Федеральное государственное бюджетное учреждение науки
Институт системного программирования им. В.П. Иванникова РАН

На правах рукописи

Каушан Вадим Владимирович

Поиск ошибок выхода за границы буфера в бинарном коде программ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
кандидат физико-математических наук
Падарян Вартан Андроникович

Москва — 2017

Оглавление

Стр.

Стр.

Введение

В настоящее время особенно остро стоит задача обеспечения безопасности информационных систем. Наиболее частой причиной нарушения безопасности в таких системах являются уязвимости в программном обеспечении этих систем, позволяющие нарушить конфиденциальность, доступность или целостность обрабатываемой информации. В связи с этим актуальной является задача поиска ошибок и уязвимостей в программном обеспечении.

Несмотря на возрастающую популярность веб-приложений, реализуемых на различных скриптовых языках, большая доля программного обеспечения представляет собой программы, компилируемые в бинарный код. Многие программы написаны на языках Си и C++, которые не обеспечивают безопасность доступа к памяти, что приводит к различным ошибкам доступа к памяти, которые, в свою очередь, могут стать основой для серьёзных уязвимостей, таких как переполнение буфера и использование памяти после её освобождения. Несмотря на то, что подобные ошибки относятся к исходному коду, эксплуатируемость соответствующих уязвимостей зависит от многих факторов, таких как используемый компилятор, набор опций компиляции, наличие механизмов защиты. Кроме того, некоторые ошибки можно обнаружить только в бинарном коде из-за того, что компилятор в результате оптимизаций может порождать не тот код, который от него ожидает разработчик. Поэтому для поиска уязвимостей необходимо анализировать непосредственно бинарный код программ. Кроме того, включаемые в поставляемое ПО библиотеки зачастую доступны только в исполняемых кодах.

Одним из наиболее распространённых типов уязвимостей является уязвимость переполнения буфера, уступающая по распространённости лишь XSS и SQL-инъекциям, относящимся к уязвимостям более высокого уровня. Эксплуатация этого типа уязвимости может привести с самым различным последствиям: от утечки данных до выполнения произвольного вредоносного кода в рамках уязвимого приложения и последующей компрометации системы (рисунок 0.1). Уязвимость «Heartbleed» в OpenSSL продемонстрировала, что большую опасность может представлять не только запись за границы буфера, но и чтение. Таким образом, поиск ошибок работы с буферами в памяти является актуальной задачей.

Существующие инструменты поиска ошибок позволяют находить ошибки выхода за границы буфера как в исходном, так и в бинарном коде. К сожалению,

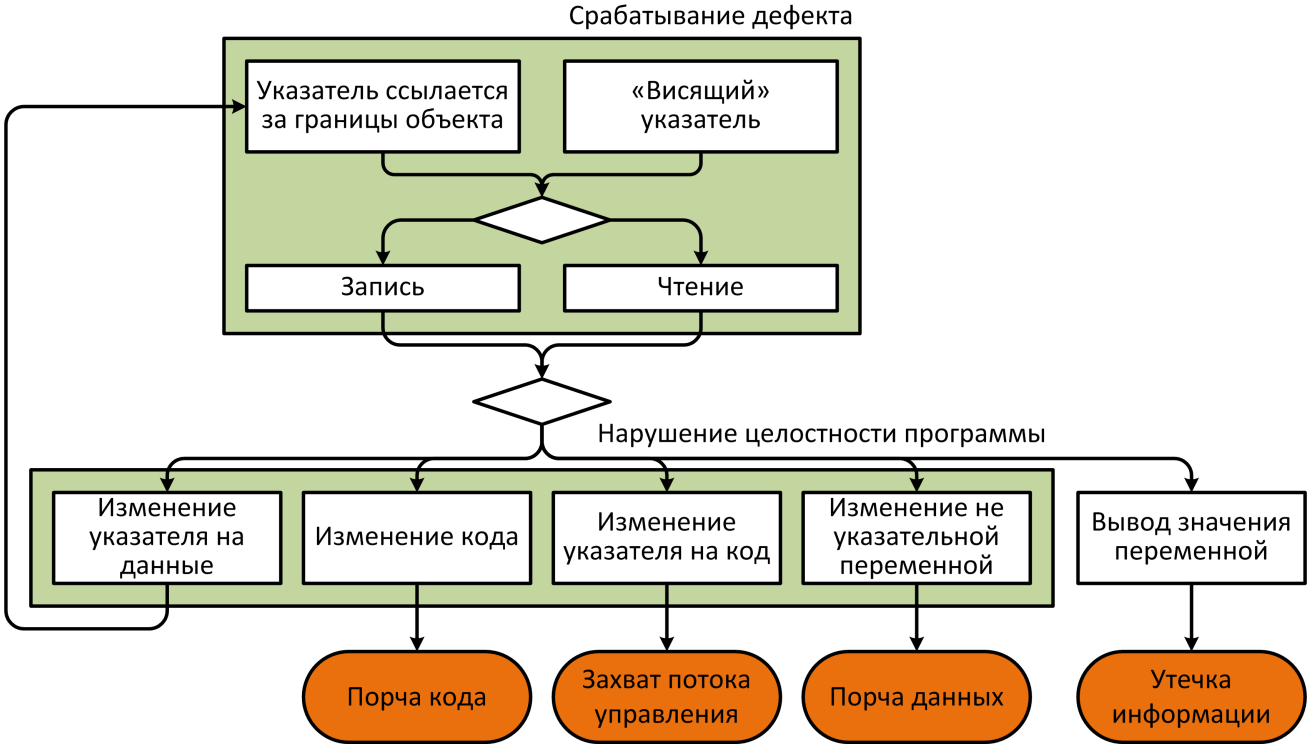

Рисунок 0.1 — Последствия срабатывания дефекта.

многие инструменты, анализирующие исходный код, не способны предоставить набор входных данных, приводящий к ошибке, что не позволяет оценить критичность найденных ошибок. В свою очередь, при анализе бинарного кода часто требуется наличие отладочной информации, которая может быть недоступна. Кроме того, при динамическом анализе бинарного кода вырабатывается набор входных данных, подтверждающих найденную ошибку, но большинство инструментов анализа позволяют находить ошибки выхода за границы буфера только по факту их срабатывания. Наибольший интерес представляет возможность целенаправленного поиска таких ошибок, а также возможность анализа в отсутствии отладочной информации.

При целенаправленном поиске ошибок выхода за границы буфера остро стоит задача анализа циклов обработки данных. Среди существующих подходов, позволяющих находить такого рода ошибки с учётом работы циклов, можно отметить работы учёных П. Годефруа, Д. Лучауп, Р. Майумдар и Р. Зу, где описываются идеи и методы, на основе которых были реализованы инструменты Splat и SAGE. В инструменте SAGE реализован подход, позволяющий влиять на число итераций некоторых циклов в программе таким образом, чтобы обработка данных в этих циклах привела к выходу за границы буфера. В инструменте

Splat реализован подход, основанный на символьной интерпретации длин обрабатываемых буферов. Этот инструмент работает с исходным кодом программ и позволяет подбирать такой размер данных, обработка которого приводит к переполнению буфера, выделенного в программе. К сожалению, в работе, описывающей инструмент Splat, не были предложены методы, позволяющие символьно анализировать длины буферов по бинарному коду, что существенно ограничивает область применимости предложенного подхода. В данной работе восполняется этот недостаток и описываются методы символьной интерпретации длин буферов по бинарному коду. Кроме того, предлагаются различные способы интерпретации как библиотечных функций, так и циклов с целью сокращения числа рассматриваемых состояний программы.

При анализе бинарного кода основной сложностью является формальное описание ошибки. В бинарном коде отсутствует информация о типах переменных, что затрудняет описание таких понятий, как выход за пределы буфера и самих буферов в памяти. Кроме того, для поиска ошибок необходим анализ потока данных, который может быть затруднён оптимизирующими преобразованиями со стороны компилятора. В современных инструментах поиска ошибок эта задача решается с помощью символьного выполнения, позволяющего представить действия, выполняющиеся в программе, с помощью набора уравнений для SMT-решателя. Большинство инструментов, использующих символьное выполнение по бинарному коду программ, применяет динамический анализ бинарного кода, при котором анализируется один или несколько путей выполнения программы. В процессе анализа производится построение предиката пути – набора уравнений, описывающих условия прохождения выполнения по этому пути. К предикату пути добавляется предикат безопасности – уравнения, описывающие проявление ошибки в коде программы. Если полученная система совместна, решением этой системы будут значения символьных переменных, при которых происходит ошибка.

Важной задачей является обеспечение универсальности алгоритмов поиска ошибок. Для расширения применимости разработанных методов динамического анализа на разные классы ПО и вычислительных устройств, они не должны зависеть от анализируемой процессорной архитектуры, а также от операционной системы, которая используется для запуска анализируемого приложения. Абстрагирование от процессорной архитектуры традиционно достигается за счёт использования промежуточного представления, позволяющего описывать операционную семантику машинных инструкций. Для обеспечения независимости

от используемой операционной системы часто используют анализ на уровне полносистемного эмулятора. Абстрагирование от архитектуры и операционной системы также позволяет анализировать различные классы программного обеспечения: не только ПО для персональных компьютеров, но и встроенное программное обеспечение маршрутизаторов, смартфонов и различных устройств, составляющих «интернет вещей». К сожалению, при использовании эмуляторов реализация сложных алгоритмов анализа значительно замедляет эмуляцию, что приводит к ощутимому влиянию на работу операционной системы. Замедление может привести к самым разным последствиям: от обрыва сетевых соединений по таймауту до полной неработоспособности пользовательских приложений. Кроме того, анализ на уровне эмулятора усложняет получение статической информации об исследуемой программе, поскольку её накопление в реальном времени требует значительных объёмов памяти. Для решения этих проблем используют post-mortem анализ, который позволяет проводить анализ после выполнения программы. Такой анализ становится возможным за счёт применения методов детерминированного воспроизведения либо трассировки.

Таким образом, актуальной является задача поиска ошибок выхода за границы буфера в бинарном коде программ без отладочной информации, а также обеспечение универсальности алгоритмов поиска таких ошибок.

Целью диссертационной работы является исследование и разработка методов автоматизации поиска ошибок выхода за границы буфера в бинарном коде программ по трассам выполнения. Разработанные методы не должны зависеть от целевой процессорной архитектуры, а также от операционной системы, используемой для запуска анализируемой программы.

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

1. Исследовать имеющиеся подходы к поиску ошибок выхода за границы буфера и оценить применимость предлагаемых идей к анализу бинарного кода.
2. Разработать архитектурно-независимый метод, позволяющий производить символьную интерпретацию трасс выполнения с использованием абстрактной длины обрабатываемых переменных-массивов.
3. Разработать архитектурно-независимый метод определения границ буферов, обрабатываемых в исследуемых программах.

4. Разработать архитектурно-независимый метод, позволяющий находить ошибки выхода за границы буфера на основе анализа длин обрабатываемых переменных-массивов во время символьной интерпретации.
5. Разработать архитектурно-независимый метод, позволяющий восстанавливать функции работы со строками, которые подверглись встраиванию в процессе компиляции.
6. Разработать метод расширения покрытия кода исследуемой программы для увеличения вероятности обнаружения ошибочных ситуаций.

Научная новизна:

1. Разработан метод символьной интерпретации трасс выполнения с использованием абстрактной длины переменных-массивов, полученных по результатам обратной инженерии бинарного кода. Метод не требует наличия исходных кодов и отладочной информации.
2. На основе метода символьной интерпретации трасс разработан метод поиска ошибок выхода за границы буфера. Метод позволяет находить ошибки, даже если они не проявлялись в анализируемой трассе.

Практическая ценность работы. Предложенные методы поиска ошибок выхода за границы буфера реализованы в рамках среды динамического анализа бинарного кода. Использование промежуточного представления при анализе бинарного кода позволяет искать ошибки в программах для широкого множества процессорных архитектур. Это существенно отличает разработанный инструмент от других инструментов, в которых поддержка часто ограничена процессорными архитектурами x86 и x86-64, а также одной из распространённых операционных систем (Windows либо Linux). Разработанный инструмент используется в образовательном процессе для обучения студентов ФУПМ МФТИ и ВМК МГУ. Полученные научные результаты могут использоваться для развития инструментов поиска ошибок, применяющихся для промышленной разработки, а также для сертификации программного обеспечения.

Методология и методы исследования. Результаты диссертации были получены с использованием методов символьной интерпретации. Математическую основу данной работы составляют теория множеств, математическая логика и теория алгоритмов.

Основные положения, выносимые на защиту:

1. Разработан метод поиска ошибок выхода за границы буфера на основе символьной интерпретации трассы с использованием абстрактной

длины переменных-массивов, полученных по результатам обратной инженерии бинарного кода. Метод не требует наличия исходных кодов и отладочной информации и позволяет находить ошибки, даже если они не проявлялись в анализируемой трассе.

2. Разработаны методы, улучшающие точность поиска ошибок выхода за границы буфера за счёт выявления функций работы со строками, которые подверглись встраиванию в процессе компиляции и предварительного расширения покрытия кода при сборе трассы выполнения.
3. На основе предложенных методов разработано и реализовано программное средство для поиска ошибок выхода за границы буфера. Реализованные методы являются машинно-независимыми, а также абстрагированы от операционной системы, используемой для запуска анализируемой программы. Для определения границ буферов используется алгоритм, позволяющий получить оценочную информацию о границах буферов на основе анализа областей динамической и автоматической памяти.

Апробация работы. Основные результаты работы докладывались на следующих конференциях.

1. IV международный форум по практической безопасности «Positive Hack Days». Москва, 21 — 22 мая 2014.
2. 24-й научно-техническая конференция «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 29 июня - 02 июля 2015.
3. Открытая конференция ИСП РАН. Москва, 01 - 02 декабря 2016.

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях [1–6], 4 из которых изданы в журналах, рекомендованных ВАК [1–4], 2 – в тезисах докладов [5;6]. Работа [1] индексирована Scopus и Web of Science. Вклад автора в работах [1; 6] заключается в разработке базового метода символьной интерпретации трасс выполнения. В работах [2; 3; 5] вклад автора состоит в разработке метода символьной интерпретации трассы с учётом символьных длин буферов, а также метода поиска ошибок выхода за границы буфера. В работе [4] вклад автора состоит в описании операционной семантики машинных инструкций различных процессорных архитектур в рамках промежуточного представления Pivot, используемого в данной работе.

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Объём и структура работы. Диссертация состоит из введения, четырёх глав, заключения и одного приложения. Полный объём диссертации составляет 92 страницы, включая 5 рисунков и 2 таблицы. Список литературы содержит 55 наименований.

Глава 1. Подходы к поиску ошибок выхода за границы буфера

В общем случае при решении задачи в качестве входных данных может быть доступен исходный или бинарный код программы. Также для поиска ошибок может применяться как статический анализ кода, так и динамический анализ. Среди подходов, основанных на динамическом анализе кода, можно выделить инструментацию и динамическое символьное выполнение.

1.1 Статический анализ кода

В рамках статического анализа различают два класса ошибок, приводящих к выходу за границы буфера: ошибки, возникающие в процессе поэлементной обработки буферов и ошибки, возникающие при вызове функций, обрабатывающих буфер целиком (таких как `strcpy`, `strcat`, `memcpy`). Ошибки из первого класса чаще всего происходят в результате обработки буферов внутри циклов. Поиск таких ошибок требует анализа циклов, что является сложной задачей для статического анализа. Для анализа циклов чаще всего используется анализ первых нескольких итераций цикла, что затрудняет обнаружение ошибки, поскольку ошибочная ситуация может возникнуть спустя сотни или тысячи итераций.

Подходы к поиску ошибок выхода за границы буфера с помощью статического анализа можно разбить на несколько классов:

- анализ пометок данных;
- ограничения;
- аннотации;
- поиск характерных шаблонов кода.

Анализ на основе пометок данных использует пометки для поиска ошибочных ситуаций. Изначально все данные, полученные из недоверенных источников, помечаются и эти пометки распространяются в соответствии с потоком данных в программе. Если помеченные данные используются в качестве аргументов уязвимых библиотечных функций, вызовы этих функций помечаются как потенциально приводящие к ошибке.

Анализ на основе ограничений позволяет описывать ограничения безопасности, нарушение которых приводит к возникновению ошибок. Ограничения могут генерироваться на основе операторов программы и вызовов библиотечных функций. Также эти ограничения могут распространяться и обновляться в ходе анализа программы. Анализатор проверяет существование решения для данных ограничений, которое указывает на наличие ошибки. Методы, использующие анализ ограничений, можно разделить на две категории: анализ числовых диапазонов и символьное выполнение.

При анализе числовых диапазонов анализируются операторы объявления буферов и операторы, работающие с этими буферами. Ограничения описываются в виде пары числовых диапазонов: выделенный размер буфера и текущий размер буфера. Для каждого буфера решается набор ограничений и результат представляется в виде числовых диапазонов для выделенного размера буфера и его фактического размера. Если обнаруживается ситуация, когда нижняя граница фактического размера буфера может быть больше верхней границы выделенного размера, фиксируется наличие ошибки переполнения буфера. Совместно с этим подходом может применяться анализ, чувствительный к потоку, что позволяет уменьшить уровень ложноположительных срабатываний.

Методы, основанные на символьном выполнении, сопоставляют переменным программы некоторые символьные значения. Программа анализируется с помощью прохода по графу потока управления. При анализе потенциально опасных операций, таких как доступ к буферу, обращение по указателю или вызов библиотечной функции, генерируются ограничения безопасности, которые описывают ошибочную ситуацию. Затем проверяется совместность этих ограничений с уже накопленными ограничениями пути с помощью решателя ограничений. Если совместность удаётся доказать, фиксируется предупреждение об ошибке. Следует отметить, что система ограничений, генерируемая по статическому представлению программы, слишком сложна, так как должна описывать одновременно все возможные пути в программе. Это приводит к ограниченной применимости символьного выполнения для статического анализа кода. Несмотря на это, похожие идеи успешно применяются в различных подходах на основе динамического анализа.

Анализ на основе аннотаций позволяет описывать аннотации для функций в терминах пред- и пост-условий. В процессе анализа проверяется возможность безопасного использования буферов в соответствии с условиями, описанными в

аннотациях. Аннотации описываются программистом на специальном языке или средствами того же языка программирования, на котором написана сама программа. Так, например, инструмент Splint анализирует аннотации, описанные в виде специальных комментариев в программе на языке Си.

Анализ на основе поиска характерных шаблонов кода использует представление программы на уровне AST для поиска ошибок. С помощью алгоритмов поиска по шаблону детектируются характерные последовательности операций, потенциально приводящие к ошибкам.

1.2 Инструментация

Инструментация позволяет обнаруживать ошибки в программном коде, запущенном на исполнение. При этом аналитик имеет возможность наблюдать или диагностировать поведение приложения во время его исполнения, в идеальном случае – непосредственно в целевой среде.

При инструментации производится модификация исходного или бинарного кода приложения с целью установления процедур-перехватчиков для проведения инструментальных измерений. С помощью таких «ловушек» можно обнаружить программные ошибки на этапе выполнения, проанализировать использование памяти, покрытие кода и проверить другие условия.

