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»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2017

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской академии наук.

Научный руководитель: кандидат физико-математических наук
**Падарян Вартан Андроникович**

Официальные оппоненты: **Галатенко Владимир Антонович,**
доктор физико-математических наук,
заведующий сектором Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

**Волконский Владимир Юрьевич,**
кандидат технических наук,
начальник отделения «Системы программирования» Публичного акционерного общества «Институт электронных управляющих машин им. И.С. Брука»

Ведущая организация: Федеральный исследовательский центр «Информатика и управление» Российской академии наук

Защита состоится 15 февраля 2018 г. в 16 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки «Институт системного программирования им. В.П. Иванникова Российской академии наук» по адресу: 109004, Москва, ул. А. Солженицына, 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки «Институт системного программирования им. В.П. Иванникова Российской академии наук».

Автореферат разослан «_____» _____ 201_ года.

Ученый секретарь
диссертационного совета
Д 002.087.01,
кандидат физико-математических наук Зеленов С.В.

## Общая характеристика работы

**Актуальность темы.** 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++, в которых отсутствует автоматическая проверка границ массивов и которые, как следствие, не обеспечивают безопасность доступа к памяти. Использование небезопасных языков 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. 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, 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, при котором длины буферов в программе представляются в виде свободных переменных и обрабатываются наравне с другими символьными значениями. 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.

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 (стековой) 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 диссертации.** Диссертация состоит из введения, четырёх глав, заключения и одного приложения. Полный объем диссертации составляет 92 страницы, включая 5 рисунков и 2 таблицы. Список литературы содержит 55 наименований.

## Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, формулируется цель, ставятся задачи работы, формулируется научная новизна и практическая значимость представляемой работы.

В **главе 1** описываются существующие 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.

В разделе 1.1 описываются подходы к поиску ошибок с помощью статического анализа кода. В рамках данного вида 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.

В разделе 1.2 описываются подходы к поиску ошибок с помощью инструментации анализируемого кода. Инструментация позволяет выполнять дополнительный код в различных точках программы, таких как операции обращения к памяти, границы базовых блоков, вызовы и возвраты из функций. Внедрение дополнительного кода позволяет выполнять различные проверки на ошибочные

ситуации во время работы программы и, таким образом, находить различного рода ошибки. 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.

Подходы к поиску ошибок, основанные на динамическом символьном выполнении, описаны в разделе 1.3. 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.

В разделе 1.4 приводится сравнение рассмотренных подходов к поиску ошибок выхода за границы буфера. В результате анализа этих подходов делается вывод, что для эффективного поиска ошибок выхода за границы буфера целесообразно использовать методы, позволяющие анализировать бинарный код программ с помощью символьного выполнения.

В **главе 2** даётся формальная постановка задачи поиска ошибок выхода за границы буфера по трассам выполнения программ. Описывается предлагаемый подход к решению этой задачи.

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 ей 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 1). Сначала аналитиком создаются наборы входных данных для одного или нескольких запусков исследуемой программы. Эти наборы выходных данных могут быть

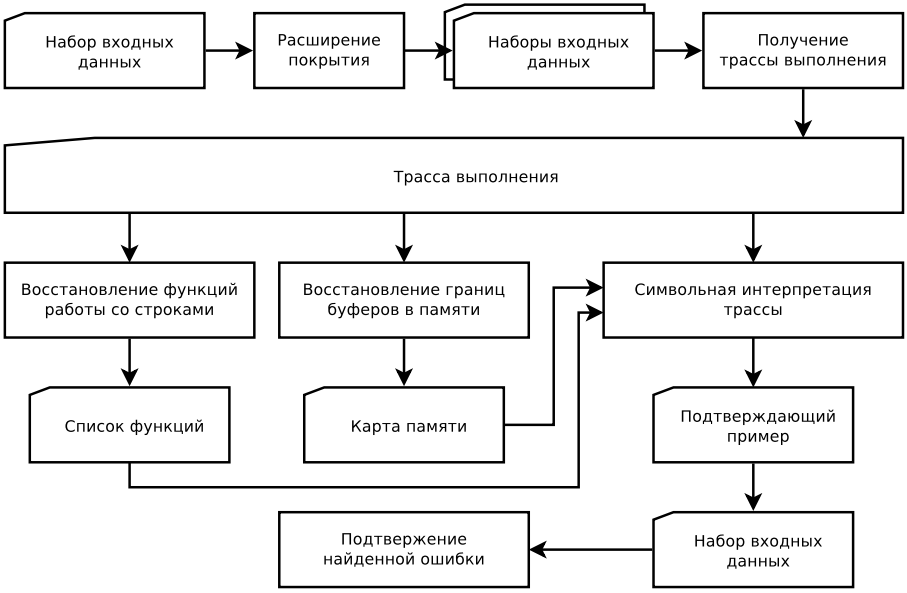


Рис. 1 — Блок-схема метода.

