: 10.15514/ISPRAS-2018-30(5)-5.

[8]. Ефимов В.Ю., Беззубиков А.А., Богомолов Д.А., Горемыкин О.В., Падарян В.А. Автоматизация разработки моделей устройств и вычислительных машин для QEMU. Труды ИСПРАН, том 29, вып. 6, 2017 г., стр. 77-104. DOI: 10.15514/ISPRAS-2017-29(6)-4 / Efimov V.Yu., Bezzubikov A.A., Bogomolov D.A., Goremykin O.V., Padaryan V.A. Automation of device and machine development for QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 77-104 (In Russian). DOI: 10.15514/ISPRAS-2017-29(6)-4.

[9]. А.И. Гетьман, Ю.В. Маркин, Д.О. Обыденков, В. А. Падарян. Архитектура системы глубокого разбора сетевого трафика. Системный администратор, том 1, вып. 2, 2018 г., стр. 83-87 / A.I. Get'man, Yu.V. Markin, D.O. Obidenkov, V. A. Padaryan. An architecture of deep packet inspection system. Sistemnyj administrator, vol. 1, issue 2, 2018, pp. 83-87 (in Russian).

[10]. Bellard F. QEMU, a fast and portable dynamic translator. In Proc. of the USENIX Annual Technical Conference, 2005, pp. 41-46.

[11]. Bezzubikov, N. Belov, K. Batuzov. Automatic dynamic binary translator generation from instruction set description. In Proc. of the 2017 Ivannikov ISPRAS Open Conference, 2017, pp. 27-33.

[12]. ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile). URL https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf, accessed 20.11.2019.

[13]. Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In Proc. of Formal Methods in Computer-Aided Design, 2016, pp. 161-168.

[14]. I3S (Instruction Set Semantics Specification) Translator. URL https://github.com/ispras/I3S, accessed 20.11.2019.

[15]. Колтунов Д.С., Ефимов В.Ю., Падарян В.А. Автоматизированное тестирование фронтенда транслятора TCG для QEMU. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 7–24 / Koltunov D.S., Efimov V.Y., Padaryan V.A. Automated testing of a TCG frontend for QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 5, 2019 г., pp. 7-24 (in Russian). DOI: 10.15514/ISPRAS–2019–31(5)–1.

[16]. А.И. Аветисян, К.А. Батузов, В.Ю. Ефимов, В.А. Падарян, А.Ю. Тихонов. Применение программных эмуляторов для полносистемного анализа бинарного кода мобильных платформ. Проблемы информационной безопасности. Компьютерные системы, №4, 2015 г., стр. 187-194 / Avetisyan A.I., Batuzov K.A., Efimov V.Y., Padaryan V.A., Tikhonov A.Y. Whole system emulators for mobile platform binary code analysis. Information Security Problems. Computer Systems, №4, 2015, pp. 187-194 (in Russian).

[17]. V. Efimov, V. Padaryan. Peripheral Device Register Support for Source Code Boilerplate Generator of QEMU Development Toolkit. In Proc. of the 2018 Ivannikov Memorial Workshop (IVMEM), 2018, pp. 36-39.

[18]. S. Sargsyan, J. Hakobyan, M. Mehrabyan, M. Mishechkin, V. Akozin, S. Kurmangaleev. ISP-Fuzzer: Extendable Fuzzing Framework. In Proc. of the of 2019 Ivannikov Memorial Workshop (IVMEM), 2019, pp. 68-71.

[19]. А.Н. Федотов. Метод оценки эксплуатирумости программных дефектов. Труды ИСП РАН, том 28, вып. 4, 2016 г., стр. 137-148 / A.N. Fedotov. Method for exploitability estimation of program bugs. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016. pp. 137-148 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-8.