Ошибки, найденные в процессе инструментации, почти всегда являются настоящими ошибками, которые программист может быстро идентифицировать и исправить. Следует заметить, что для создания ошибочной ситуации на этапе выполнения должны существовать точно необходимые условия, при которых проявляется программная ошибка. Соответственно, разработчики должны создать некоторый контрольный пример для реализации конкретного сценария.

Инструментация может проводиться как на этапе выполнения анализируемой программы (динамическая инструментация), так и во время компиляции (статическая инструментация). К средам динамической инструментации можно отнести Valgrind, DynamoRIO и Pin. Наиболее распространённым применением статической инструментации является профилирование, но также данный вид инструментации нашёл применение во встроенных детекторах (sanitizer) ошибок компилятора clang. Эти детекторы используют информацию из исходного кода

программы для добавления проверок на различные ошибочные ситуации, такие как выход за границы буфера, разыменование нулевого указателя или некорректное использование операций выделения и освобождения памяти.

1.2.1 Среда анализа Valgrind

Valgrind [7; 8] – это система динамического профилирования с последующим анализом для платформы x86/Linux. Является свободно распространяемым ПО. Анализ проводится по следующей схеме. Анализируемая программа не выполняется непосредственно, вместо этого она динамически транслируется в промежуточное представление Valgrind VEX (далее IR). Промежуточное представление не зависит от целевой платформы и представлено в SSA-виде. Разработанные на базе Valgrind инструменты выполняют необходимые преобразования над IR, после чего Valgrind транслирует IR обратно в машинный код.

Каждый исполняемый базовый блок транслируется в IR. Трансляция разбивается на пять этапов.

1. Исполняемый код представляется в виде двухадресных инструкций Valgrind. При этом используются виртуальные регистры.
2. Оптимизация IR.
3. Внесение инструментального кода. Инструменты, выбранные для анализа, добавляют необходимый код в промежуточное представление программы.
4. Распределение регистров. Виртуальные регистры, используемые в коде IR, отображаются на 6 аппаратных регистров: eax, ebx, ecx, edx, esi, edi. Регистры ebp и esp зарезервированы: на ebp хранится адрес текущего базового блока, на esp – адрес стека Valgrind.
5. Генерация кода. Каждая инструкция IR независимо от других конвертируется в одну или несколько инструкций целевой архитектуры. При этом счётчик инструкций должным образом модифицируется.

Стоит отметить, что Valgrind поддерживает FP- и SIMD-инструкции. Valgrind не может транслировать код ядра ОС, поэтому системные функции выполняются непосредственно, без внесения в код каких-либо изменений. Используется следующий алгоритм:

1. Сохранение стека анализируемой программы.
2. Копирование значений из памяти на аппаратные регистры (за исключением счетчика команд).
3. Исполнение системной функции.
4. Копирование значений аппаратных регистров после системного вызова в память (за исключением счётчика команд).
5. Восстановление стека анализируемой программы.

Valgrind поддерживает обработку сигналов. Valgrind не позволяет анализировать самомодифицирующийся код. Собственно анализ кода выполняется подключаемыми к Valgrind модулями-расширениями. В каждом модуле должны определяться как минимум четыре функции: функции инициализации (одна запускается до обработки параметров, вторая – после), функция добавления инструментального кода в базовый блок, функция для проведения заключительных этапов анализа (например, для сохранения результатов).

Код для проведения анализа может быть добавлен тремя способами. Первый способ – встроить анализирующий код непосредственно в промежуточное представление. Второй способ – добавить вызов ассемблерной подпрограммы, определённой в коде инструмента, посредством инструкции IR CALLM. Третий способ – добавить вызов функции языка Си, определённой в коде инструмента, посредством инструкции IR CCALL.

Наиболее популярным инструментом Valgrind является модуль memcheck. Он предназначен для выявления ошибок при работе с памятью в программах языков Си/Си++.

- С каждым байтом памяти связывается бит *адресуемости* (addressability bit, A). Он определяет возможность обращения к соответствующему байту.
- С каждым байтом регистров и каждым байтом памяти связываются 8 бит *корректности* (validity bits, V). Определяют, были ли проинициализированы соответствующие биты в байте.
- Для каждого блока динамически выделенной памяти сохраняется его адрес, а также функция (malloc()/calloc()/realloc(), new, new[]), при помощи которой блок был выделен.

По ходу выполнения программы информация об используемой памяти модифицируется: Valgrind заменяет вызовы функций malloc(), memcpy(), strcpy(), strcat() на вызовы аналогичных функций, сохраняющих информацию об используемых участках памяти.

С помощью модуля memcheck выявляются следующие ошибки:

– попытка использования неинициализированной памяти;
– чтение/запись в память после её освобождения;
– чтение/запись с конца выделенного блока;
– утечки памяти.

Memcheck неспособен обнаруживать ошибки при использовании статических или помещённых на стек данных. Подобные ошибки могут быть обнаружены модулем Ptrcheck для Valgrind.

Прочие модули Valgrind:

– **Massif** [9] – профилировщик кучи памяти (детальное описание изменений кучи за счёт создания снапшотов; построение графа, демонстрирующего использование кучи в процессе выполнения анализируемой программы),
– **Helgrind** [10], **DRD** [11] – инструменты для детектирования ошибок синхронизации в многопоточных приложениях (языки Си/Си++),
– **Cachegrind** [12] – профилировщик кэш памяти (L1, D1, L2; указываются участки кода, в которых обнаружены кэш-промахи),
– **Redux** [13] – построение динамических графов потоков данных (DDFG), отражающих процесс получения любого значения в каждой точке программы,
– **TaintCheck** [14] – анализ «помеченных» данных, в частности, отслеживание потоков данных, полученных из внешних источников (сетевые пакеты, ввод с клавиатуры).

Основным недостатком Valgrind является поддержка только ОС Linux для архитектуры x86.

1.2.2 Среда анализа DynamoRIO

DynamoRIO [15] – это среда анализа, которая позволяет выполнять преобразования кода любой части программы во время её выполнения. DynamoRIO предоставляет интерфейс для построения инструментов динамического анализа для широкого спектра использования: анализ программ и алгоритмов, профилировка, инструментация, оптимизация, трансляция и другие. В отличие от других

систем инструментации, DynamoRIO не ограничена лишь вставкой трамплинов и вызовов и позволяет проводить произвольную модификацию инструкций программы. DynamoRIO позволяет проводить эффективные и прозрачные для программы манипуляции над программами, выполняющимися под ОС Windows, Linux и Android, а также поддерживает архитектуры IA-32, AMD64, ARM и AArch64.

На базе DynamoRIO построено множество инструментов анализа, из которых наибольший интерес представляет инструмент Dr. Memory [16]. Этот инструмент позволяет обнаруживать ошибки работы с памятью, такие как обращения к неинициализированной или неадресуемой памяти (включая невыделенные области динамической памяти, а также выход за границы выделенной области памяти), обращения к освобождённой памяти, двойное освобождение памяти, утечки памяти и (для ОС Windows) дескрипторов, а также доступ к незанятым слотам локального хранилища потока.

Для обнаружения ошибок доступа к памяти в Dr. Memory поддерживается теневая карта памяти. Каждому байту памяти сопоставляются метаданные, отражающие одно из трёх состояний:

- *неадресуемая* память – память, к которой не должна обращаться программа;
- *неинициализированная* память – адресуемая память, для которой не было операций записи с момента выделения памяти;
- *инициализированная* память – адресуемая память, которая была инициализирована.

Понятие адресуемости используется более строгое, чем то, которое определяется используемой операционной системой или процессором. Неадресуемыми считаются не только невалидные страницы памяти, исследуется также структура стека и кучи: память за вершиной стека считается неадресуемой, также как и динамическая память за пределами областей, выделенных программе аллокатором.

Метаданные из теневой карты памяти используются для обнаружения ошибок доступа к памяти. Чтение или запись неадресуемой памяти трактуется как ошибка. Анализ чтения неинициализированных данных более сложное из-за того, что программа может загружать из памяти машинное слово целиком, даже если требуется чтение только одного байта. Если выдавать ошибку на каждое такое обращение к неинициализированным данным, это приведёт к большому

количеству ложных срабатываний. Для решения этой проблемы пометки неинициализированной памяти распространяются вместе с загруженными значениями, что требует поддерживания теневой карты не только для памяти, но и для регистров. Выдача ошибок происходит только если выполняется *существенное* обращение к данным, такое как сравнение значений или передача данных в качестве параметра системному вызову.

Анализ выделенных областей динамической памяти происходит путём перехвата функций выделения и освобождения памяти. Распространение метаданных происходит во время анализа отдельных инструкций исследуемой программы. В случае, когда инструкция (например, выполняющая арифметическую операцию) работает одновременно с инициализированным и неинициализированным значением, результатом её выполнения считается неинициализированное значение.

1.2.3 Среда анализа Pin

Pin [17; 18] – это система для проведения динамического анализа посредством добавления инструментального кода в исполняемые файлы. Поддерживаются следующие архитектуры – x64, x86_64, IA-64. Pin предоставляет пользователю API для разработки собственных инструментов анализа. Инструментальный код добавляется в запущенное приложение посредством JIT-компиляции – блоки инструкций перекомпилируются и инструментируются непосредственно перед выполнением. Модифицированные инструкции записываются в программный кэш. Перекомпиляция вносит существенное замедление в работу анализируемой программы (время выполнения тестов из набора SPEC [19] (целочисленная арифметика) в среднем увеличивается в два раза). Существует возможность внесения инструментального кода непосредственно в исполняемый файл, загруженный в память (probe mode) – при этом доступна лишь часть API, однако накладных расходов практически не возникает.

Архитектура системы Pin включает в себя запускающий процесс, процесс-сервер и процесс анализируемой программы. Запускающий модуль создаёт процесс анализируемой программы и процесс-сервер, а также производит

внедрение библиотек Pin. Процесс-сервер осуществляет взаимодействие с анализируемой программой посредством механизма разделяемой памяти. В адресное пространство процесса анализируемой программы дополнительно загружаются библиотеки инструментов Pin, а также pinvm.dll. Библиотека pinvm.dll включает в себя модуль-монитор (Virtual Machine Monitor, VMM), а также API для инструментов Pin. Модуль-монитор (JIT-компилятор, диспетчеры, компонента взаимодействия с ОС) управляет выполнением анализируемой программы и добавляет инструментальный код.

Pin контролирует выполнение программы в режиме пользователя, но не в режиме ядра ОС. Для того, чтобы по завершении системного вызова управление вновь попало в модуль-монитор, необходим специальный обработчик. Перехват системного вызова осуществляется при выполнении инструкции, передающей управление из режима пользователя в режим ядра. В 32-битной ОС Windows используются инструкции sysenter и int 2e, в 64-битной – syscall. В 32-битных приложениях, запущенных в 64-битной ОС Windows, для перехода управления в режим ядра выполняется инструкция jmp far. Для каждого отслеживаемого Pin системного вызова в ntdll.dll существует функция-обёртка; она записывает на регистр номер вызова и осуществляет сам вызов. Pin сохраняет номер системного вызова, а также расположение его аргументов относительно вершины стека.

Некоторые системные вызовы требуют дополнительной обработки Pin (например, если анализируемое приложение запускает процесс-потомок, Pin необходимо создать соответствующий объект-описатель), однако большинство вызовов выполняются ”как есть”.

Для обработки исключений, возникших в режиме пользователя, в ОС Windows используется процедура KiUserExceptionDispatcher библиотеки ntdll.dll. Перехватив вызов этой функции, Pin вызывает собственный обработчик. В первую очередь выполняется проверка того, где возникло исключение – в коде анализируемой программы или в коде Pin. Для аппаратного исключения восстанавливается исходный контекст памяти, после чего выполняется транслированный код обработчика. Для программного исключения восстанавливать контекст памяти не нужно, поскольку параметры исключения были установлены в режиме пользователя.

В Pin поддерживается анализ многопоточных приложений, в рамках которого отслеживается создание потоков и их взаимодействие. Реализованы функции-обёртки для соответствующих механизмов синхронизации ОС Windows.

Помимо реализации Pin для ОС Windows, описанной выше, существует реализация для ОС Linux, а также поддержка архитектуры ARM под управлением ОС Linux.

Наиболее известным инструментом поиска ошибок, использующим инструментацию Pin, является система Mayhem [20].

1.2.4 Встроенные детекторы компилятора clang

Компилятор clang [21; 22] предоставляет возможность использования различных встроенных детекторов (sanitizers) во время компиляции. Эти детекторы используют информацию из исходного кода программы для добавления в компилируемый код проверок на различные ошибочные ситуации. На данный момент поддерживаются следующие детекторы:

- **AddressSanitizer** [23] – обнаружение различных ошибок доступа к памяти (выход за границы буфера, use-after-free, double-free, утечки памяти).
- **ThreadSanitizer** [24] – обнаружение состояний гонки.
- **MemorySanitizer** [25] – обнаружение доступа к неинициализированным переменным, доступа к объектам после их уничтожения.
- **UndefinedBehaviorSanitizer** [26] – обнаружение ситуаций в коде, приводящих к неопределённому поведению (знаковое целочисленное переполнение, преобразования значений с плавающей точкой, приводящих к округлению и другие).
- **DataFlowSanitizer** [27] – обнаружение различных ошибок утечки чувствительных данных с помощью анализа помеченных данных.
- **LeakSanitizer** [28] – обнаружение утечек памяти.
- **SafeStack** [29] – обнаружение ошибок переполнения буфера на стеке.

Детектор SafeStack разделяет область стека в программе на две различных области: безопасный и небезопасный стек. Область безопасного стека хранит адреса возврата из функций, сохранённые регистры и локальные переменные, доступ к которым должен осуществляться безопасно, а область небезопасного стека содержит всё остальное. Такое разделение позволяет получить уверенность в том,

что переполнение буфера в области небезопасного стека не может привести к перезаписи чего-либо в области безопасного стека. Данная инструментация вносит очень незначительное (порядка 0.1%) замедление в работу программы.

Детектор AddressSanitizer позволяет обнаруживать ошибки выхода за границы буфера не только на стеке, но также в области статической и динамической памяти. Среднее замедление, вносимое инструментацией, составляет в среднем 73% для тестового набора SPEC 2006 [19]. Если инструментировать только операции записи, замедление составляет в среднем 26%.

1.3 Динамическое символьное выполнение

Идея символьного выполнения заключается в замене конкретных значений переменных в программе символьными значениями, при этом конкретное выполнение операций заменяется на формулы, отражающие эффект выполнения этих операций. Операции в программе интерпретируются одна за другой, порождая новые символьные значения и соответствующие им формульные выражения. Если во время выполнения встречается ветвление, выполнение продолжается по всем возможным веткам ветвления, при этом на каждой из веток добавляется уравнение, отражающее условие перехода на соответствующую ветку. Набор таких уравнений описывает прохождение некоторого пути в программе и называется предикатом пути. Решение этого набора уравнений даёт набор значений для переменных в программе, при которых программа пройдёт по этому же пути.

Системы поиска ошибок, основанные на динамическом символьном выполнении выполняют перебор путей в программе и проверку различных ошибочных ситуаций. К предикату пути добавляются уравнения, называемые предикатом безопасности, которые описывают ошибочную ситуацию (например, разыменование нулевого указателя) и если полученная система оказалась совместной, полученный набор значений переменных является подтверждением найденной ошибки. Чаще всего применяется смешанное выполнение, представляющее собой сочетание конкретного и символьного выполнения. При этом символьные значения ставятся в соответствие лишь некоторому подмножеству переменных программы, а операции, не использующие символьные значения, выполняются конкретно.

Обычно символьные значения ставятся в соответствие данным, которые программа получает на вход.

Существует два подхода к проведению символьного выполнения: онлайн и оффлайн выполнение. При оффлайн выполнении программа выполняется с самого начала для каждого пути, а при онлайн выполнении сохраняются промежуточные состояния программы, что позволяет ускорить выполнение. Недостатком онлайн выполнения является повышенные требования к оперативной памяти, которая расходуется на хранение промежуточных состояний.

Классическим примером описанного выше подхода является инструмент EXE [30]. Он позволяет перебирать пути выполнения в программах на языке Си и генерировать наборы входных данных, приводящие к аварийным ситуациям. Аналогичным образом работает инструмент KLEE, который анализирует программы на языках Си/C++ с использованием промежуточного представления LLVM. Инструменты Avalanche [31], SAGE [32] и Mayhem [20] также направлены на поиск ошибок в программах, но работают уже с бинарным кодом.

1.3.1 Виртуальная машина KLEE

KLEE [33] – это виртуальная машина на базе компиляторной инфраструктуры LLVM с возможностью смешанного выполнения. Цель – обеспечить максимальное покрытие кода приложения и выявить возможные ошибки и уязвимости. В рамках проводимого анализа параллельно выполняются символические процессы (states), в каждом из которых реализуется одна траектория CFG программы. Каждая траектория характеризуется уникальным набором входных данных. Эффективная реализация создания процессов-потомков на каждом ветвлении программы позволяет анализировать большое количество путей одновременно.

Большая часть инструкций выполняется «естественным» образом. В качестве примера рассмотрим команду add. Если хотя бы один из операндов является символьным выражением, создаётся символьное выражение для суммы, и результат записывается на регистр.

Результат выполнения инструкции-ветвления зависит от истинности условного перехода. Проверка истинности осуществляется посредством STP [34]. Если

условие перехода зависит от символьных данных, процесс клонируется, после чего анализируются обе траектории. При этом соответствующим образом изменяются наборы наложенных на траектории ограничений.

Если значение указателя является символьным выражением и выполняется его разыменование, то соответствующий процесс клонируется столько раз, сколько конкретных значений может принимать символьное выражение.

Реализовано два алгоритма выбора траекторий для дальнейшего анализа. Первый задает траекторию случайным образом. Второй алгоритм выбирает траекторию, которая с наибольшей вероятностью покроет ранее не исполнявшийся код (используются эвристики).

В KLEE к STP-запросам применяются оптимизации, позволяющие уменьшить время обработки. Первая – это декомпозиция запроса на непересекающиеся подзапросы (в соответствии с используемыми символьными выражениями). Вторая оптимизация заключается в кэшировании карты, которая каждому выполненному запросу ставит в соответствие его результат. В карту попадают только те запросы, для которых STP генерирует контрпример. При обработке очередного запроса в карте ищутся подмножества и надмножества данного набора ограничений. Используются следующие соображения:

1. Если некоторое подмножество набора ограничений противоречиво, этот набор противоречив.
2. Если некоторое надмножество набора ограничений имеет решение, набор разрешим.
3. Если некоторое подмножество набора ограничений имеет решение, велика вероятность того, что набор разрешим.

Взаимодействие программы с окружением (ОС, пользователь) осуществляется разными способами: чтение из файла и запись в файл, отправка и получение сетевых пакетов, использование командной строки и системных переменных. В результате чтения данных из файла появляются символьные данные, для которых нет никаких ограничений. Для обработки подобных случаев аналитик добавляет свой код (на языке Си) в модель KLEE.

Отличительной особенностью KLEE является поддержка «символьной» файловой системы, состоящей из одной директории и нескольких символьных файлов. Количество и максимальный размер этих файлов задаётся в качестве параметров.