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 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, 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.

В разделе 2.1 описывается метод символьной интерпретации длин буферов. Вводится понятие $L$-буфера, а также определяется контекст интерпретации. $L$-буфером называется буфер в памяти, имеющий символьную длину. Такой буфер задаётся тройкой $l = [\![base, clen, slen]\!]$, содержащей адрес базы ($base$), реальную ($clen$) и символьную ($slen$) длину, отвечающую за абстрактный размер буфера. Символьный буфер схематично изображён на рисунке 2.

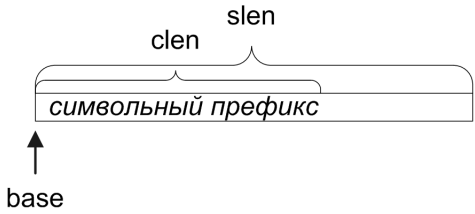


Рис. 2 — Структура символьного буфера.

$L$-буфер может относиться к обрабатываемым данным либо определять выделенный буфер в памяти. В зависимости от его типа, $L$-буфер может находиться в одном из двух множеств $A$ или $I$, которые определяют множество выделенных буферов и множество буферов с обрабатываемыми данными, соответственно.

Для описания символьного состояния вводится множество символьных переменных $\mathfrak{S}$, а также предикат пути $Pp$, представляющий собой набор уравнений, описывающий прохождение программы по определённому пути. Отображение $\Delta$ связывает ячейки (машинные слова) памяти и выражения над символьными переменными. Получение символьного и конкретного численного значения ячейки памяти обозначается как $\Delta[m]$ и $M[m]$ соответственно.

Контекст интерпретации представляется кортежем $\{\Delta, Pp, A, I, \mathfrak{S}\}$. Отображение $M$ не входит в контекст интерпретации, так как операция $M[m]$ вычисляет конкретные значения ячеек памяти, которые не зависят от состояния интерпретации.

Длина буфера связывается с буфером с помощью объявления соответствующего $L$-буфера в одном из множеств. Интерпретация заключается в модификации контекста интерпретации в соответствии с правилами интерпретации. Правила можно разбить на две группы:

1. Интерпретация вызовов функций с известной семантикой.
2. Интерпретация отдельных машинных команд.

Функции с известной семантикой интерпретируются как единое целое, оставшиеся инструкции интерпретируются согласно своему набору правил. Сами буферы и их символьная длина определяются во время интерпретации функций получения входных данных, а также функций выделения памяти.

Правила интерпретации для функций описаны в разделе 2.1.2. 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 копируются 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. 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.

Правила интерпретации для отдельных инструкций описаны в разделе 2.1.3. Соверменные процессорные архитектуры довольно сложны и содержат машинные команды с нетривиальными побочными эффектами. Описание правил анализа для каждой машинной команды нецелесообразно, для этого традиционно используется тот или иной вариант промежуточного представления, позволяющий выразить операционную семантику машинных инструкций в терминах более простых операций некоторой виртуальной машины. Такая виртуальная машина обычно представлят из себя RISC-машину с ограниченным

набором команд. Для анализа бинарного кода требуется описать лишь правила анализа для инструкций этой виртуальной машины.

Инструкцию виртуальной машины можно представить в виде тройки, описывающей операцию над данными и ее фактические операнды: $\langle OP, \{use_i\}, \{def_j\}\rangle$, где $OP$ – код операции, $use$, $def$ – множества ячеек, считываемых и записываемых данной инструкцией. В наборах операндов явно указываются ячейки памяти, которые считываются и записываются при выполнении команды. В данной работе используется виртуальная машина со следующими типами инструкций:

- выполнение унарной операции $\langle \diamondsuit_u, \{use\_cell\}, \{def\_cell\}\rangle$,
- выполнение бинарной операции $\langle \diamondsuit_b, \{use\_cell_1, use\_cell_2\}, \{def\_cell\}\rangle$,
- 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 $\langle jct, \{cond\_cell, [addr\_cell]\}, \{\}\rangle$, $\langle jcf, \{cond\_cell\}, \{\}\rangle$.