[20]. Падарян В.А. О представлении результатов обратной инженерии бинарного кода. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 31-42 / Padaryan V.A. Automated vulnerabilities exploitation in presence of modern defense mechanisms. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017. pp. 31-42 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-3

[21]. S. Horwitz, T. Reps, D. Binkley. Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, 1990, pp. 26-60.

[22]. J. Ferrante, K. J. Ottenstein, J. D. Warren. The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, 1987, pp. 319-349.

[23]. В. А. Падарян, М. А. Соловьев, А. И. Кононов. Моделирование операционной семантики машинных инструкций. Программирование, том 37, № 3, 2011 г., стр. 50-64 / V. A. Padaryan, M. A. Solov'ev, A. I. Kononov. Simulation of operational semantics of machine instructions. Programming and Computer Software, vol. 37, Issue 3, 2011, pp 161–170.

[24]. Соловьев М.А., Бакулин М.Г., Горбачев М.С., Манушин Д.В., Падарян В.А., Панасенко С.С. О новом поколении промежуточных представлений, применяемом для анализа бинарного кода. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 39-68 / Solovev M.A., Bakulin M.G., Gorbachev M.S., Manushin D.V., Padaryan V.A., Panasenko S.S. Next generation intermediate representations for binary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 39-68 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-3.

[25]. M. Jung, S. Kim, H. Han, J. Choi, S. Kil Cha. B2R2: Building an Efficient Front-End for Binary Analysis. In Proc. of the NDSS Workshop on Binary Analysis Research, 2019.

[26]. T. Dullien, S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In Proc. of the CanSecWest Applied Security Conference, 2009, 7 p.

[27]. S. Alam, R.N. Horspool, I. Traore. MAIL: Malware Analysis Intermediate Language: A Step Towards Automating and Optimizing Malware Detection. In Proc. of the 6th International Conference on Security of Information and Networks, 2013, pp. 233–240.

[28]. A formal specification for BIL: BIL Instruction Language, 2015. URL https://github.com/BinaryAnalysisPlatform/bil/blob/master/bil.tex, accessed 20.11.2019.

[29]. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. Lecture Notes in Computer Science, vol. 5352, 2008, pp. 1-25.

[30]. Sepp, B. Mihaila, A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In Proc. of the 18th Working Conference on Reverse Engineering, 2011, pp. 357-366.

[31]. N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentatio. SIGPLAN Notices, vol. 42, no. 6, 2007, pp. 89-100.

[32]. E. J. Schwartz, J. Lee, M. Woo, D. Brumley. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring. In Proc. of the 22Nd USENIX Conference on Security, 2013, pp. 353-368.

[33]. K. Yakdan, S. Dechand, E. Gerhards-Padilla, M. Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In Proc. of the IEEE Symposium on Security and Privacy (SP), 2016, pp. 158-177.

[34]. Retargetable Decompiler. URL https://retdec.com/, accessed 20.11.2019.

[35]. T. Ben-Nun, A. S. Jakobovits, T. Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics. In Proc. of the 32Nd International Conference on Neural Information Processing Systems, 2018, pp. 1-13.

[36]. O. Katz, Y. Olshaker, Y. Goldberg, E. Yahav. Towards Neural Decompilation. arXiv preprint arXiv:1905.08325, 2019.

[37]. P. Lestringant, F. Guihéry, P.-A. Fouque. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In Proc. of the 10th ACM Symposium on Information, Computer and Communications Security, 2015, pp. 203–214.

[38]. D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, D. Song. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. Lecture Notes in Computer Science, vol. 81-34, 2013, pp. 164-181.

[39]. Slowinska, T. Stancescu, H. Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In Proc. of the NDSS Symposium, 2011, 20 p.

[40]. Z. Lin, X. Zhang, D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In Proc. of the 11th Annual Information Security Symposium, 2010, Article no. 5.

[41]. G. Ramalingam, J. Field, F. Tip. Aggregate Structure Identification and Its Application to Program Analysis. In Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999. Pp. 119–132.