1.3.2 Среда S^{2}E

Среда символьного выполнения S^{2}E [35] позволяет проводить символьное выполнение на бинарном коде программ. Эта среда не решает непосредственно задачу поиска ошибок в программах, но предоставляет программисту инфраструктуру для реализации алгоритмов анализа поверх среды символьного выполнения.

Анализируемая программа выполняется на эмуляторе QEMU [36]. За символьное выполнение отвечает виртуальная машина KLEE [33]. Некоторые данные, используемые программой, помечаются как символьные. Инструкции, результат выполнения которых зависит от символьных данных, транслируются в представление LLVM и выполняются на виртуальной машине KLEE. Если символьные данные влияют на выполнение инструкции перехода, ветвь выполнения раздваивается и для каждой дочерней ветви дублируется состояние системы (состояние виртуального ЦПУ, состояние виртуальных устройств, состояние физической памяти). Для каждой из дочерних ветвей вводятся ограничения на диапазон возможных значений символьных данных в зависимости от того, по какой ветке пошло выполнение. Далее каждая ветка анализируется отдельно.

Существует два способа управления процессом символьного выполнения: можно либо встроить вызовы S^{2}E в код программы (эти вызовы транслируются в машинные инструкции), либо создать lua-скрипт, который будет запускать программу и нужные вызовы S^{2}E при помощи гостевых утилит. В тексте скрипта нужно указать модули S^{2}E, которые будут использоваться во время анализа. S^{2}E автоматически детектирует загрузку/выгрузку модулей для ОС Windows (поддерживаются версии XPSP2, XPSP3, XPSP2-CHK, XPSP3-CHK, SRV2008SP2), детектирует появление BSOD, позволяет указать точки загрузки модулей для ОС Linux. S^{2}E позволяет встраивать собственные обработчики событий (например, можно вызвать какую-нибудь функцию, когда счётчик инструкций виртуального процессора попадёт на заданный адрес). В QEMU добавлена возможность подключения модельного PCI-устройства в гостевую систему, что позволяет эмулировать получение произвольных данных от устройств с целью тестирования драйверов.

Скорость работы S^2E, несмотря на внесённые разработчиками изменения, сравнима со скоростью оригинальной версии QEMU. S^2E позволяет увеличить скорость перебора путей путём распараллеливания на несколько ядер.

1.3.3 Инструмент Avalanche

Avalanche [31] – это инструмент для динамического поиска дефектов в бинарном коде, разработанный в ИСП РАН. Avalanche использует возможности динамического профилирования программы, предоставляемые Valgrind, для сбора и анализа трассы выполнения программы. Результатом такого анализа становится либо набор входных данных, на которых в программе возникает ошибка, либо набор новых тестовых данных, позволяющий обойти ранее не выполнявшиеся и, соответственно, ещё не проверенные фрагменты программы. Таким образом, имея единственный набор тестовых данных, Avalanche реализует итеративный динамический анализ, при котором программа многократно выполняется на различных, автоматически сгенерированных тестовых данных, при этом каждый новый запуск увеличивает покрытие кода программы.

На начальном этапе разработки область применимости инструмента ограничена определенным классом программ – входные данные программа получает из одного файла. В настоящий момент реализовано получение входных данных посредством сетевого интерфейса (TCP, UDP).

Инструмент Avalanche состоит из четырёх основных компонентов: двух модулей расширения Valgrind – Tracegrind и Covgrind, инструмента проверки выполнимости ограничений STP и управляющего модуля. Tracegrind динамически отслеживает поток помеченных данных в анализируемой программе и накапливает условия для обхода ещё не пройденных частей и для срабатывания опасных операций. Эти условия при помощи управляющего модуля передаются STP для исследования их выполнимости. Если какое-то из условий выполнимо, то STP определяет те значения всех входящих в условия переменных (в том числе и значения байтов входного файла), которые обращают условие в истину.

В случае выполнимости условий для срабатывания опасных операций программа запускается управляющим модулем повторно (на этот раз без профилирования) с соответствующим входным файлом для подтверждения найденной ошибки.

Выполнимые условия для обхода ещё не пройденных частей программы определяют набор возможных входных файлов для новых запусков программы. Таким образом, после каждого запуска программы инструментом STP автоматически генерируется множество входных файлов для последующих запусков анализа. Далее встаёт задача выбора из этого множества наиболее «интересных» входных данных – в первую очередь должны обрабатываться входные данные, на которых наиболее вероятно возникновение ошибки. Для решения этой задачи используется эвристическая метрика – количество ранее не обойдённых базовых блоков в программе. Для измерения значения эвристики используется компонент Covgrind, в функции которого входит также фиксация возможных ошибок выполнения.

Существенным ограничением Avalanche является экспоненциальная сложность решаемых задач – проверки выполнимости булевских формул и обхода дерева условных переходов программы. Наличие подобного ограничения приводит к тому, что эффективно обнаруживаются лишь те ошибки, которые находятся близко к началу пути выполнения программы.

1.3.4 Анализ циклов

Отдельной проблемой при динамическом символьном выполнении является анализ циклов. Если условие выхода из цикла зависит от символьных данных, в точках выхода из цикла порождается большое количество новых путей, что значительно усложняет анализ. Решением этой проблемы может быть абстрагирование от числа итераций цикла и анализ цикла как единого целого.

В работе [37] предлагается метод поиска выхода за границы буфера, совмещающий символьное выполнение бинарного кода с моделированием переменного числа итераций в циклах. По результатам статического дизассемблирования выявляются циклы. Для каждого объемлющего цикла заводится свободная символьная переменная, описывающая число итераций соответствующего цикла

(*trip counter*). На основе анализа линейных отношений строятся формулы, показывающие вычисления над символьными данными с учётом счётчиков итераций. Анализ линейных отношений консервативно определяет, будет ли в теле цикла некоторая величина прирастать на одно и то же слагаемое на всех возможных путях выполнения. Счётчики связываются с форматом входных данных, каждое поле переменной длины соотносится со счётчиком итераций. После символьного выполнения солвер пытается решить систему ограничений. Новые входные данные генерируются с учётом вычисленных значений счётчиков, исходя из них и формата данных строится новый входной буфер. К сожалению, текст опубликованной статьи не содержит исчерпывающего описания разработанного метода, что позволяет судить только о некоторых принципиальных аспектах.

Инструмент Splat [38] позволяет автоматически анализировать исходный код на языке Си и генерировать входные данные, приводящие к нарушению доступа к памяти. В данном инструменте используется понятие абстрактной длины и символьная интерпретация некоторых функций. Основным ограничением инструмента является то, что он предназначен только для исходных текстов программ на языке Си, в отличие от предлагаемого метода, который анализирует бинарный код.

К сожалению, инструменты, описанные в работах [37; 38], недоступны.

1.4 Сравнение подходов

Подходы на основе статического анализа кода хорошо справляются с поиском ошибок выхода за границы буфера и являются наиболее распространённым способом выявления ошибок при промышленной разработке. К сожалению, инструменты статического анализа кода выдают ощутимую долю ложных срабатываний: около половины от общего числа, лидирующие инструменты на отдельных проверках показывают около 20% ложных срабатываний. Кроме того, качественная оценка критичности найденной ошибки требует изучения состояния исполняющейся программы в момент сбоя, для чего необходимы соответствующие входные данные. Статический анализ такие данные предоставить не способен.

Подходы на основе инструментации кода программы позволяют находить ошибки, которые гарантированно присутствуют в программе. Однако, подавляющее большинство инструментов, осуществляющих инструментацию бинарного кода, не универсальны: существуют сложности в применении их к анализу кода ядра, а также затруднён перенос этих инструментов на другие процессорные архитектуры. Общим недостатком подходов на основе инструментации оказывается то, что ошибки в программах обнаруживаются только на тех путях выполнения, которые были задействованы в процессе тестирования программы и только в случае проявления ошибки. Таким образом, для выявления ошибок критически важно качество набора тестов, они не только должны давать высокое покрытие кода, но и вырабатывать необходимое состояние программы в заданной точке для срабатывания ошибки. При регрессионном тестировании обеспечить нужное состояние возможно, но при поиске неизвестных, не всегда срабатывающих ошибок – случайным образом фактически невозможно, требуется анализировать свойства программного кода.

Подходы на основе символьного выполнения предоставляют более гибкие возможности для поиска ошибок, поскольку учитывают взаимосвязи между потоками данных в программе. К недостаткам можно отнести проблему экспоненциального взрыва числа состояний, присущую подходам, основанным на онлайн символьном выполнении. Среди рассмотренных инструментов полносистемный анализ поддерживается только в S^2E, однако эта среда символьного выполнения является только инфраструктурой для анализа, в которой не реализованы методы и алгоритмы для поиска ошибок выхода за границы буфера. Более того, реализация таких методов потребует предварительных действий по восстановлению статического представления по бинарному коду для выявления функций ввода и вывода, циклов и других важных программных конструкций.

Проанализировав достоинства и недостатки рассмотренных методов, можно выделить свойства, которые необходимо обеспечить при разработке более совершенного метода поиска ошибок:

– Отсутствие или минимизация количества ложноположительных срабатываний. Этого свойства можно достичь с помощью предоставления подтверждающего примера и его дальнейшей проверки.
– Универсальность: абстрагирование от ОС и процессорной архитектуры.

– Полносистемность: возможность анализировать не только отдельные приложения, но и код ядра ОС, а также взаимодействие нескольких процессов.
– Обнаружение ошибок, которые не реализовались во время запуска программы.

Первое и последнее требования трудносовместимы, поскольку обнаружение нереализованных ошибок предполагает анализ множества возможных путей выполнения с абстрактным состоянием программы, а это неизбежно приводит к ложным срабатываниям. Возможный компромисс представляется исследованием фиксированного набора трасс, что типично для прогона тестов, но с некоторыми обобщениями возможных состояний программы, аналогично тому, как это было предложено в работе [38].

Относительная независимость анализа от процессорной архитектуры достигается посредством применения промежуточного представления, такого как VEX системы Valgrind или REIL системы BinNavi. Анализу кода предшествует бинарная трансляция, сами алгоритмы анализа работают с промежуточным представлением, без привязки к системе команд конкретного процессора. Абстрагирование от ОС достигается применением модели, описывающей работу системных вызовов. В задаче поиска выхода за пределы буфера точность моделирования системных вызовов не так критична, достаточно иметь общий способ описания видимых для приложения потоков данных, а также учитывать семантику функций, отвечающих за ввод/вывод и управление динамической памятью. В современных ОС общего назначения (Windows, Linux, FreeBSD) число таких системных вызовов невелико. Достаточно часто встроенное ПО реализуется на основе ОС Linux, что открывает возможность анализировать программы не только для персональных компьютеров, но и для маршрутизаторов, смартфонов и различных устройств, составляющих интернет вещей.

Возможность полносистемного анализа даёт применение соответствующего программного эмулятора, на базе которого реализуется среда контролируемого выполнения. К сожалению, при использовании эмуляторов реализация сложных алгоритмов анализа значительно замедляет эмуляцию, что приводит к ощутимому влиянию на работу операционной системы. Замедление может привести к самым разным последствиям: от обрыва сетевых соединений по таймауту до полной неработоспособности пользовательских приложений. Кроме того, анализ на уровне эмулятора усложняет получение статической информации об исследуемой

программе, поскольку её накопление в реальном времени требует значительных объёмов памяти. Для решения этих проблем используют post-mortem анализ, который позволяет проводить анализ после выполнения программы. Такой анализ становится возможным за счёт применения методов детерминированного воспроизведения либо трассировки.

Глава 2. Поиск ошибок по трассам выполнения

2.1 Метод символьной интерпретации длин буферов

В основе метода поиска ошибок по трассам выполнения лежит метод символьной интерпретации трасс, детально описанный в работе [39]. В рамках этого метода по трассе выполнения производится построение уравнений, описывающих прохождение программы по некоторому пути, отражённому в трассе. Такой набор уравнений называется предикатом пути. К этому набору уравнений добавляются уравнения, описывающие проявление некоторой ошибки в программе, называемые предикатом безопасности. Если система уравнений, состоящая из предиката пути и предиката безопасности оказалась совместной, решение этой системы является доказательством существования ошибки в исследуемой программе.

Предметом анализа являются трассы машинных инструкций, полученные в результате работы полносистемного эмулятора [40; 41]. Трассы содержат состояния регистров, информацию о прерываниях и взаимодействии с периферийными устройствами, что позволяет восстанавливать статико-динамическое представление для всех работавших в системе программ и эффективно исследовать его свойства.

Современные процессорные архитектуры содержат множество различных инструкций со сложной семантикой и побочными эффектами. Распространенным приёмом, обеспечивающим поддержку различных архитектур, является использование промежуточного представления. В рамках разрабатываемого метода используется промежуточное представление Pivot [42; 43], позволяющее унифицировано описывать операционную семантику инструкций различных архитектур. Данное промежуточное представление отвечает требованиям SSA-формы, что позволяет значительно упростить анализ. Основные операторы Pivot, используемые при построении предиката пути, следующие.

- Оператор NOP – не имеет никакого эффекта.
- Оператор INIT – инициализирует локальную переменную константным значением.

– Оператор APPLY – применяет одну из модельных операций. В качестве параметров и результата используются локальные переменные.
– Оператор BRANCH – осуществляет передачу управления.
– Оператор LOAD – производит загрузку значения из одного из адресных пространств.
– Оператор STORE – записывает значение в одно из адресных пространств.

Для описания памяти и регистров в Pivot используется модель единообразных адресных пространств. С точки зрения этой модели все адресуемые операнды целевой архитектуры, с которыми может работать машина (регистры, память, порты ввода-вывода), размещаются в линейных адресных пространствах, обращение к ним использует пару (идентификатор пространства, смещение). Для учёта побочных эффектов работы операций используется модельное слово состояния, которое аналогично регистру флагов в x86.

Для поиска ошибок необходимо получить трассу, содержащую один или несколько запусков исследуемой программы на некотором наборе входных данных. В процессе анализа такой трассы достаточно рассматривать только те инструкции, которые относятся к обработке входных данных с момента их получения до момента аварийного завершения программы. Для отбора таких инструкций используется алгоритм динамического слайсинга трассы [44], дополненный анализом помеченных данных. В качестве начальных буферов для алгоритма динамического слайсинга трассы используются буферы в памяти в точках получения входных данных.

Описываемый в данной работе подход основан на символьной интерпретации длин буферов, обрабатываемых в программе. Каждому буферу памяти ставится в соответствие символьная переменная, описывающая длину буфера, значение которой распространяется вместе с буфером в процессе выполнения программы. Несмотря на то, что в рамках одного запуска программы длины обрабатываемых буферов фиксированы, символьный анализ длин буферов позволяет абстрагироваться от конкретных значений и рассматривать общий случай. Такой подход позволяет описывать ошибочную ситуацию в виде выражения над длинами буферов и находить такие значения длин, при которых эта ошибочная ситуация проявится. Если при этом длины обрабатываемых буферов зависят от входных данных программы, возможно построение такого набора входных данных, при обработке которого произойдёт выход за границы буфера.

Отдельной проблемой при решении поставленной задачи является описание ошибочных ситуаций. Для описания условий выхода за границы буфера требуется определение границ буферов, которые являются живыми в каждой точке анализа. Буфер считается живым на некотором шаге трассы, если этот шаг попадает в отрезок между первой записью в буфер и его последним использованием. Информация о границах буферов в программе доступна в исходном коде программы, но теряется в процессе компиляции. Поскольку по бинарному коду восстановить точные границы исходных буферов в общем случае невозможно, используется консервативный подход, позволяющий получить границы объемлющих буферов. Если будет обнаружена ошибка выхода за границы объемлющего буфера, это будет автоматически означать существование ошибки выхода за границы другого буфера, который обрабатывался в программе.

Чаще всего, ошибка выхода за границы буфера проявляется во время выполнения машинных команд, которые обращаются к памяти с использованием косвенной адресации. При косвенном обращении к памяти адрес ячейки памяти, в который происходит загрузка или выгрузка данных, может зависеть от входных данных, что потенциально позволяет обращаться за границы допустимой области памяти. Эти машинные команды могут входить в состав библиотечных функций работы с памятью. Современные версии библиотечных функций используют расширения процессора, такие как SSE2, для ускорения работы. Анализ машинных команд SSE2 значительно усложняет задачу поиска ошибок. В то же время семантика многих библиотечных функций работы с памятью известна, что позволяет обрабатывать их как единое целое вместо обработки отдельных инструкций, входящих в эти функции.

Работу метода поиска ошибок выхода за границы буфера можно представить в виде последовательности следующих шагов (рисунок 2.1). Сначала аналитиком создаются наборы входных данных для одного или нескольких запусков исследуемой программы. Эти наборы входных данных могут быть расширены с помощью метода, подробно изложенного в разделе 2.4. Далее, для полученных наборов входных данных выполняется получение трассы выполнения, содержащей соответствующие запуски исследуемой программы. Для полученной трассы выполняется повышение уровня представления: восстанавливаются границы циклов, вызовы функций и их параметры, а также другая информация, необходимая для последующего анализа. Кроме того, с помощью метода, изложенного в разделе 2.3, восстанавливаются циклы, соответствующие функциями работы со

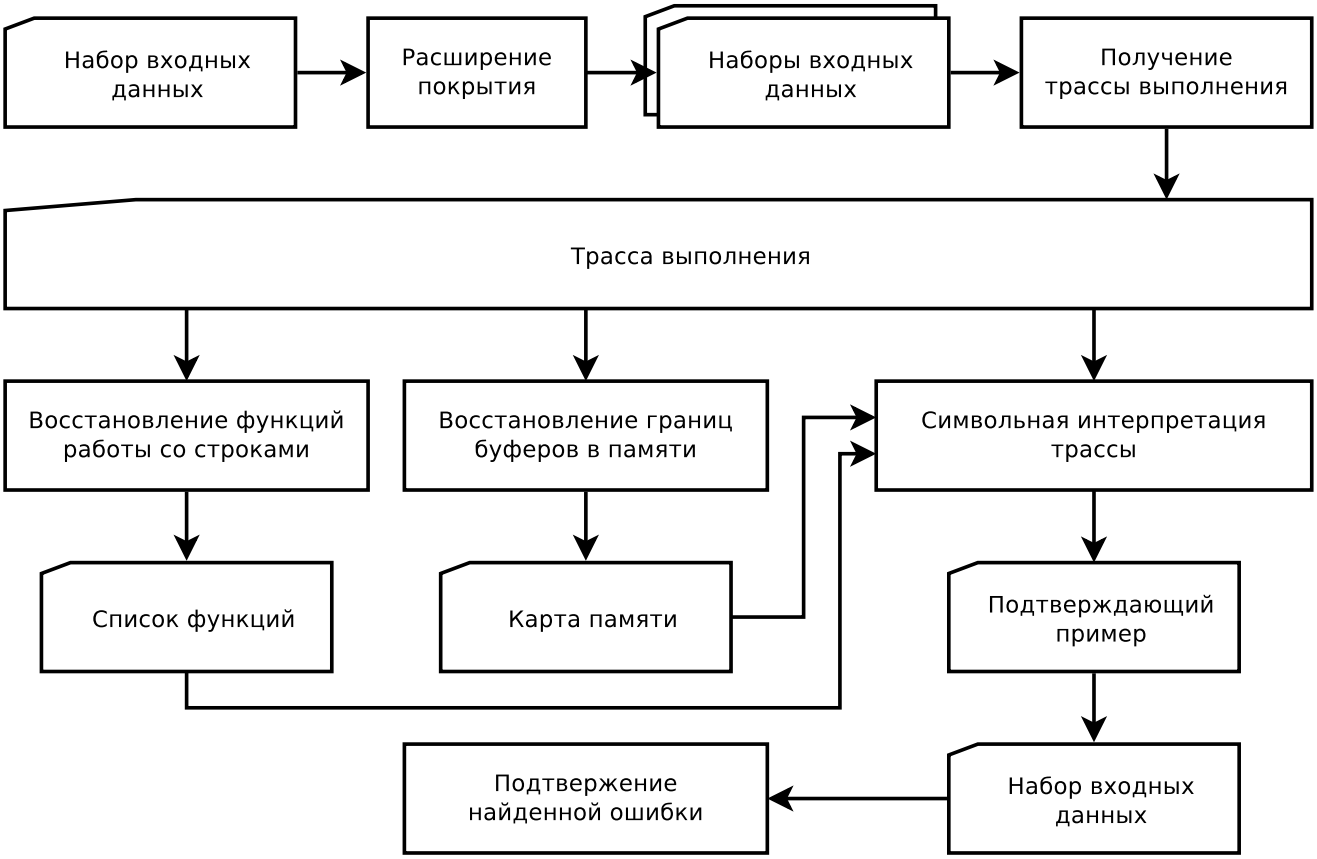

Рисунок 2.1 — Блок-схема метода.