Для удобства интерпретации явно разделены сработавшая ($jct$) и несработавшая ($jcf$) инструкция условной передачи управления, поскольку при их обработке к предикату пути добавляются различные ограничения.

Обращение к регистрам процессора моделируется так же, как и обращение к ячейкам памяти: с помощью инструкций загрузки и выгрузки. Для разделения памяти и регистров вводится понятие адресного пространства. При этом инструкции виртуальной машины оперируют над значениями, размещёнными на виртуальных регистрах, количество которых неограниченно.

Инструкции, выполняющие унарные и бинарные операции над данными транслируются в соответствующие им операции над символьными значениями. Операции загрузки и выгрузки обновляют отображение $\Delta$ 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 $cond\_cell$.

В разделе 2.2 описывается метод восстановления границ буферов по трассе выполнения. Области памяти, используемые в программе, можно разделить на три класса: статическая, динамическая и автоматическая (стековая) 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 намного 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 них 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.

В разделе 2.3 описывается метод восстановления функций работы со строками, 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 их параметры. 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.

В разделе 2.4 описывается метод, позволяющий улучшить результаты работы алгоритмов анализа за счёт увеличения покрытия кода анализируемой программы. Такое увеличение покрытия достигается за счёт анализа трассы, содержащей несколько запусков анализируемой программы на различных наборах входных данных. За основу для увеличения покрытия берётся один запуск программы, для которого известен набор входных данных.

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 символьные 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. Это возможно сделать с помощью классической задачи о покрытии множества. Данная задача 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.

**Алгоритм 1** Минимизация множества состояний

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

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 выполняться 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.

В **главе 3** приводится описание программной реализации методов, предложенных в главе 2. В качестве платформы для реализации методов выбрана среда анализа бинарного кода, разрабатываемая в ИСП РАН. Среда имеет модульную расширяемую архитектуру и предназначена для решения широкого класса задач

обратной инженерии. Архитектура реализованной системы приведена на рисунке 3.

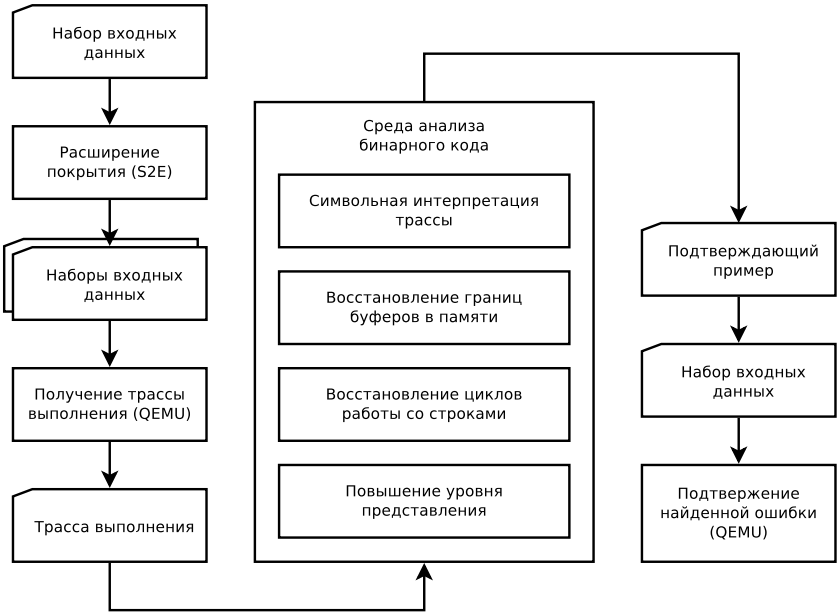


Рис. 3 — Архитектура системы.

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

В разделе 3.1 описывается процесс подготовки трассы к анализу. Подготовка заключается в повышении уровня представления: выполняется предварительная обработка, в ходе которой восстанавливаются высокоуровневые сущности в трассе, такие как процессы и потоки ОС, карта исполняемых модулей, вызовы функций, циклы и граф потока управления.

В разделе 3.2 описывается подсистема интерпретации трассы, поддерживающая символьную длину буферов.