[42]. R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, vol. 17, 1978, pp. 348-375.

[43]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 277-296 / Dovgalyuk P.M., Makarov V.A., Padaryan V.A., Romaneev M.S., Fursova N.I. Application of software emulators for thebinary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol, 26, issue 1, 2014, pp. 277-296 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-9.

[44]. C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM SIGPLAN Notices, vol. 40, no. 6, 2005, pp. 190-200.

[45]. D. Bruening, S. Amarasinghe. Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[46]. V.A. Padaryan, I.N. Ledovskikh. On the Representation of Results of Binary Code Reverse Engineering. Programming and Computer Software, vol. 44, issue 3, 2018, pp. 200–206.

Информация об авторах / Information about authors

Александр Борисович БУГЕРЯ – кандидат физико-математических наук, старший научный сотрудник ИПМ им. М.В. Келдыша РАН, программист в ИСП РАН. Сфера научных интересов: методы и средства анализа бинарного кода программ, восстановление алгоритмов по бинарному коду, параллельное программирование: модели, языки, алгоритмы, инструментарий; распределенные системы, построение компиляторов, системное программирование.

Alexander Borisovich BUGERYA – PhD in Physical and Mathematical Sciences, Senior Researcher in Keldysh Institute of Applied Mathematics; Programmer at ISP RAS. His research interests include methods and tools for binary code analysis, recovery of program algorithm by its binary code, parallel programming: models, languages, algorithms and tools; distributed computing systems, compiler construction, system programming.

Василий Юрьевич ЕФИМОВ – младший научный сотрудник ИСП РАН. Область научных интересов: двоичная трансляция, программная эмуляция, генерация кода, инструментальные средства разработки программного обеспечения.

Vasilii Yur'evich EFIMOV – Junior Researcher at ISP RAS. Research interests: binary translation, software emulation, code generation, software development tools.

Иван Иванович КУЛАГИН – кандидат технических наук, инженер ИСП РАН. Область научных интересов: построение компиляторов, анализ бинарного кода, оптимизирующие компиляторы, полиэдральная компиляция, генерация кода, модели параллельного программирования.

Ivan Ivanovich KULAGIN – Candidate of Technical Science (PhD), Software Engineer at ISP RAS. Research interests: compiler construction, binary code analysis, compiler optimizations, polyhedral compilation, code generation, parallel programming models.

Вартан Андроникович ПАДАРЯН – кандидат физико-математических наук, ведущий научный сотрудник ИСП РАН, доцент кафедры системного программирования ВМК МГУ. Сфера научных интересов: компиляторные технологии, анализ бинарного кода, компьютерная безопасность, высокопроизводительные вычисления.

Vartan Andronikovich PADARYAN – Candidate of Physical and Mathematical Sciences, Leading Researcher at ISP RAS, Associate Professor of the Department of system programming of CMC of Lomonosov Moscow State University. Research Interests: compiler technologies, binary code analysis, cybersecurity, high performance calculating.

Михаил Александрович СОЛОВЬЕВ – кандидат физико-математических наук, старший научный сотрудник ИСП РАН, старший преподаватель кафедры системного программирования ВМК МГУ. Сфера научных интересов: абстрактная интерпретация, промежуточные представления, анализ бинарного кода, архитектура микропроцессоров.

Mikhail Aleksandrovich SOLOVEV – Candidate of Physical and Mathematical Sciences, senior researcher at ISP RAS, senior lecturer of the Department of system programming of CMC of Lomonosov Moscow State University. Research Interests: abstract interpretation, intermediate representations, binary code analysis, microprocessor architecture.

Андрей Юрьевич ТИХОНОВ – преподаватель МГТУ им. Н.Э. Баумана. Сфера научных интересов: компиляторные технологии, программная эмуляция, сетевые технологии.

Andrei Yur'evich TIKHONOV – lecturer at BMSTU. Research Interests: compiler technologies, software emulation, network technologies.