строками. Далее, с помощью метода, описанного в разделе 2.2 восстанавливаются границы буферов в памяти. Когда вся необходимая информация восстановлена, определяются вызовы функций получения входных данных, получаемые ими данные помечаются символьными и запускается символьная интерпретация трассы. Если в процессе интерпретации обнаруживается ошибка выхода за границы буфера, выполняется построение подтверждающего примера с помощью SMT-решателя [45], после чего интерпретация завершается. На основе подтверждающего примера выполняется построение набора входных данных и подтверждение найденной ошибки с помощью запуска исследуемой программы на этом наборе входных данных.

2.1.1 Трансляция машинных инструкций

Отобранные машинные команды транслируются в промежуточное представление Pivot согласно правилам трансляции. Правила трансляции для нетривиальных инструкций могут содержать ветвления, реализованные с помощью соответствующих инструкций условной передачи управления. Над промежуточным представлением производится дополнительная интерпретация, в результате которой формируется трасса Pivot-инструкций, отражающая путь выполнения в промежуточном представлении машинной команды при конкретных значениях параметров.

В приложении А в листинге А.1 приводится описание вариантов инструкции `ADC` для различных комбинаций операндов. Высокоуровневый язык описания операционной семантики предоставляет возможность описания семантики машинной команды сразу для нескольких типов операндов. В данном примере по высокоуровневому описанию будет сгенерировано десять низкоуровневых описаний инструкции `ADC`: четыре описания для первого случая, три описания для второго и по одному для оставшихся. При трансляции машинной команды используется наиболее подходящее описание исходя из типов операндов машинной команды. К примеру, при трансляции инструкции `ADC EDX, ESI` используется описание, соответствующее четвёртому случаю. В листинге А.2 приводится пример трансляции инструкции `ADC EDX, ESI` в код промежуточного представления Pivot, а в листинге А.3 – трасса выполнения этой инструкции на конкретных входных данных (в данном случае, при значении флага CF=0).

Следует отметить, что при трансляции машинной инструкции в промежуточное представление Pivot транслируется не только сама инструкция, но и её параметры. Например, при трансляции инструкции `INC DWORD PTR [EAX + 000005C4h]` в результирующий код добавляются инструкции, вычисляющие выражение `EAX + 000005C4h`.

В ходе трансляции трасса машинных инструкций преобразуется в трассу Pivot-инструкций, которая и анализируется в дальнейшем. Такой подход позволяет алгоритмам анализа абстрагироваться от архитектуры процессора, которая использовалась для выполнения исследуемой программы.

2.1.2 Контекст интерпретации

Полученная трасса Pivot-инструкций представляется в виде упорядоченной последовательности пар

$$(instr_0, M_0)$$
$$(instr_1, M_1)$$
$$(instr_2, M_2)$$
$$\dots$$
$$(instr_{N-1}, M_{N-1})$$

где $instr_i, 0 \leqslant i < N$ – выполнявшаяся на шаге i инструкция, а M_i – состояние памяти компьютера перед выполнением этой инструкции. Память представляет набор адресуемых машинных слов $M = (m_0 \dots m_{TOM})$, в котором единообразно объединены все ячейки, обладающие состоянием: как оперативная память компьютера, так и регистры. Здесь инструкция – тройка, описывающая операцию над данными и ее фактические операнды: $instr = \langle OC, \{use_i\}, \{def_i\}\rangle$, где OC – код операции, $use, def \in M$ – множества ячеек, считываемых и записываемых соответствующей машинной командой или набором команд. В наборах операндов явно указываются ячейки памяти, которые считываются и записываются при выполнении инструкции. Непосредственно адресуемые операнды не указываются. В случае косвенной адресации явно указываются обе ячейки: фактический операнд и ячейка, задающая адрес.

В работах, описывающих динамический анализ помеченных данных, перечень кодов операций реализует минимальный набор RISC команд [46], расширив его инструкцией ввода пользовательских данных. В данном случае базовый набор инструкций выглядит следующим образом:

- выполнение унарной операции $\langle \Diamond_u, \{use\_cell\}, \{def\_cell\}\rangle$,
- выполнение бинарной операции $\langle \Diamond_b, \{use\_cell_1, use\_cell_2\}, \{def\_cell\}\rangle$,
- загрузка данных из памяти $\langle load, \{src\_cell, src\_addr\_cell\}, \{dest\_cell\}\rangle$,
- выгрузка данных в память $\langle store, \{src\_cell, dest\_addr\_cell\}, \{dest\_cell\}\rangle$,
- безусловная передача управления $\langle jmp, \{[addr\_cell]\}, \{\}\rangle$,
- условная передача управления $\langle jct, \{cond\_cell, addr\_cell\}, \{\}\rangle$, $\langle jcf, \{cond\_cell\}, \{\}\rangle$.

Легко видеть, что каждой инструкции промежуточного представления Pivot соответствует одна из инструкций из списка выше. Для удобства интерпретации явно разделены сработавшая (jct) и несработавшая (jcf) инструкция условной передачи управления, поскольку при их обработке к предикату пути добавляются различные ограничения.

Инструкции, которые входят в состав функций с известной семантикой, не обрабатываются, а вместо этого применяется правило для соответствующей функции. Каждой такой функции соответствует специальная инструкция. Для функций с известной семантикой поддерживается следующий набор инструкций:

– выделение памяти $\langle malloc, \{size\}, \{addr\}\rangle$,
– освобождение памяти $\langle free, \{addr\}, \{\}\rangle$,
– определение длины строки $\langle strlen, \{addr\}, \{len\}\rangle$,
– копирование строк $\langle strcpy, \{dest\_addr, src\_addr\}, \{\}\rangle$,
– конкатенация строк $\langle strcat, \{dest\_addr, src\_addr\}, \{\}\rangle$,
– копирование фиксированной длины $\langle memcpy, \{dest\_addr, src\_addr, n\}, \{\}\rangle$,
– ввод пользовательских данных $\langle read, \{addr, size\}, \{size\_read\}\rangle$.

Стоит отметить, что приведённый набор инструкций не является окончательным и при необходимости может быть расширен.

В процессе интерпретации поддерживается множество символьных переменных $\mathfrak{S}$, над которыми допустимы унарные и бинарные операции, а также операции загрузки и выгрузки данных.

Для описания работы с памятью используется понятие буфера, имеющего символьную длину [38], далее в тексте обозначаемый как L-буфер. Такой буфер задаётся тройкой $l = [\![base, clen, slen]\!]$, содержащей адрес базы, реальную и символьную длину, отвечающую за абстрактный размер буфера. Символьный буфер схематично изображён на рисунке 2.2.

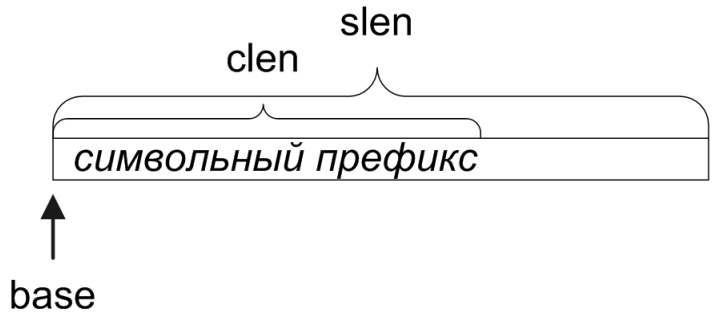

Рисунок 2.2 — Структура символьного буфера.

Интерпретация шагов трассы инструкций происходит в контексте, который отражает текущее символьное состояние, предикат пути, множество символьных переменных и два множества L-буферов памяти: A и I.

Отображение Δ связывает ячейки (машинные слова) памяти и выражения над символьными переменными. Получение символьного и конкретного численного значения ячейки памяти будем описывать $\Delta[m]$ и $M[m]$ соответственно.

В ходе интерпретации трассы на основе предиката пути и дополнительных ограничений проверяются условия выхода за границы буферов. Для обозначения проверок вводится функция *Assert*(*условие*), которая проверяет истинность заданного условия.

Также существует отдельный вид L-буферов, которые соответствуют областям выделенной памяти. У таких буферов абстрактная длина может быть константным выражением. В отличие от работы [38], в предлагаемом подходе каждый L-буфер помещён в одно из множеств: A или I, в зависимости от того, соответствует буфер выделению памяти или вводу данных. Входные данные могут поступать из различных источников: сеть, файлы, аргументы командной строки. Поддержка двух множеств A и I позволит в дальнейшем выполнять дополнительные проверки.

Для работы с L-буферами вводится отображение, позволяющее получить для заданного адреса буфер в заданном множестве $T(X, addr) = x, x \in X$, адрес $addr$ указывает на одну из ячеек буфера x, а X – одно из множеств (A или I). Для краткости в дальнейшем отображения, связанные с конкретным множеством буферов, будут обозначаться как $T(A)$ и $T(I)$.

Для отображений T и Δ вводится операция обновления $\leftarrow$. Например, задание связи между ячейкой памяти m и символьным выражением s будет обозначаться как $\Delta[m \leftarrow s]$. Правила интерпретации команд представлены в последующих разделах в виде продукций. Для каждой команды приводится ее запись, под которой размещена продукция, описывающая преобразование контекста. В верхней части продукции приведены выполняющиеся действия, в нижней – состояния контекста до и после интерпретации команды. Исходя из рассмотренного выше, контекст представляется кортежем, состоящим из пяти компонент: $\Delta, Pp, A, I, \mathfrak{S}$. Для краткости, в нижней части продукции показываются только изменившиеся элементы контекста.

2.1.3 Правила интерпретации для функций

Правила можно разделить на три группы:

1. интерпретация функций копирования и вычисления длины строк;
2. интерпретация функций менеджера кучи;
3. интерпретация функций получения входных данных.

При интерпретации функций из первой группы выполняется распространение символьных значений и длин буферов согласно правилам интерпретации. Так, например, при копировании строки в другую область памяти, копируется и символьное значение её длины. Аналогичным образом значение, возвращаемое функцией вычисления длины строки, в результате интерпретации будет иметь символьное значение длины строки, а не фактически вычисленное значение. Кроме того, при интерпретации функций копирования данных выполняются дополнительные проверки на выход за границы буфера. К системе уравнений, описывающей предикат пути, добавляются уравнения, описывающие выход за границы буфера при чтении или при записи. Если полученная система уравнений оказалось совместной, фиксируется факт обнаружения ошибки с предоставлением набора входных данных, который можно использовать для подтверждения ошибки.

При интерпретации функций менеджера кучи производится обновление множества выделенных L-буферов A. При этом добавляемые L-буферы имеют символьную длину, которая вычисляется на основе значения размера, передаваемого в функцию выделения памяти. Такие правила интерпретации позволяют корректно обрабатывать случаи, когда размер выделенного буфера соответствует размеру копируемых в него данных, если этот размер является символьным значением.

Интерпретация функций получения входных данных заключается в добавлении L-буферов в множество I. В рамках предлагаемого метода все входные данные считаются символьными. Кроме того, новые свободные символьные переменные ставятся в соответствие не только данным буфера, но и его размеру. Если в процессе анализа будет обнаружена ошибка выхода за границы буфера, решением системы уравнений будут значения для буфера с входными данными (символьный префикс) и его длины.

Следует отметить, что значений для входных данных, получаемых в результате решения системы уравнений, не всегда достаточно для подтверждения найденной ошибки. Если требуемый для срабатывания ошибки размер входных данных больше, чем размер данных, поданных изначально на вход программе, эти данные должны быть расширены до соответствующего размера. Эта задача в общем случае неразрешима, так как требует знания формата входных данных, однако на практике получение подтверждающего набора входных данных в большинстве случаев является простой задачей для аналитика. К примеру, в случае, когда программа обрабатывает параметр командной строки целиком как строку и не анализирует её содержимое, процесс построения строки заданной длины тривиален: достаточно дополнить исходную строку нетерминальными символами (например, символами 'A') до требуемой длины.

При интерпретации инструкции `malloc` во множестве A происходит создание L-буфера. В зависимости от операнда $size$, абстрактная длина L-буфера может иметь как символьное, так и константное (конкретное) значение.

$$\langle malloc, \{size\}, \{addr\}\rangle$$

$$\frac{l = [[M(addr), M(size), \Delta[size]]]}{A \vdash A \cup l} \tag{2.1}$$

При интерпретации инструкции `free` происходит удаление L-буфера из множества A, конкретное значение операнда $addr$ должно соответствовать полю $base$ у L-буфера l.

$$\langle free, \{addr\}, \{\}\rangle$$

$$\frac{l = T(A, M(addr)), Assert(M(addr) = l.base)}{A \vdash A \setminus l} \tag{2.2}$$

При интерпретации инструкции `strlen` происходит присваивание операнду len значения абстрактной длины L-буфера, который найден, исходя из значения операнда $addr$.

$$\langle strlen, \{addr\}, \{len\}\rangle$$

$$\frac{l = T(I, M(addr))}{\Delta \vdash \Delta[len \leftarrow l.slen - (M(addr) - l.base)]} \tag{2.3}$$

При интерпретации инструкций `strcpy` и `strcat` происходит проверка на переполнение буфера при копировании (конкатенации), также создаётся новый буфер из множества I, затем обновляется отображение ячеек памяти на символьные переменные.

$$\langle strcpy, \{dst\_addr, src\_addr\}, \{\}\rangle$$

$$\frac{\begin{gathered} i = T(I, M(src\_addr)), i' = \left[\begin{bmatrix} M(dst\_addr), i.clen - (M(src\_addr) - i.base), \\ i.slen - (M(src\_addr) - i.base) \end{bmatrix}\right], \\ a = T(A, M(dst\_addr)), Assert(i' \sqsubset a) \end{gathered}}{\Delta, I \vdash \Delta[m_{i'.base}, \ldots, m_{i'.base+i'.clen-1} \leftarrow m_{M(src\_addr)}, \ldots, m_{M(src\_addr)+i'.clen-1}], I \cup i'} \tag{2.4}$$

$$\langle strcat, \{dst\_addr, src\_addr\}, \{\}\rangle$$

$$\frac{\begin{gathered} i_{src} = T(I, M(src\_addr)), i_{dst} = T(I, M(dst\_addr)), \\ \mathit{offset}_{dst} = (M(dst\_addr) - i_{dst.base}), \mathit{offset}_{src} = (M(src\_addr) - i_{src.base}) \\ i' = \left[\begin{bmatrix} M(dst\_addr), i_{src.clen} - \mathit{offset}_{src} + i_{dst.clen}, \\ i_{src.slen} - \mathit{offset}_{src} + i_{dst.slen} \end{bmatrix}\right], \\ a = T(A, M(dst\_addr)), Assert(i' \sqsubset a) \end{gathered}}{\Delta, I \vdash \Delta\begin{bmatrix} m_{i_{dst.base}+i_{dst.clen}}, \ldots, m_{i_{dst.base}+i'_{clen}-1} \leftarrow \\ m_{M(src\_addr)}, \ldots, m_{M(src\_addr)+i_{src.clen}-\mathit{offset}_{src}-1} \end{bmatrix}, I \cup i' \setminus i_{dst}} \tag{2.5}$$

При интерпретации инструкции `memcpy` происходит проверка на выходы за границы при чтении буфера источника и на переполнение буфера назначения

при копировании, далее создаётся новый буфер из множества I, затем обновляется отображение ячеек памяти на символьные переменные.

$$\langle memcpy, \{dest\_addr, src\_addr, n\}, \{\}\rangle$$

$$\frac{\begin{gathered} i_{src} = T(I,M(src\_addr)), i_{dst} = T(I,M(dst\_addr)), \\ i' = [[M(dst\_addr),M(n),\Delta(n)]], \\ Assert(i_{src}.slen - (src\_addr - i_{src}.base) < \Delta(n)), \\ a_{src} = T(A,M(src\_addr)), Assert(i_{src} \sqsubset a_{src}), \\ a = T(A,M(dst\_addr)), Assert(i' \sqsubset a) \end{gathered}}{\Delta, I \vdash \Delta[m_{i'.base}, \ldots, m_{i'.base+i'.clen-1} \leftarrow m_{M(src\_addr)}, \ldots, m_{M(src\_addr)+i'.clen-1}], I \cup i'} \tag{2.6}$$

При интерпретации инструкции `read` происходит проверка на выход за границы буфера при чтении данных из внешнего источника, учитывая максимальный размер считанных данных, также создаётся новый буфер из множества I, затем обновляется отображение ячеек памяти на символьные переменные.

$$\langle read, \{addr, size\}, \{size\_read\}\rangle$$

$$\frac{\begin{gathered} a_{src} = T(A,M(addr)), s_0, \ldots, s_{M(size\_read)} \subset \mathfrak{S}, \\ i_{src} = [[M(addr),M(size\_read),s_{len}]], s_{len} \subset \mathfrak{S}, \\ Assert(i_{src} \sqsubset a_{src}), Assert(i_{src}.slen \leqslant M(size)) \end{gathered}}{\Delta, I \vdash \Delta[m_{i_{src}.base}, \ldots, m_{i_{src}.clen-1} \leftarrow s_0, \ldots, s_{M(size\_read)}], I \cup i_{src}} \tag{2.7}$$

2.1.4 Правила интерпретации для инструкций

Инструкции, выполняющие унарные и бинарные операции над данными транслируются в соответствующие им операции над символьными значениями.