Перед началом анализа определяются источники входных данных, представляющие собой пару (шаг трассы, буфер в памяти). Источниками входных данных могут быть как вызовы функций получения данных (такие как `read`, `recv`), так и буферы в памяти, заданные аналитиком. Источником входных данных может быть также функция `main` программы, в которую передаются параметры командной строки. Таких низкоуровневых источников входных данных достаточно для описания получения данных из файлов, сокетов и командной строки.

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.

Отслеживание потоков данных происходит на уровне физических адресов памяти, что позволяет анализировать, в частности, межпроцессное взаимодействие программ.

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.

Каждая инструкция, отобранная с помощью подсистемы анализа потоков данных, 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. Это форма представления, в которой каждой переменной значение присваивается лишь единожды, что значительно упрощает проведение различных видов 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. Символьная интерпретация одновременно всех ветвей значительно усложняет систему уравнений, поэтому для интерпретации используется только то подмножество Pivot-инструкций, которое было реально задействовано во время выполнения анализируемой инструкции на процессоре. Для получения такого подмножества операций выполняется интерпретация Pivot-кода: множество Pivot-инструкций транслируется в машинный код архитектуры x86_64 и затем выполняется.

Полученное множество Pivot-инструкций анализируется и для каждой инструкции генерируются уравнения, отражающие её выполнение. 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.

Помимо трансляции отдельных инструкций, выполняется трансляция вызовов функций из заранее определённого списка: интерпретация инструкций

внутри вызова заменяется интерпретацией правила для соответствующей функции. Добавление уравнений и выполнение проверок происходит на точках вызова и возврата из функции.

В разделе 3.3 описывается подсистема восстановления границ буферов в памяти. В рамках данной подсистемы реализованы два алгоритма: для восстановления границ буферов в динамической и автоматической памяти.

Алгоритм, восстанавливающий границы буферов в динамической памяти, получает на вход список вызовов функций некоторого менеджера кучи и с помощью однократного прохода по списку вызовов в трассе обновляет список выделенных буферов в памяти в соответствии с параметрами вызываемых функций.

Для восстановления границ буферов в автоматической памяти используется информация из статического представления программных модулей. С помощью статического анализатора IDA Pro 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. Время жизни стекового фрейма определяется границами вызова функции.

В разделе 3.4 описывается 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 его границы 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.

В разделе 3.5 описывается 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 в результате минимизации 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.

В разделе 3.6 приводится перечень технических ограничений текущей реализации системы. Обсуждаются способы преодоления некоторых из этих ограничений.

В **главе 4** описываются результаты применения описанных методов на различных примерах. В качестве объектов анализа выступали приложения, работающие под управлением операционных систем семейства Windows и Linux, а также встроенное программное обеспечение для сетевого маршрутизатора. Детально рассмотрены различные ситуации в программах, для которых удалось обнаружить ошибку выхода за границы буфера.

На примере приложений `httpdx` и `GoldMp4Player` (раздел 4.1) демонстрируется обнаружение ошибки переполнения буфера, расположенного на куче. Для обнаружения выхода за границы буферов на куче используется 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.

На примере приложения `httpdx` (раздел 4.2) также демонстрируется особый случай ошибки переполнения буфера: когда переполнение происходит не из-за большого размера записываемых данных, а из-за недостаточного размера выделенной памяти. Этот размер вычисляется на основе значения поля Content-Length в обрабатываемом HTTP-запросе, которое является частью входных данных. Несмотря на тот факт, что значение поля 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.

В разделе 4.3 демонстрируется обнаружение известной уязвимости Heartbleed в библиотеке `OpenSSL`. В коде этой библиотеки, отвечающем за обработку TLS-пакета Heartbeat, отсутствует контроль размеров копируемых данных, что приводит к выходу за границы буфера при чтении и дальнейшей утечке чувствительных данных в сеть. 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.

В разделе 4.4 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. 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.

В **заключении** перечисляются основные результаты работы, а также излагаются планы развития и применения разработанных методов поиска ошибок.

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.

## Основные результаты диссертационной работы

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 (стековой) памяти.

## Публикации автора по теме диссертации


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.

*Каушан Вадим Владимирович*

Поиск ошибок выхода за границы буфера в бинарном коде программ

Автореф. дис. на соискание ученой степени канд. тех. наук

Подписано в печать ___.___._____. Заказ № _____
Формат 60×90/16. Усл. печ. л. 1. Тираж 100 экз.
Типография _____