Операции загрузки и выгрузки обновляют отображение Δ в контексте интерпретации, позволяя загружать и сохранять символьные значения. Если загружаемой ячейке памяти не соответствует символьное значение, результатом загрузки будет конкретное значение этой ячейки. Инструкции условной передачи управления добавляют ограничения в предикат пути Pp, выраженные через условия выполнения перехода, передаваемые в качестве параметра $cond\_cell$.

Интерпретация инструкции безусловной передачи управления не предусматривает какого-либо изменения символьного контекста. Однако, выполнение дополнительных проверок при интерпретации таких инструкций позволило бы находить ошибки, связанные с отсутствием ограничений на адрес перехода, поиск которых выходит за рамки данной работы.

При интерпретации инструкции, соответствующей унарной (бинарной) операции, происходит обновление отображения ячеек памяти на символьные переменные с учётом выполнения операции.

$$\langle \Diamond_u, \{use\_cell\}, \{def\_cell\} \rangle$$

$$\frac{}{\Delta \vdash \Delta[def\_cell \leftarrow \Diamond_u \Delta(use\_cell)]} \tag{2.8}$$

$$\langle \Diamond_b, \{use\_cell_1, use\_cell_2\}, \{def\_cell\} \rangle$$

$$\frac{}{\Delta \vdash \Delta[def\_cell \leftarrow \Delta(use\_cell_1) \Diamond_b \Delta(use\_cell_1)]} \tag{2.9}$$

При интерпретации инструкции load происходит проверка на выход за границы L-буфера при чтении данных из него. Дополнительно производится обновление состояния в соответствии с семантикой операции загрузки данных из памяти.

$$\langle load, \{src\_cell, src\_addr\_cell\}, \{dst\_cell\}\rangle$$

$$\frac{\begin{array}{c} a_{src} = T(A, M(src\_addr\_cell)), \\ Assert(\Delta(src\_addr\_cell) < a_{src}.base + a_{src}.slen) \end{array}}{\Delta \vdash \Delta[dst\_cell \leftarrow src\_cell]} \tag{2.10}$$

При интерпретации инструкции store происходит проверка на выход за границы L-буфера при записи данных в него. Дополнительно производится обновление состояния в соответствии с семантикой операции выгрузки данных в память.

$$\langle store, \{src\_cell, dst\_addr\_cell\}, \{dst\_cell\}\rangle$$

$$\frac{\begin{array}{c} a_{src} = T(A, M(dst\_addr\_cell)), \\ Assert(\Delta(dst\_addr\_cell) < a_{dst}.base + a_{dst}.slen) \end{array}}{\Delta \vdash \Delta[dst\_cell \leftarrow src\_cell]} \tag{2.11}$$

При интерпретации инструкций jct и jcf в предикат пути добавляются уравнения, описывающие выполнение (не выполнение) операции условного перехода.

$$\langle jct, \{cond\_cell, addr\_cell\}, \{\}\rangle$$

$$\frac{}{Pp \vdash Pp \cup (\Delta(cond\_cell) = true)} \tag{2.12}$$

$$\langle jcf, \{cond\_cell\}, \{\}\rangle$$

$$\frac{}{Pp \vdash Pp \cup (\Delta(cond\_cell) = false)} \tag{2.13}$$

2.1.5 Правила обработки прерываний

Поскольку символьная интерпретация выполняется в рамках полносистемного анализа, изменение состояния машины обуславливается не только выполнением команд, явно зафиксированных в трассе, но и возникающими прерываниями. В момент возникновения прерывания меняются значения регистров процессора и различные регистры сохраняются на стеке. В общем случае анализируется весь поток инструкций, включая код ОС и всех процессов, работающих на компьютере. Отслеживаются потоки данных, проходящие между различными процессами, передача данных между пользовательским и привилегированным кодом. Обработка прерывания процессором интерпретируется как совокупность операций копирования (сохранение состояния в памяти) и присвоения значений (выставление и сброс флагов, запись констант в регистры). Этот аспект работы компьютера формально представляется последовательностью инструкций промежуточного представления, размещённой в соответствующем месте трассы.

Такого рода сквозной анализ исполняемого кода необходим при исследовании работы вредоносного ПО, в частности вирусов. При поиске ошибок в разрабатываемом прикладном ПО целесообразно ограничивать область интерпретации, что требует изменения подхода.

Прерывания разделяются на три категории: аппаратные (асинхронные), исключения и ловушки. В случае аппаратных прерываний естественно предположить, что ядерный обработчик не будет менять пользовательские данные программы. Однако, программные прерывания могут инициироваться самим кодом исследуемой программы и могут приводить к намеренному изменению пользовательской памяти. Например, выполнение системного вызова `read()` приводит к частичной или полной перезаписи указанного пользователем буфера в памяти процесса. Игнорирование таких ситуаций нарушит корректность интерпретации.

Для преодоления указанной проблемы предлагается следующий подход. Поскольку трассируемый код выполняется на единственном вычислительном ядре, трасса выполнения пользовательской программы рассматривается как набор диапазонов трассы, границы которых образованы обработчиками прерываний. Работа многопоточного приложения не выходит за рамки такой модели, поскольку переключение потоков, как и процессов, осуществляется ядром ОС. Эффект

выполнения кода внутри ядра ОС атомарно применяется в момент возврата в соответствующее приложение, исходя из аннотации системного вызова. Пусть W – множество отслеживаемых (символьных) ячеек на определённом шаге трассы, W_i, W_o – множества отслеживаемых ячеек непосредственно перед входом в код ядра и после выхода из него. Тогда можно определить множества $A = W_o - W_i$ и $R = W_i - W_o$, соответствующие множествам добавленных и удалённых ячеек. После выхода из прерывания обновляется множество отслеживаемых ячеек W и карта Δ:

$$\begin{aligned} W &\vdash W_i - A \\ \Delta &\vdash \Delta[e \leftarrow M(e), \forall e \in R] \end{aligned} \tag{2.14}$$

В результате обновляется символьная карта Δ и множество отслеживаемых ячеек W. Кроме того, из этого множества удаляются все ячейки, которые были созданы в процессе обработки прерывания. Это удаление необходимо выполнить по той причине, что без полного анализа инструкций невозможно вычислить символьные значения, соответствующие этим ячейкам. Следует отметить, что на практике подобное ослабление не приведёт к заметному ухудшению точности анализа, поскольку в большинстве случаев из карты Δ удаляются ячейки, которые были перезаписаны данными, не зависящими от символьных переменных.

Применимость данного подхода не распространяется на определённый класс высокопроизводительных программ, использующих средства передачи данных без копирования из пространства пользовательского кода (в англоязычной литературе используется термин user-space zero copy). Примером такого средства является библиотека PF_RING [47], позволяющая передавать и принимать сетевые данные без какого-либо участия ядра ОС при наличии Ethernet-адаптера с поддержкой технологии DNA (Direct NIC Access). Данный класс программ не будет обладать изолированной от внешнего мира памятью.

2.1.6 Корректность представления операционной семантики целевых машинных команд

Действия, производимые программой, изначально представлены в виде машинного кода некоторой целевой процессорной архитектуры. Автоматическое

заключение о наличии или отсутствии некоторого свойства, в данном случае – выхода за границы буфера – требует представления программы в таком виде, которое пригодно для формального анализа. В отличие от традиционных компиляторных задач, при изучении бинарного кода распространён подход, когда анализ свойств программы сводится к задаче выполнимости формул в теориях. Построенная система формул передаётся SMT-решателю, который автоматически выдаёт заключение о ее совместности.

В существующих SMT-решателях имеется поддержка множества различных теорий (логик). Библиотека SMT-LIB [45] поддерживает несколько десятков логик (Рис. 2.3). Поскольку представляются действия, выполняемые машиной, значительная часть работ использует безкванторные формулы над теорией битовых векторов и массивов битовых векторов (`QF_ABV`).

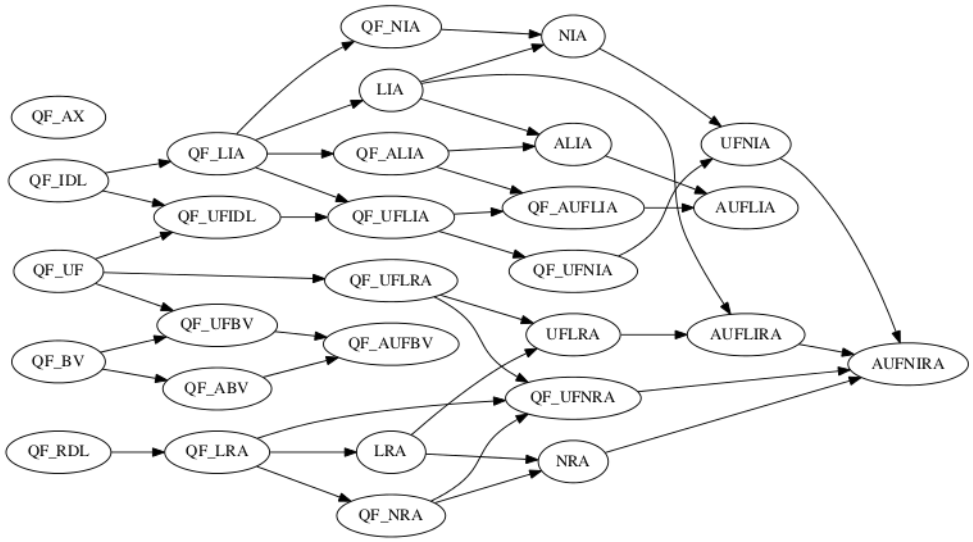

Рисунок 2.3 — Поддерживаемые в SMT-LIB логики и связи между ними. Стрелки показывают включение логик.

При переходе от полносистемного анализа к анализу одного процесса теряется информация об эффектах выполнения кода ядра ОС и других процессов, взаимодействующих с анализируемым. Поэтому анализ следует проводить в следующих предположениях:

– исследуемая программа не использует механизмы межпроцессного взаимодействия и регионы общей памяти с другими процессами;
– эффект выполнения кода ядра ОС известен и описан явно, либо же не влияет на обработку данных в программе.

В этих предположениях изолированность среды выполнения процесса позволяет утверждать о корректности кусочной интерпретации в рамках действия

адресного пространства заданного процесса (выбрасывание кода других процессов, интерпретация кода ядра в виде резюме – влияние на виртуальную память одного процесса). Можно утверждать, что при переходе от полносистемного анализа к анализу одного процесса анализ будет оставаться корректным и полным.

Следует отметить, что описание эффектов функций работы с файлами должно учитывать операции перемещения указателя файла. Это необходимо для корректной обработки ситуации, когда программа повторно считывает некоторый участок файла.

Расширение области возможных состояний, увеличение длин буферов (только массивы) влияет только на значения переменных (меняется карта памяти), содержимое массива в символьном суффиксе не должно влиять на поток управления, за исключением циклов.

2.2 Восстановление границ буферов по трассе выполнения

Области памяти, используемые в программе, можно разделить на три класса: статическая, динамическая и автоматическая память. Наибольший интерес представляет поиск ошибок выхода за границы буфера в динамической и автоматической памяти, так как эти ошибки наиболее опасны с точки зрения их эксплуатируемости и должны быть исправлены как можно скорее. В данной работе поиск ошибок выхода за границы буферов в статической памяти не рассматривается, но предложенные методы могут быть легко расширены для поддержки этого класса памяти.

Использование автоматической и динамической памяти предполагает размещение буферов в рамках некоторой ограниченной области памяти. Для автоматической памяти такой областью будет стековый фрейм функции, для динамической памяти – единица выделения памяти на куче. Несмотря на то, что выделяемые области памяти зачастую сильно больше размещённых в них буферов, именно эти области памяти используются для определения границ живых буферов. Такой выбор рассматриваемых областей памяти обусловлен возможностью поиска потенциально эксплуатируемых ошибок. Переполнение буфера с выходом за границы стекового фрейма может привести к перезаписи адреса возврата из функции и перехвату потока управления, а переполнение на куче с выходом за границу выделенного блока памяти – к повреждению служебных данных другого выделенного блока с дальнейшей модификацией критических областей памяти.

Границы буферов в динамической памяти определяются с помощью анализа вызовов функций выделения и освобождения памяти. В стандартной библиотеке `libc` ОС Linux для этого используются функции `malloc`, `calloc`, `realloc`, `free`. ОС Windows предоставляет около десятка аналогичных функций работы с памятью. Несмотря на такое обилие различных функций работы с памятью, каждую из функций можно отнести к одному из трёх классов:

1. выделение памяти (`malloc`, `calloc`, `LocalAlloc`, `GlobalAlloc`);
2. перераспределение памяти (`realloc`, `LocalReAlloc`, `GlobalReAlloc`);
3. освобождение памяти (`free`, `LocalFree`, `GlobalFree`).

Для анализа живых буферов в динамической памяти вводится понятие *слоя менеджера кучи*. Слой определяется набором функций выделения, перераспределения и освобождения памяти. Такой подход позволяет анализировать несколько различных менеджеров кучи, определяя для каждого из них отдельный слой. Эти менеджеры кучи также могут быть вложенными друг в друга: например, менеджер кучи, предоставляющий функцию `malloc` может использовать функции выделения страниц виртуальной памяти, относящиеся к другому менеджеру. Также этот подход позволяет анализировать многие нестандартные менеджеры кучи, такие как `jemalloc` и менеджер, предоставляемый библиотекой `glib`. Предложенный метод поиска ошибок выхода за границы буфера анализирует только один слой менеджера кучи, однако при необходимости можно работать с несколькими слоями, выполняя последовательно анализ для каждого из них.

2.3 Восстановление функций работы со строками

Для увеличения точности метода поиска ошибок следует анализировать не только те функции работы со строками, которые могут быть обнаружены с помощью анализа экспортируемых библиотечных функций, но и те функции, которые подверглись встраиванию в процессе компиляции и представляются в виде циклов работы со строками. Метод позволяет находить в трассе циклы, работа которых эквивалентна функциям `strcpy` либо `strlen` и описывать параметры этих циклов. Наличие информации о таких циклах позволяет распространять семантику символьной длины строки не только через функции работы со строками, но и через эквивалентные им циклы.

Обнаружение циклов работы со строками происходит в несколько этапов. На первом этапе для каждого цикла определяется множество индуктивных переменных и характер их изменения. Под переменной понимается регистр или ячейка памяти с постоянным адресом, используемая в конкретной инструкции. Для таких переменных анализируются значения переменной на последовательных итерациях цикла. На основе анализа нескольких итераций цикла делается вывод о характере изменения значения переменной. Если значение переменной изменяется линейно с одним и тем же шагом на всех итерациях цикла, переменная включается в множество индуктивных переменных цикла.

Далее для каждого цикла определяется множество буферов в памяти, соответствующих следующим критериям:

- переменная, используемая в качестве адреса ячейки памяти, является индуктивной;
- на каждой итерации цикла происходит обращение к последовательным адресам памяти;
- размеры обрабатываемых ячеек соответствуют размеру символа для одного из типов строк (1 или 2 байта) и равны шагу переменной, используемой в качестве адреса;
- данные буфера представляют собой нуль-терминированную строку.

На этом этапе фиксируется характер доступа к каждому буферу (чтение и/или запись), его размер, а также размер адресуемой ячейки.

На основе полученных данных производится классификация циклов на принадлежность к одному из видов. Цикл считается циклом копирования строк, если выполнены следующие критерии:

– существует два буфера, к которым осуществляется доступ на каждой итерации цикла;
– к первому буферу обращаются только на чтение;
– ко второму буферу обращаются только на запись;
– на каждой итерации цикла значения, прочитанные из первого буфера, и значения, записанные во второй, совпадают.

Цикл считается циклом вычисления длины строки, если выполнены следующие критерии:

– существует один буфер, к которому осуществляется доступ на каждой итерации цикла;
– к буферу обращаются только на чтение;
– значение одной из индуктивных переменных после последней итерации цикла равно фактической длине строки, находящейся в буфере.

Все остальные циклы классифицируются как циклы с неизвестной семантикой. Если в цикле происходит одновременно копирование строки и вычисление её длины, то циклу присваиваются оба класса. В дальнейшем различные семантики такого цикла обрабатываются независимо друг от друга.

Циклы копирования и вычисления длины строк маркируются в трассе аналогично вызовам функций `strcpy` и `strlen`. Для каждого такого цикла определяются его границы и описываются значения фактических параметров, как если бы цикл являлся вызовом соответствующей функции. Такой подход позволяет единообразно обрабатывать как вызовы строковых функций, так и эквивалентные им циклы.

2.4 Увеличение покрытия кода

Недостатком анализа последовательности инструкций для одного запуска программы является отсутствие возможности итеративного анализа путей выполнения. Этот недостаток можно компенсировать возможностью анализа нескольких запусков исследуемой программы. Для увеличения вероятности обнаружения ошибки имеет смысл подбирать входные данные для этих запусков так, чтобы максимизировать суммарное покрытие кода исследуемой программы. В предлагаемом методе за основу берётся один запуск программы, для которого известен набор входных данных. Для этого запуска выполняется изменение пути выполнения программы с порождением большого количества новых путей, отличных от исходного пути. Для каждого из этих путей оценивается покрытие кода и генерируется набор входных данных, требуемый для прохождения программы по этому пути.

Метод основан на динамическом символьном выполнении, выполняющемся в онлайн-режиме. В процессе символьного выполнения назначаются символьные значения заданным переменным программы и эти символьные значения распространяются по мере выполнения программы, при этом все преобразования в программе, в которых участвуют значения, транслируются в соответствующие уравнения. Для хранения символьных значений переменных программы поддерживается *состояние*, описывающее отображение множества переменных программы на множество соответствующих им символьных значений, а также выполняемую на данный момент операцию программы. Если в процессе выполнения встречается ветвление, зависящее от символьных данных, порождается два состояния, соответствующие двум веткам ветвления, и выполнение продолжается дальше для каждого из состояний.

Обычно символьные значения назначают ячейкам памяти, содержащим входные данные программы. В этом случае различные состояния соответствуют различным путям выполнения в программе при обработке входных данных. С помощью SMT-решателя для каждого состояния и соответствующего ему пути можно получить подтверждающий набор входных данных. Кроме того, если производится выполнение бинарного кода, часто можно вычислить достигнутое покрытие кода в терминах некоторой метрики покрытия. Анализ покрытия кода

для каждого из состояний позволяет получить набор входных данных, для которого, в совокупности, достигается существенный прирост покрытия кода по сравнению с единичным запуском программы.

Пути выполнения, соответствующие порождаемым состояниям, представляют собой древовидную структуру. Это приводит к тому, что покрытие кода, соответствующее двум соседним состояниям, отличается незначительно. Кроме того, с каждым ветвлением количество состояний в программе растёт по экспоненциальному закону, что приводит к огромному количеству анализируемых состояний и неэффективности анализа трассы, содержащей по одному запуску программы для каждого из состояний. Для решения этой проблемы можно выделить такое подмножество состояний, совокупное покрытие кода для которого будет таким же, как и для всего множества состояний. Это возможно сделать с помощью классической задачи о покрытии множества. Задача о покрытии относится к классу NP-полных, но может быть эффективно решена приближенным алгоритмом, который даёт приемлемый результат. Таким способом на практике удаётся уменьшить количество рассматриваемых состояний на несколько порядков.

Решение данной задачи представлено в алгоритме 1. Этот алгоритм находит такое подмножество путей, на котором достигается такое же совокупное покрытие кода, как и на полном наборе путей. На вход алгоритм получает отображение из номера состояния в покрытие, выраженное в виде множества элементов покрытия. Данное множество содержит некоторый элемент покрытия, если на соответствующем пути программы было выполнено некоторое условие покрытия. К примеру, для метрики покрытия по базовым блокам элемент покрытия соответствует покрытому базовому блоку, а для метрики покрытия условий – комбинации значений переменных в булевом выражении. В результате работы алгоритма вычисляется набор номеров состояний, который затем используется для получения входных данных для каждого из этих состояний.

Поскольку, в общем случае, процесс перебора путей выполнения может проводиться неопределённо долго, требуется критерий завершения процесса перебора путей. Таким критерием может быть оценка прироста покрытия за определённый период времени. Если за заданный интервал времени прирост покрытия в терминах базовых блоков оказался меньше, чем некоторое пороговое значение, перебор путей завершается.

Алгоритм 1 Минимизация множества состояний

procedure MinimizeStateSet($coverageSet$)
    $result \leftarrow \varnothing$
    $covered \leftarrow \varnothing$
    loop
        $deltaSets \leftarrow [\forall i : coverageSet[i] - covered]$
        $maxIndex \leftarrow maxind_i(|deltaSets[i]|)$
        $deltaSet \leftarrow deltaSets[maxIndex]$
        if $deltaSet = \varnothing$ **then**
            break
        end if
        $covered \leftarrow covered \cup deltaSet$
        $result \leftarrow result \cup maxIndex$
    end loop
    return $result$
end procedure

Глава 3. Программная реализация

В качестве платформы для реализации методов выбрана среда анализа бинарного кода [4; 48; 49], разрабатываемая в ИСП РАН. Среда имеет модульную расширяемую архитектуру и предназначена для решения широкого класса задач обратной инженерии. С точки зрения решаемой задачи данная среда анализа обладает следующими преимуществами:

- Поддерживает быструю разработку модулей расширения, которые могут обращаться к другим алгоритмам анализа с помощью интерфейсов.
- Среда анализа является архитектурно-независимой, позволяет проводить анализ, не вдаваясь в детали реализации конкретной архитектуры. На данный момент среда анализа поддерживает процессорные архитектуры x86, x86-64, ARM, MIPS, PowerPC, а также трансляцию машинных команд этих архитектур в промежуточное представление Pivot.
- Среда анализа позволяет абстрагироваться от анализируемых гостевых операционных систем. На данный момент поддерживается разметка процессов и потоков для ОС Windows и Linux, а также существует универсальный механизм разметки, позволяющий размечать процессы и потоки для других операционных систем, что позволяет анализировать программное обеспечение различных маршрутизаторов.
- В среде анализа уже реализованы многие алгоритмы, позволяющие получить данные, необходимые для анализа, такие как список функций, список вызовов и их параметров, список циклов, список и размещение исполняемых модулей.

Архитектура реализованной системы приведена на рисунке 3.1.

Для получения трасс используется полносистемный эмулятор QEMU с механизмом детерминированного воспроизведения. Этот механизм позволяет скрыть от гостевой операционной системы замедление работы эмулятора, обусловленное трассировкой.

Анализируемая трасса выполнения содержит:

- инструкции, выполнявшиеся на процессоре, и их двоичный код;
- значения регистров процессора перед выполнением каждой инструкции;
- список прерываний (номер шага трассы и тип прерывания);
- информацию о содержимом буферов ассоциативной трансляции (TLB);

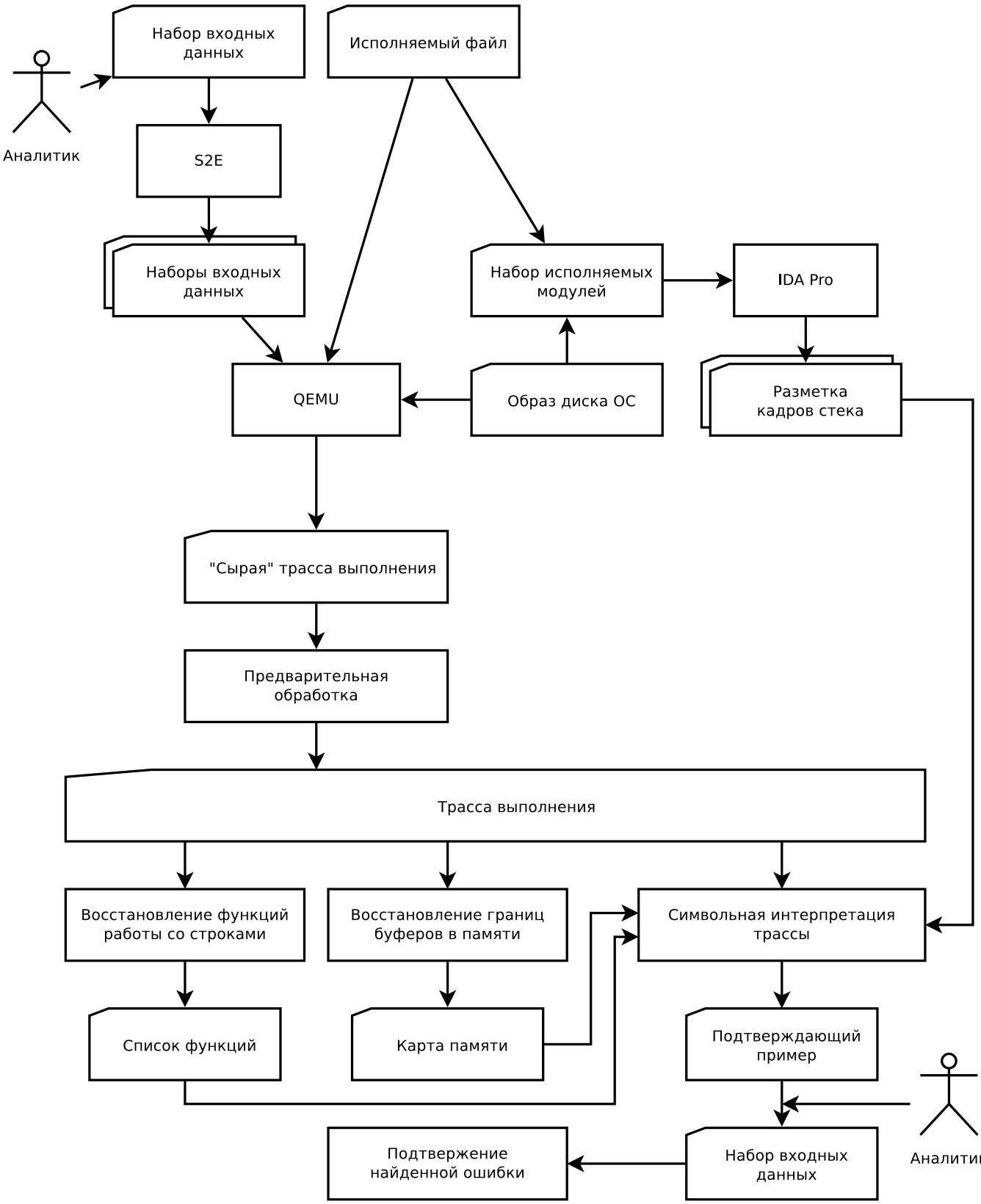

Рисунок 3.1 — Архитектура системы.

– информацию об асинхронных доступах к памяти с помощью DMA.

3.1 Повышение уровня представления

Подготовка заключается в повышении уровня представления: выполняется предварительная обработка, в ходе которой восстанавливаются высокоуровневые сущности в трассе. Среди используемых алгоритмов предварительной обработки можно выделить следующие.

– Алгоритм разметки процессов и потоков – позволяет сопоставить каждому шагу трассы идентификатор соответствующего ему процесса и потока. Информация о разметке процессов и потоков используется в других алгоритмах анализа, в частности в алгоритме поиска модулей.
– Алгоритм определения границ прерываний – позволяет определить входы и выходы из прерываний с учётом переключений процессов и потоков. Эта информация используется в других алгоритмах при анализе потока управления для ”перешагивания”через функции-обработчики прерываний.
– Алгоритм поиска вызовов в трассе – позволяет определить границы вызовов (шаги вызова и возврата) в трассе.
– Алгоритм получения информации о статической памяти – строит карту статической памяти процессов, которая потом используется для поиска модулей.
– Алгоритм поиска модулей – устанавливает соответствие между инструкцией, выполняющейся на каждом шаге трассы и исполняемым модулем, которому она может принадлежать.
– Алгоритм построения графа потока управления. Используется для извлечения статической информации о функциях.
– Алгоритм построения графа вызовов – определяет множество функций, которые выполнялись в трассе, а также строит граф вызовов этих функций.
– Алгоритм восстановления буферов в трассе. Сохранение всех значений оперативной памяти в процессе трассировки является слишком затратной

процедурой, поэтому трасса не содержит значений ячеек памяти. Алгоритм позволяет восстановить значения некоторых буферов в памяти с помощью анализа инструкций, которые обращались к этим буферам.

– Алгоритм поиска циклов – устанавливает границы циклов, а также число итераций для каждого выполненного в трассе цикла.

3.2 Подсистема интерпретации трассы с учётом символьной длины буферов

3.2.1 Поиск точек получения входных данных

Перед началом анализа определяются источники входных данных, которые должны быть представлены в виде пар *{шаг трассы, буфер в памяти}*. Эти пары могут быть непосредственно заданы аналитиком, однако можно получить необходимую информацию с помощью анализа вызовов функций, с помощью которых данные попадают в исследуемую программу. Среди таких функций можно отметить следующие:

– функции чтения из файла (`read`, `ReadFile`);
– функции чтения из сокета (`recv`, `WSARecv`);
– функция `main` в программе (либо точка входа в программу) в случае получения данных из командной строки или переменных окружения.

Такой набор функций позволяет описать получение данных из файлов, сетевых сокетов, командной строки и переменных окружения. Для каждого вызова этих функций в трассе определяется буфер с данными исходя из значений передаваемых в функцию параметров, а также определяется шаг трассы, на котором эти данные актуальны: таким шагом может быть шаг вызова или возврата соответствующей функции.

Для упрощения задачи сборки данных в файл используется следующий подход. Трассы выполнения исследуемой программы получаются таким образом, чтобы все данные, являющиеся входными для программы, считывались из одного или нескольких файлов. Данные, считываемые из этих файлов промежуточными программами, не должны изменяться до попадания их в программу. Далее по трассе выполнения запускается алгоритм анализа помеченных данных, который

восстанавливает отображение между данными во входных файлах и данными, попадающими в исследуемую программу с помощью одной из функций, описанных выше. В результате такого анализа для каждого байта входных данных в исследуемой программе становится известен файл и смещение в файле, откуда этот байт был получен. Кроме того, для каждого входного файла создаётся символьный буфер соответствующего размера. Это позволяет сформировать содержимое файла на основе значений для буфера, полученных в результате решения системы уравнений.

3.2.2 Отбор инструкций и вызовов функций

Трасса выполнения, полученная в результате полносистемной эмуляции, содержит не только инструкции исследуемой программы, но также отражает выполнение кода ядра операционной системы, а также других процессов, выполнявшихся во время получения трассы. Целесообразно анализировать только те инструкции, которые непосредственно участвуют в обработке входных данных. Такое сокращение области анализа позволяет сократить количество анализируемых инструкций (а значит, и время анализа) на несколько порядков. Для отбора анализируемых инструкций используется анализ потоков данных.

Подсистема символьной интерпретации использует подсистему анализа потоков данных, которая позволяет выполнять фильтрацию трассы по различным критериям. Подсистема позволяет менять конфигурацию анализа непосредственно во время её работы, в частности:

- отслеживать зависимости по данным в инструкциях трассы и отбирать инструкции, имеющие отношение к обработке отслеживаемых данных;
- добавлять регистры и ячейки памяти в множество отслеживаемых ячеек;
- удалять регистры и ячейки памяти из множества отслеживаемых ячеек;
- выполнять анализ инструкций, не относящихся к потоку данных, с помощью задания точек останова.

Отслеживание потоков данных происходит на уровне физических адресов памяти, что потенциально позволяет анализировать межпроцессное взаимодействие программ. Однако, анализ взаимодействия процессов сопряжён с некоторыми трудностями, поскольку в общем случае разным процессам соответствуют разные

адресные пространства и разные отображения виртуальных адресов в физические. Это приводит к необходимости выполнять символьную интерпретацию полностью в физических адресах. Поскольку операционная семантика машинных инструкций описывается в терминах виртуальных адресов, потребуется выполнение трансляции адресов при обработке практически каждой машинной команды, что является весьма затратной по времени операцией. Для увеличения скорости анализа в инструменте на данный момент реализована возможность обновления контекста только для одного процесса и соответствующего ему адресного пространства. Оптимизация процесса трансляции адресов и переход на символьную интерпретацию в физических адресах является темой дальнейших исследований.

Анализ трассы выполняется в рамках прямого прохода по трассе. Во время прохода, на соответствующих шагах трассы, в множество отслеживаемых ячеек добавляются буферы в памяти, относящиеся к источникам входных данных. Одновременно с этим в контексте символьной интерпретации для каждого такого буфера создаётся символьная переменная (битовый вектор), отражающая содержимое буфера, а также символьная переменная, отражающая его размер. Поскольку отслеживаются буферы с входными данными, результатом анализа потоков данных является последовательность инструкций, имеющих отношение к обработке отслеживаемых данных.

Для отбора функций с известной семантикой используется возможность задания точек останова: с их помощью производится дополнительный анализ на шагах вызова и возврата из функций. При этом все отобранные инструкции между шагом вызова и шагом возврата из функции исключаются из анализа.

Отобранные инструкции, а также вызовы функций передаются на анализ, описанный далее.

3.2.3 Трансляция инструкций

Трансляция и анализ инструкций выполняются параллельно работе подсистемы анализа потоков данных. Каждая инструкция, отобранная с помощью этой подсистемы, транслируется в набор инструкций промежуточного представления Pivot. Данное промежуточное представление позволяет описывать операционную семантику инструкций различных процессорных архитектур. Pivot-описание

каждой машинной инструкции представляет собой дерево операций в SSA форме, что позволяет легко проводить различные виды анализа над таким представлением.

Поскольку в современных процессорных архитектурах часто используются достаточно сложные инструкции, их эквивалентное Pivot-представление может содержать ветвления, отражающие возможные варианты выполнения инструкции. Несмотря на нетривиальные зависимости в коде, содержащем ветвления, он может быть представлен в виде уравнений с использованием операции ite (if-then-else), однако, такое представление значительно усложняет систему уравнений. Поэтому для интерпретации используется трасса Pivot-инструкций: подмножество Pivot-инструкций, которое было реально задействовано во время выполнения анализируемой инструкции на процессоре. Фактически, выполняется конкретизация всех условных переходов в Pivot-представлении анализируемой машинной инструкции. Для получения подмножества Pivot-инструкций выполняется интерпретация Pivot-кода: множество Pivot-инструкций транслируется в машинный код архитектуры x86_64 и затем выполняется на конкретных значениях регистров и памяти, доступных на момент выполнения соответствующей машинной инструкции.

3.2.4 Анализ инструкций и вызовов функций

Для каждой инструкции, отобранной с помощью анализа потоков данных, анализируется трасса Pivot-инструкций и выполняется обработка каждой Pivot-инструкции в соответствии с правилами, описанными в разделе 2.1.3. Обработка инструкций условного перехода непосредственно приводит к добавлению в предикат пути уравнений, отражающих выполнение перехода. Обработка остальных инструкций приводит к модификации отображения между регистрами и ячейками памяти на символьные значения.

Для каждого отобранного вызова функции выполняется обработка в соответствии с правилами, описанными в разделе 2.1.2. Добавление уравнений и выполнение проверок происходит на точках вызова и возврата из функции.

При обработке инструкций и вызовов функций также выполняются дополнительные проверки, описанные операторами $Assert$. Для каждого оператора

генерируется система уравнений, состоящая из предиката пути, а также инвертированного условия оператора $Assert$. Если полученная система уравнений оказалась совместной, фиксируется факт обнаружения ошибки работы с памятью.

3.2.5 Завершение анализа

Анализ исследуемой программы завершается в одном из двух случаев:

– исчерпана последовательность отобранных инструкций анализируемой программы;
– в исследуемой программе найдена ошибка работы с памятью.

Ошибка работы с памятью считается найденной, если одна из проверок, описанных операторами $Assert$ не прошла. В этом случае у решателя запрашивается модель данных, соответствующая символьным переменным, затем эта модель конкретизируется и на основе полученных данных генерируется подтверждающий пример в виде набора входных файлов. Поскольку в общем случае размеры входных файлов тоже являются символьными значениями, эти значения предоставляются аналитику вместе с входными файлами.

Для построения набора входных данных, приводящего к воспроизведению найденной ошибки, выполняется анализ полученного подтверждающего примера и при необходимости выполняется модификация данных в соответствии с новыми размерами файлов.

3.3 Подсистема восстановления границ буферов в памяти

В рамках данной подсистемы реализованы два алгоритма: для восстановления границ буферов в динамической и автоматической памяти.

3.3.1 Разметка динамической памяти

Составление карты динамической памяти основывается на использовании моделей функций [50].

Под моделью функции будем понимать функцию с описанными входными и выходными параметрами в виде ячеек памяти и регистров. Задаются три модели, каждая из которых соответствует функциям выделения (`alloc`), освобождения (`free`), и изменения размера уже выделенной памяти (`realloc`). Для каждой модели задаются параметры, имеющие определенную семантику. Для модели `alloc` задаётся размер (входной параметр) и адрес выделенной памяти (выходной параметр). Для модели `free` задаётся адрес освобождаемой памяти (входной параметр). Для модели `realloc` задаётся адрес буфера, размер которого будет изменён (входной параметр), новый размер (входной параметр) и адрес нового буфера (выходной параметр). После того, как модели заданы для каждого экземпляра в трассе, происходит обновление карты динамической памяти в соответствии с семантикой модели. Стоит отметить, что в исследуемой программе может быть несколько вложенных менеджеров памяти, для работы с которыми используются разные наборы библиотечных функций. Для отличия областей выделенной памяти используется идентификатор менеджера памяти. Карта динамической памяти реализована в виде последовательности кортежей *{идентификатор менеджера памяти, шаг трассы при создании буфера, шаг трассы при удалении буфера, идентификатор процесса, идентификатор потока, адрес начала буфера, размер буфера}*. Модель функции, связанная с конкретным вызовом этой функции в трассе, называется экземпляром модели этой функции.

Обработка экземпляра модели `alloc` добавляет кортеж в карту памяти, инициализируя все значения, кроме шага трассы на котором происходит удаление буфера.

Обработка экземпляра модели `free` добавляет в кортеж шаг трассы, на котором происходит удаление буфера.

Обработка экземпляра модели `realloc` является комбинацией обработки предыдущих двух моделей. Сначала записывается шаг трассы, на котором входной буфер удаляется, затем создаётся кортеж, описывающий новый буфер. Адрес и размер нового буфера соответствуют параметрам модели `realloc`.

С помощью обработки всех экземпляров моделей по описанным выше правилам составляется разметка для динамической памяти.

3.3.2 Разметка автоматической памяти

Для восстановления границ буферов в автоматической памяти используется информация из статического представления программных модулей.

Создание карты автоматической памяти происходит в два этапа:

– получение информации о кадрах стека для каждого исполняемого модуля при помощи IDA Pro;
– отображение полученной информации на трассу.

С помощью статического анализатора IDA Pro [51] производится предварительный анализ для каждого модуля, присутствующего в трассе. В ходе анализа автоматически восстанавливаются границы многих функций, а также карты стекового фрейма этих функций. Для каждой функции собирается информация, необходимая для восстановления границ фреймов в трассе:

– имя модуля;
– смещение функции внутри модуля;
– размер фрейма функции;
– смещение ячейки внутри фрейма, содержащей адрес возврата.

Собранная информация сопоставляется с трассой. Для каждого вызова функции, для которой есть информация о стековом фрейме, производится отображение границ фрейма на фактические адреса в памяти. Для каждого такого вызова создаётся кортеж в карте, аналогичный кортежу в карте динамической памяти. Шагом создания буфера является шаг вызова функции, а шагом удаления является шаг возврата из функции.

Совокупность разметок автоматической и динамической памяти используется при дальнейшем анализе.

3.4 Подсистема восстановления циклов работы со строками

Для своей работы подсистема использует список циклов, которые были восстановлены в трассе на этапе предобработки. Восстановление циклов происходит в два этапа. Сначала восстанавливаются циклы на основе статико-динамического представления программы в среде анализа. Затем найденные циклы отображаются на трассу: для каждого экземпляра цикла в трассе вычисляются границы цикла в терминах позиций трассы, а также число итераций.

Рассматриваются циклы, в которых выполнялись как минимум две итерации. Для каждого такого экземпляра цикла выполняется поиск индуктивных переменных. Для этого анализируются значения регистров, а также адреса и значения элементов в памяти, с которыми работали инструкции, принадлежащие циклу. На первых двух итерациях цикла строятся предположения относительно характера изменения переменных. Если на последующих итерациях это предположение нарушается, соответствующие переменные исключаются из рассмотрения. Кроме того, выполняется классификация и определение параметров цикла в соответствии с правилами, описанными в разделе 2.3.

Результатом проведения описанных выше действий будет список экземпляров циклов и значений их параметров, который в дальнейшем используется для анализа целиком экземпляров циклов по аналогии с анализом вызовов библиотечных функций.

3.5 Реализация метода увеличения покрытия кода

В качестве используемого инструмента для онлайн символьного выполнения выступает S^2E. Этот инструмент использует эмулятор `QEMU` в качестве среды выполнения и позволяет выполнять анализ с помощью модулей расширения. В качестве гостевых операционных систем поддерживаются ОС семейства Windows и Linux. Управление символьным анализом поддерживается как изнутри эмулятора с помощью специальных инструкций процессора, так и снаружи с помощью соответствующих интерфейсов для модулей расширения. В данной работе управление работой инструмента осуществляется с помощью модуля `Annotation`,

который позволяет управлять инструментом с помощью скриптов на языке Lua. Этот модуль позволяет задавать точки останова и обрабатывать их с помощью функций-обработчиков. В обработчиках доступны многие базовые возможности S^2E, в частности добавление и удаление символьных пометок, а также завершение текущего состояния. С помощью таких скриптов аналитиком задаются правила для обработки исследуемой программы: точки останова для внедрения символьных данных, точки для предварительного завершения программы.

S^2E поддерживает возможность трассировки и позволяет, в частности, сохранять трассу базовых блоков, а также конкретные значения символьных переменных для завершённых путей. Эта информация используется для вычисления покрытия на каждом пути. Информация о покрытии для каждого пути затем используется для минимизации множества состояний. Полученное множество состояний используется для получения входных данных для каждого из этих состояний. По набору входных данных формируется скрипт запуска исследуемой программы, который используется для получения трассы, содержащей все эти запуски.

В данной работе используется метрика покрытия кода по базовым блокам, так как использование этой метрики оказывается достаточным для поиска ошибок, привязанных к конкретному месту программы. В большинстве случаев выход за границы буфера происходит из-за отсутствия проверок на размер буфера непосредственно перед операцией копирования. Для поиска таких ошибок требуется покрыть как можно большее количество мест в программе, в которых происходит копирование данных, что, в свою очередь, можно свести к задаче получения хорошего покрытия в терминах базовых блоков.

Алгоритм поиска ошибок выхода за границы буфера поддерживает одновременный анализ нескольких запусков программы в трассе, для этого в среде анализа бинарного кода запускается отдельный поток обработки для каждого процесса, найденного в трассе. Для предотвращения чрезмерной нагрузки на систему количество одновременно работающих потоков ограничивается количеством ядер процессора.

3.6 Технические ограничения реализации

К техническим ограничениям реализации можно отнести недостатки используемой среды анализа бинарного кода. Основным её недостатком является отсутствие информации о значениях в ячейках памяти, что приводит к необходимости косвенно восстанавливать эти значения с помощью анализа инструкций доступа к памяти. Восстановление значений ячеек памяти вносит значительное замедление в работу алгоритмов, а в некоторых случаях не позволяет восстановить значения, например, в случае отсутствия доступа к ячейкам памяти в трассе. Это ограничение можно обойти, если дополнить формат трассы данными из памяти, получаемыми непосредственно из эмулятора во время трассировки.

Также в среде анализа реализована поддержка архитектур x86, x86_64, ARM, PowerPC, MIPS64 для трансляции во внутреннее представление Pivot, однако подсистема символьной интерпретации трассы на данный момент поддерживает только архитектуры x86, x86_64.

Ещё одним ограничением является область применимости инструмента S^2E. На данный момент этот инструмент поддерживает лишь гостевые операционные системы Windows и Linux, причём поддерживаются не все версии Windows из-за относительно старой версии QEMU, используемой в S^2E. Кроме того, S^2E поддерживает ограниченный набор процессорных архитектур: x86, x86_64 и ARM.

Среда символьного выполнения, реализованная в S^2E, также обладает некоторыми недостатками:

- ограничение возможности распространения символьных пометок через регистры сопроцессора и XMM регистры;
- сложности анализа сетевых служб: отсутствует поддержка подсистемы SLIRP, традиционно используемой в QEMU для предоставления доступа к сети;
- подсистема анализа загрузки/выгрузки исполняемых модулей полноценно поддерживает только Windows XP SP3.

Глава 4. Результаты применения

Работа системы оценивалась на рабочей станции с процессором Core i5-2500 и 16 Гб оперативной памяти, работающей под управлением ОС Windows. Под гостевые ОС во время трассировки выделялось 128 Мб оперативной памяти. В качестве объектов анализа выступали приложения, работающие под управлением операционных систем семейства Windows и Linux, а также встроенное программное обеспечение для сетевого маршрутизатора. Четыре программы из этого списка содержат уже известные уязвимости. Список анализируемых программ приведён в табл. 1. В табл. 2 приведены результаты работы алгоритма. Для всех исследуемых программ время работы SMT-решателя не превышало одной секунды, большая часть времени работы уходила на построение предиката пути. Ввиду технических ограничений, метод расширения покрытия применялся только к программам, работающим под ОС Linux. При получении результатов было добавлено дополнительное ограничение сверху на абстрактную длину буфера, ограничивающее длину значением 4096. Без этого ограничения SMT-решатель может подобрать слишком большое значение для абстрактной длины буфера, которое не позволит создать в памяти буфер такого размера.

В двух программах удалось обнаружить переполнение буфера на куче. Для многих из исследуемых программ были обнаружены ошибки доступа к памяти, связанные с записью входных данных, размер которых можно контролировать, в буфер фиксированного размера. Однако, для некоторых программ (`OpenSSL` и `httpdx`) были обнаружены ошибки доступа к памяти другого характера.

Далее рассматриваются различные ситуации в программах, для которых удалось обнаружить ошибку выхода за границы буфера.

4.1 Ошибка в GoldMp4Player

В приложении `GoldMp4Player` была обнаружена ошибка переполнения буфера [52], расположенного в динамической памяти. `GoldMp4Player` является программой для проигрывания аудио- и видеозаписей, работающей под управлением ОС Windows. Программа позволяет отрывать файлы не только с диска, но

Таблица 1 — Список анализируемых программ

ОС	Приложение	Версия приложения	CVE / OSVDB
Linux	iwconfig	iwconfig v26	CVE: 2003-0947
Linux	get_driver	sysfsutils 2.1.0	–
Linux	mkfs.jfs	jfsutils 1.1.15	–
Linux	alsa_in	jack 0.124.1	–
Linux	openssl	openssl 1.0.0f	CVE: 2014-0160
Windows XP SP2	httpdx	httpdx 1.5.4	OSVDB-ID: 84454
Windows XP SP2	GoldMP4Player	GoldMP4Player 3.3	OSVDB-ID: 103826
Linux	faad	faad2 2.7	–
Linux	gencnval	icu 54.1	–
Linux	lou_checktable	liblouis 2.5.2	–
Linux	mysql_plugin	mariadb 10.0.17	–
VxWorks	ПО маршрутизатора	–	–

Таблица 2 — Результаты работы инструмента

Приложение	Размер префикса	Размер входных данных	Тип доступа к памяти	Память	Время работы, с
iwconfig	32	93	запись	стек	3
get_driver	14	489	запись	стек	4
mkfs.jfs	31	436	запись	стек	2
alsa_in	34	355	запись	стек	4
openssl	18	25	чтение	стек	5
httpdx	329	330	запись	куча	338
GoldMP4Player	36	505	запись	куча	255
faad	13	302	запись	стек	<1
gencnval	13	574	запись	стек	<1
lou_checktable	15	527	запись	стек	8
mysql_plugin	16	578	запись	стек	14
ПО маршрутизатора	90	91	чтение	куча	64

и загружать их по протоколу HTTP. Ошибка может проявиться при использовании возможности открытия файла по HTTP-ссылке. Для поиска ошибок в данной программе использовалась следующая ссылка:

```
http://AAAAAA.swf
```

Псевдокод ошибочной ситуации представлен на листинге 4.1. Ссылка считывается из поля ввода программы с помощью функции `GetWindowTextA`. Далее текст ссылки копируются в буфер фиксированного размера, выделенный в динамической памяти, с помощью функции `lstrcpy` из библиотеки `kernel32.dll`, которая аналогична функции `strcpy` из стандартной библиотеки языка Си.

Листинг 4.1 Псевдокод ошибочной ситуации

```
1 ptr = malloc(0x210);
2 data = GetWindowTextA(...);
3 lstrcpy(ptr+0x18, &data);
```

Ошибка выхода за границы буфера обнаруживается во время анализа функции `lstrcpy`. Проверка корректности доступа по записи приводит к обнаружению ошибочной ситуации. Размер копируемых данных равен размеру L-буфера из множества I и на него не накладываются никакие ограничения в программе. Поскольку размер буфера, в который копируются данные, фиксирован, существует значение символьной длины L-буфера, при котором произойдёт переполнение. Для обнаружения выхода за границы буферов на куче использовалась карта выделенной памяти, полученная на основе анализа вызовов функций менеджера кучи (`malloc`, `free`, `realloc`). Наличие такой карты позволяет отслеживать множество выделенных буферов для любого шага трассы.

4.2 Ошибка в httpdx

В приложении `httpdx` был обнаружен особый случай ошибки переполнения буфера: переполнение [53] происходило не из-за большого размера записываемых данных, а из-за недостаточного размера выделенной памяти. Приложение `httpdx` является легковесным веб-сервером для ОС Windows и обрабатывает запросы по протоколу HTTP. В качестве клиентского приложения выступал

скрипт на языке Python, с помощью которого осуществлялась отправка следующего HTTP-запроса:

```
POST /test.pl HTTP/1.0
Content-Length: 100
Content-Type: text
Host: 192.168.1.2

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Псевдокод ошибочной ситуации представлен на листинге 4.2. В приложении `httpdx` отсутствует контроль размеров копируемых данных при обработке HTTP-запроса. Если размер данных (`size(content)`) как минимум на два байта превышает значение поля `Content-Length`, произойдёт переполнение буфера `buffer` при копировании данных.

Листинг 4.2 Псевдокод ошибочной ситуации

```
1 content_length, content = parse(packet)
2 buffer = malloc(content_length + 1)
3 memcpy(buffer, content, size(content))
```

Ошибка выхода за границы буфера обнаруживается во время анализа вызова функции `memcpy`. Проверка корректности доступа по записи приводит к обнаружению ошибочной ситуации, поскольку размер копируемых данных (`size(content)`) может оказаться больше, чем размер L-буфера из множества A, соответствующего области памяти, выделенной функцией `malloc`. Проверка корректности доступа по чтению проходит успешно, так как количество копируемых данных всегда меньше размера L-буфера из множества I, соответствующего сетевому пакету.

Несмотря на тот факт, что значение поля `Content-Length` передаётся в текстовом виде, система символьного выполнения порождает уравнения, связывающие текстовый эквивалент значения и размер, передаваемый в качестве параметра функции `malloc`, что позволяет обратить преобразования, выполненные в программе и получить такой HTTP-запрос, при обработке которого произойдёт переполнение буфера. Следует отметить, что для найденной ошибки подтверждающий пример генерируется таким образом, что ошибка проявляется не из-за увеличения размера HTTP-запроса, а из-за уменьшения размера буфера, выделенного для хранения данных этого запроса.

4.3 Ошибка в OpenSSL (Heartbleed)

На примере библиотеки `libssl` из `OpenSSL` было продемонстрировано обнаружение известной уязвимости Heartbleed [54; 55]. Код библиотеки анализировался в рамках запуска приложения `openssl`, использующего эту библиотеку. Приложение `openssl` было запущено в режиме веб-сервера с помощью следующей строки запуска:

```
openssl s_server -cert serv.crt -key serv.key -WWW -accept 443 -tls1
```

В качестве клиентского приложения использовался скрипт на языке Python, устанавливающий соединение и отправляющий пакет Heartbeat, содержащий корректные значения полей.

В коде библиотеки `libssl`, отвечающем за обработку TLS-пакета Heartbeat, отсутствует контроль размеров копируемых данных. Псевдокод ошибочной ситуации представлен на листинге 4.3. Вызов функции `memcpy` может привести к выходу за границы буфера `buf1` по чтению, так как значение `size_from_packet` берётся из сетевого пакета и может принимать любые значения от 0 до 65535. Контролируя значение этого поля, злоумышленник может прочитать данные из областей памяти, которые находятся рядом с областью, выделенной под буфер `buf1`, что может привести к утечке чувствительных данных.

Листинг 4.3 Псевдокод ошибочной ситуации

```
1 size = recv(&buf1)
2 size_from_packet = buf1[1..2]
3 buf2 = malloc(size_from_packet)
4 memcpy(buf2, buf1, size_from_packet)
5 send(&buf2)
```

Ошибка выхода за границы буфера обнаруживается во время анализа вызова функции `memcpy`. При этом выполняется две проверки: на корректность доступа по чтению и на корректность доступа по записи. Первая проверка приводит к обнаружению ошибочной ситуации, так как размер копируемых данных может оказаться больше, чем размер L-буфера из множества I, который соответствует полученному сетевому пакету. Вторая проверка проходит успешно, так как размер копируемых данных в точности равен размеру L-буфера из множества A, который был создан во время обработки вызова функции `malloc`.

Подобные ошибки трудно обнаружимы с помощью фаззинга, так как чтение из памяти не приводит к аварийному завершению. В то же время, с помощью символьной интерпретации длин буферов возможно обнаружение ошибок выхода за границы буфера при чтении. Кроме того, в этом примере, так же как и в приложении `httpdx`, для срабатывания ошибки требуется не увеличение размера входных данных, а изменение поля длины, передаваемого среди входных данных.

4.4 Ошибка во встроенном ПО маршрутизатора

На данном примере демонстрируются возможности полносистемного анализа на примере обнаружения ошибки выхода за границы буфера во встроенном ПО маршрутизатора. Система символьного выполнения использует анализ потоков данных на уровне физических адресов, что позволяет отслеживать прохождение данных от точки получения сетевого пакета до точки обработки сообщения протокола высокого уровня. В данном примере в качестве источника входных данных использовалась функция сетевого драйвера, в которую передаются сетевые пакеты для последующей обработки.

Псевдокод ошибочной ситуации представлен на листинге 4.4. С маршрутизатором устанавливалось PPTP-соединение, в рамках которого происходила аутентификация по протоколу CHAP. Функция, обрабатывающая содержимое аутентификационного пакета Response, копирует отдельные поля пакета в локальные переменные. Размер копируемых данных извлекается непосредственно из сетевого пакета и передаётся в функцию копирования без каких-либо проверок, что может привести к ошибке выхода за границы буфера при чтении.

Листинг 4.4 Псевдокод ошибочной ситуации

```
1 valueSize = packet[0];
2 valuePtr = &packet[1];
3 memcpy(ptr1, valuePtr, valueSize);
4 namePtr = &packet[1 + valueSize];
5 nameSize = packetSize – (1 + valueSize);
6 if (nameSize < 0) return;
7 memcpy(ptr2, namePtr, nameSize);
```

Ошибка выхода за границы буфера обнаруживается во время анализа первого вызова функции `memcpy`. При этом выполняется две проверки: на корректность доступа по чтению и на корректность доступа по записи. Первая проверка приводит к обнаружению ошибочной ситуации, так как размер копируемых данных может оказаться больше, чем размер L-буфера (за вычетом размеров заголовков пакета) из множества I, который соответствует полученному сетевому пакету. Вторая проверка проходит успешно, так как размер копируемых данных не может превышать 255 байтов и поэтому не приводит к переполнению буфера по указателю `ptr1`, размер которого составляет 256 байтов.

Несмотря на сходства с ошибкой в `OpenSSL`, в данном примере эксплуатация ошибки не может привести к утечке данных: при передаче по сети ошибочного значения в поле размера сработает проверка в строке 6 и разбор пакета будет приостановлен с последующим закрытием сетевого соединения.

Заключение

Основные результаты работы заключаются в следующем.

1. Разработан метод поиска ошибок выхода за границы буфера на основе символьной интерпретации трассы с использованием абстрактной длины переменных-массивов, полученных по результатам обратной инженерии бинарного кода. Метод не требует наличия исходных кодов и отладочной информации и позволяет находить ошибки, даже если они не проявлялись в анализируемой трассе.
2. Разработаны методы, улучшающие точность поиска ошибок выхода за границы буфера за счёт выявления функций работы со строками, которые подверглись встраиванию в процессе компиляции и предварительного расширения покрытия кода при сборе трассы выполнения.
3. На основе предложенных методов разработано и реализовано программное средство для поиска ошибок выхода за границы буфера. Реализованные методы являются машинно-независимыми, а также абстрагированы от операционной системы, используемой для запуска анализируемой программы. Для определения границ буферов используется алгоритм, позволяющий получить оценочную информацию о границах буферов на основе анализа областей динамической и автоматической памяти.

В диссертации исследованы и разработаны новые методы, позволяющие находить ошибки переполнения буфера в бинарном коде программ. На основе предложенных методов был реализован набор инструментов, который позволяет находить ошибки и уязвимости, не выявляемые другими открытыми инструментами.

Среди особенностей разработанного метода, отличающих его от близких подходов, можно отметить следующие:

1. Для поиска ошибок используется символьная интерпретация, что позволяет находить новые виды ошибок, для проявления которых требуется выполнение сложных условий.
2. Возможность поиска ошибок в рамках полносистемного анализа приложений, в том числе для анализа встроенного программного обеспечения различных устройств.

3. Эффективно решаются проблемы, связанные с межпроцедурным анализом, а также проблема алиасинга, присущие многим инструментам анализа кода.
4. Метод позволяет находить ошибки доступа к памяти во время чтения данных, а не только во время записи, что открывает возможности для обнаружения новых видов ошибок.
5. Потенциальная возможность применения описанных методов для анализа программного обеспечения для различных процессорных архитектур за счёт использования машинно-независимого подхода.

Дальнейшие исследования по рассматриваемой теме могут быть связаны с интеграцией предложенных методов с методами статического анализа, а также исследование возможности применения предложенных методов в рамках подходов, основанных на онлайн-анализе бинарного кода.

Планы по применению разработанных методов включают в себя поиск ошибок в программном обеспечении современных маршрутизаторов, а также массовый поиск ошибок в программном обеспечении популярных дистрибутивов ОС Linux.

Список сокращений и условных обозначений

API application programming interface, интерфейс прикладного программирования

ARM advanced RISC machine, усовершенствованная RISC-машина

AST abstract syntax tree, абстрактное синтаксическое дерево

BSOD blue screen of death, отладочный механизм ядра ОС Windows

CFG control flow graph, граф потока управления

CHAP challenge-handshake authentication protocol, протокол аутентификации

DMA direct memory access, режим обмена данными между устройствами и памятью без участия ЦПУ

FP floating point, числа с плавающей точкой

HTTP hypertext transfer protocol, протокол передачи гипертекста

IR intermediate representation, промежуточное представление

JIT just-in-time compilation, компиляция «на лету»

LLVM low level virtual machine, инструментарий для разработки компиляторов

PCI peripheral component interconnect, шина ввода-вывода для подключения периферийных устройств

PPTP point-to-point tunneling protocol, туннельный протокол типа точка-точка

QEMU quick emulator, открытое ПО для эмуляции различных процессорных архитектур

RISC reduced instruction set computer, семейство архитектур с сокращённым набором команд

S^2E selective symbolic execution, платформа динамического символьно выполнения

SIMD single instruction, multiple data, принцип компьютерных вычислений, позволяющий обеспечить параллелизм на уровне данных

SMT satisfiability modulo theories, задача выполнимости формул в теориях

SPEC standard performance evaluation corporation, набор тестов, предназначенных для измерения производительности компьютеров

SQL structured query language, язык структурированных запросов

SSA static single assignment form, промежуточное представление, используемое компиляторами, в котором каждой переменной значение присваивается лишь единожды

SSE2 streaming SIMD extensions 2, набор инструкций для параллельных вычислений над данными

STP simple theorem prover, решатель ограничений, представитель SMT-решателей

TCP transmission control protocol, протокол управления передачей

TLB translation lookaside buffer, буфер ассоциативной трансляции

TLS transport layer security, протокол защиты транспортного уровня

UDP user datagram protocol, протокол пользовательских датаграмм

VMM virtual machine monitor, монитор виртуальных машин

XSS cross-site scripting, межсайтовый скриптинг

ВАК высшая аттестационная комиссия

ПО программное обеспечение

ЦПУ центральное процессорное устройство

Список литературы

1. *Padaryan VA, Kaushan VV, Fedotov AN*. Automated exploit generation for stack buffer overflow vulnerabilities // *Programming and Computer Software*. — 2015. — Vol. 41, no. 6. — Pp. 373–380.

2. *Каушан В. В., Мамонтов А. Ю., Падарян В. А. и др.* Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ // *Труды Института системного программирования РАН*. — 2015. — Т. 27, № 2. — С. 105–126.

3. *Каушан В. В.* Поиск ошибок выхода за границы буфера в бинарном коде программ // *Труды Института системного программирования РАН*. — 2016. — Т. 28, № 5. — С. 135–144.

4. *Падарян В. А., Каушан В. В., Гетьман А. И. и др.* Методы и программные средства, поддерживающие комбинированный анализ бинарного кода // *Труды Института системного программирования РАН*. — 2014. — Т. 26, № 1. — С. 251–276.

5. *Федотов А. Н., Каушан В. В., Падарян В. А. и др.* Поиск некоторых типов ошибок работы с памятью в бинарном коде программ // Материалы 24-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации». — 2015. — С. 103–105.

6. *Каушан В. В., Федотов А. Н.* Развитие технологии генерации эксплойтов на основе анализа бинарного кода // Материалы 24-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации». — 2015. — С. 77–79.

7. Valgrind. — URL: http://valgrind.org/ (дата обращения: 19.09.2017).

8. *Nethercote Nicholas, Seward Julian*. Valgrind: a framework for heavyweight dynamic binary instrumentation // ACM Sigplan notices / ACM. — Vol. 42. — 2007. — Pp. 89–100.

9. Massif: a heap profiler. — URL: http://valgrind.org/docs/manual/ms-manual.html (дата обращения: 12.12.2017).

10. Helgrind: a thread error detector. — URL: http://valgrind.org/docs/manual/hg-manual.html (дата обращения: 12.12.2017).

11. DRD: a thread error detector. — URL: http://valgrind.org/docs/manual/drd-manual.html (дата обращения: 12.12.2017).

12. Cachegrind: a cache and branch-prediction profiler. — URL: http://valgrind.org/docs/manual/cg-manual.html (дата обращения: 12.12.2017).

13. *Nethercote Nicholas, Mycroft Alan*. Redux: A dynamic dataflow tracer // *Electronic Notes in Theoretical Computer Science*. — 2003. — Vol. 89, no. 2. — Pp. 149–170.

14. *Newsome James, Song Dawn*. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software // In In Proceedings of the 12th Network and Distributed Systems Security Symposium / Citeseer. — 2005.

15. DynamoRIO. — URL: http://www.dynamorio.org/ (дата обращения: 19.09.2017).

16. *Bruening Derek, Zhao Qin*. Practical memory checking with Dr. Memory // Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization / IEEE Computer Society. — 2011. — Pp. 213–223.

17. Dynamic program analysis of microsoft windows applications / Alex Skaletsky, Tevi Devor, Nadav Chachmon et al. // Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on / IEEE. — 2010. — Pp. 2–12.

18. Pin – A Dynamic Binary Instrumentation Tool. — URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool (дата обращения: 19.09.2017).

19. SPEC CPU 2006. — URL: https://www.spec.org/cpu2006/ (дата обращения: 12.12.2017).

20. Unleashing mayhem on binary code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // Security and Privacy (SP), 2012 IEEE Symposium on / IEEE. — 2012. — Pp. 380–394.

21. clang: a C language family frontend for LLVM. — URL: https://clang.llvm.org/ (дата обращения: 12.12.2017).

22. *Lattner Chris*. LLVM and Clang: Next generation compiler technology // The BSD Conference. — 2008. — Pp. 1–2.

23. AddressSanitizer. — URL: https://clang.llvm.org/docs/AddressSanitizer.html (дата обращения: 12.12.2017).

24. ThreadSanitizer. — URL: https://clang.llvm.org/docs/ThreadSanitizer.html (дата обращения: 12.12.2017).

25. MemorySanitizer. — URL: https://clang.llvm.org/docs/MemorySanitizer.html (дата обращения: 12.12.2017).

26. UndefinedBehaviorSanitizer. — URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (дата обращения: 12.12.2017).

27. DataFlowSanitizer. — URL: https://clang.llvm.org/docs/DataFlowSanitizer.html (дата обращения: 12.12.2017).

28. LeakSanitizer. — URL: https://clang.llvm.org/docs/LeakSanitizer.html (дата обращения: 12.12.2017).

29. SafeStack. — URL: https://clang.llvm.org/docs/SafeStack.html (дата обращения: 12.12.2017).

30. EXE: A system for automatically generating inputs of death using symbolic execution / Cristian Cadar, Vijay Ganesh, Peter Pawlowski et al. // Proceedings of the ACM Conference on Computer and Communications Security. — 2006.

31. *Isaev IK, Sidorov DV*. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs // *Programming and Computer Software*. — 2010. — Vol. 36, no. 4. — Pp. 225–236.

32. Automated whitebox fuzz testing. / Patrice Godefroid, Michael Y Levin, David A Molnar et al. // NDSS. — Vol. 8. — 2008. — Pp. 151–166.

33. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. / Cristian Cadar, Daniel Dunbar, Dawson R Engler et al. // OSDI. — Vol. 8. — 2008. — Pp. 209–224.

34. STP Constraint Solver. — URL: http://stp.github.io/ (дата обращения: 31.10.2017).

35. *Chipounov Vitaly, Kuznetsov Volodymyr, Candea George*. S2E: A platform for in-vivo multi-path analysis of software systems // *ACM SIGPLAN Notices*. — 2011. — Vol. 46, no. 3. — Pp. 265–278.

36. *Bellard Fabrice*. QEMU, a fast and portable dynamic translator. // USENIX Annual Technical Conference, FREENIX Track. — 2005. — Pp. 41–46.

37. Loop-extended symbolic execution on binary programs / Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song // Proceedings of the eighteenth international symposium on Software testing and analysis / ACM. — 2009. — Pp. 225–236.

38. *Xu Ru-Gang, Godefroid Patrice, Majumdar Rupak*. Testing for buffer overflows with length abstraction // Proceedings of the 2008 international symposium on Software testing and analysis / ACM. — 2008. — Pp. 27–38.

39. *Падарян В. А., Каушан В. В., Федотов А. Н.* Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке // *Труды Института системного программирования РАН*. — 2014. — Т. 26, № 3. — С. 127–144.

40. *Довгалюк ПМ, Фурсова НИ, Дмитриев ДС*. Перспективы применения детерминированного воспроизведения работы виртуальной машины при решении задач компьютерной безопасности // *Системы высокой доступности*. — 2013. — Т. 9, № 3. — С. 046–050.

41. Применение программных эмуляторов в задачах анализа бинарного кода / ПМ Довгалюк, ВА Макаров, ВА Падарян и др. // *Труды Института системного программирования РАН*. — 2014. — Т. 26, № 1.

42. *Падарян ВА, Соловьев МА, Кононов АИ*. Моделирование операционной семантики машинных инструкций // *Труды Института системного программирования РАН*. — 2010. — Т. 19.

43. *Соловьев М.А.* Восстановление алгоритма по набору бинарных трасс: дис. ... канд. физ.-мат. наук: 05.13.11; [Место защиты: Федеральное государственное

бюджетное учреждение науки Институт системного программирования РАН]. — 2013.

44. *Тихонов А.Ю., Падарян В.А.* Применение программного слайсинга для анализа бинарного кода, представленного трассами выполнения // Материалы XVIII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». — 2009. — С. 131.

45. SMT-LIB. — URL: http://smtlib.cs.uiowa.edu/ (дата обращения: 03.08.2017).

46. *Schwartz Edward J, Avgerinos Thanassis, Brumley David*. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) // Security and privacy (SP), 2010 IEEE symposium on / IEEE. — 2010. — Pp. 317–331.

47. PF_RING ZC *ZeroCopy*. — URL: https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/ (дата обращения: 11.10.2017).

48. *Тихонов А.Ю., Аветисян А.И.* Комбинированный (статический и динамический) анализ бинарного кода // *Труды Института системного программирования РАН*. — 2012. — Т. 22.

49. *Тихонов А.Ю., Аветисян А.И., Падарян В.А.* Методика извлечения алгоритма из бинарного кода на основе динамического анализа // *Проблемы информационной безопасности. Компьютерные системы*. — 2008. — № 3. — С. 73–79.

50. *Падарян Вартан Андроникович*. О представлении результатов обратной инженерии бинарного кода // *Труды института системного программирования РАН*. — 2017. — Т. 29, № 3. — С. 31–42.

51. IDA. — URL: https://www.hex-rays.com/products/ida/ (дата обращения: 12.12.2017).

52. Gold MP4 Player 3.3 - Buffer Overflow. — URL: https://www.exploit-db.com/exploits/31914/ (дата обращения: 12.12.2017).

53. httpdx 1.5.4 - Remote Heap Overflow. — URL: https://www.exploit-db.com/exploits/20120/ (дата обращения: 12.12.2017).

54. The Heartbleed Bug. — URL: http://heartbleed.com/ (дата обращения: 11.10.2017).

55. CVE-2014-0160. — URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160 (дата обращения: 11.10.2017).

Список рисунков

Список таблиц

Приложение А

Примеры описаний на языке Pivot

Листинг А.1 Исходный код описания машинной команды ADC

```
/* Case 1.  */
match "ADC" pointer #, auto # with i8, i16, i32, i64
begin
    /* Test CF.  */
    discard and.i16(r:flags, (i16) 0x0001)

    /* Branch.  */
    branch.nz L1

    /* CF is not set, so just add.  */
    $1 = add.#($1, $2)

    /* Branch to end.  */
    branch L2

label L1:
    /* CF is set, add and increment.  */
    $1 = addi.#($1, $2)

label L2:
    /* Update flags.  */
    r:flags = x86.uf(r:flags, (i16) 0x08D5)
end


/* Case 2.  */
match "ADC" pointer #, const i8 with i16, i32, i64
begin
    /* Test CF.  */
    discard and.i16(r:flags, (i16) 0x0001)

    /* Branch.  */
    branch.nz L1

    /* CF is not set, so just add.  */
    $1 = add.#($1, sx.i8.#($2))
```

```
37
38      /* Branch to end.  */
39      branch L2
40
41  label L1:
42      /* CF is set, add and increment.  */
43      $1 = addi.#($1, sx.i8.#($2))
44
45  label L2:
46      /* Update flags.  */
47      r:flags = x86.uf(r:flags, (i16) 0x08D5)
48  end
49
50
51  /* Case 3.  */
52  match "ADC" pointer i64, const i32
53  begin
54      /* Test CF.  */
55      discard and.i16(r:flags, (i16) 0x0001)
56
57      /* Branch.  */
58      branch.nz L1
59
60      /* CF is not set, so just add.  */
61      $1 = add.i64($1, sx.i32.i64($2))
62
63      /* Branch to end.  */
64      branch L2
65
66  label L1:
67      /* CF is set, add and increment.  */
68      $1 = addi.i64($1, sx.i32.i64($2))
69
70  label L2:
71      /* Update flags.  */
72      r:flags = x86.uf(r:flags, (i16) 0x08D5)
73  end
74
75
76  /* Case 4.  */
77  match "ADC" pointer(r) i32, auto i32
78  begin
79      /* Test CF.  */
```

```
80      discard and.i16(r:flags, (i16) 0x0001)
81
82      /* Branch.  */
83      branch.nz L1
84
85      /* CF is not set, so just add.  */
86      local t1 = add.i32($1, $2)
87
88      /* Update flags.  */
89      r:flags = x86.uf(r:flags, (i16) 0x08D5)
90
91      @Store64GPR &1, local t1
92
93      /* Branch to end.  */
94      branch L2
95
96  label L1:
97      /* CF is set, add and increment.  */
98      local t2 = addi.i32($1, $2)
99
100     /* Update flags.  */
101     r:flags = x86.uf(r:flags, (i16) 0x08D5)
102
103     @Store64GPR &1, local t2
104
105 label L2:
106     nop
107 end
108
109
110 /* Case 5.  */
111 match "ADC" pointer(r) i32, const i8
112 begin
113     /* Test CF.  */
114     discard and.i16(r:flags, (i16) 0x0001)
115
116     /* Branch.  */
117     branch.nz L1
118
119     /* CF is not set, so just add.  */
120     local t1 = add.i32($1, sx.i8.i32($2))
121
122     /* Update flags.  */
```

```
123     r:flags = x86.uf(r:flags, (i16) 0x08D5)
124
125     @Store64GPR &1, local t1
126
127     /* Branch to end.  */
128     branch L2
129
130 label L1:
131     /* CF is set, add and increment.  */
132     local t2 = addi.i32($1, sx.i8.i32($2))
133
134     /* Update flags.  */
135     r:flags = x86.uf(r:flags, (i16) 0x08D5)
136
137     @Store64GPR &1, local t2
138
139 label L2:
140     nop
141 end
142
143
144 #   %1% — 16—bit r offset
145 #   %2% — 32—bit value
146 #macro Store64GPR
147     r[%1%] = zx.i32.i64(%2%)
148 #endm
```

Листинг А.2 Результат трансляции машинной команды ADC EDX, ESI в промежуточное представление Pivot

```
1  INIT   o.0:i16  = 0010h
2  INIT   o.1:i16  = 0030h
3  INIT   t.0:i16  = 0088h
4  LOAD   t.1:i16  = r[t.0]
5  INIT   t.2:i16  = 0001h
6  APPLY  t.3      = and.i16(t.1, t.2)
7  BRANCH 0Ch ? 0000h—0040h
8  LOAD   t.4:i32  = r[o.0]
9  LOAD   t.5:i32  = r[o.1]
10 APPLY  t.6      = add.i32(t.4, t.5)
11 INIT   t.7:i16  = 0088h
12 LOAD   t.8:i16  = r[t.7]
13 INIT   t.9:i16  = 08D5h
14 APPLY  t.10     = x86.uf(t.8, t.9)
```

```
STORE  r[t.7]   = t.10
APPLY  t.11     = zx.i32.i64(t.6)
STORE  r[o.0]   = t.11
BRANCH 0Bh
LOAD   t.12:i32 = r[o.0]
LOAD   t.13:i32 = r[o.1]
APPLY  t.14     = addi.i32(t.12, t.13)
INIT   t.15:i16 = 0088h
LOAD   t.16:i16 = r[t.15]
INIT   t.17:i16 = 08D5h
APPLY  t.18     = x86.uf(t.16, t.17)
STORE  r[t.15]  = t.18
APPLY  t.19     = zx.i32.i64(t.14)
STORE  r[o.0]   = t.19
NOP
```

Листинг А.3 Трасса Pivot-инструкций при выполнении команды ADC EDX, ESI

```
INIT   o.0:i16  = 0010h
INIT   o.1:i16  = 0030h
INIT   t.0:i16  = 0088h
LOAD   t.1:i16  = r[t.0]
INIT   t.2:i16  = 0001h
APPLY  t.3      = and.i16(t.1, t.2)
BRANCH 0Ch ? 0000h—0040h   ; Not executed
LOAD   t.4:i32  = r[o.0]
LOAD   t.5:i32  = r[o.1]
APPLY  t.6      = add.i32(t.4, t.5)
INIT   t.7:i16  = 0088h
LOAD   t.8:i16  = r[t.7]
INIT   t.9:i16  = 08D5h
APPLY  t.10     = x86.uf(t.8, t.9)
STORE  r[t.7]   = t.10
APPLY  t.11     = zx.i32.i64(t.6)
STORE  r[o.0]   = t.11
BRANCH 0Bh
NOP
```