Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К магистерской диссертации

«Разработка алгоритмов статического поиска выходов за пределы динамического массива в C/C++ программах»

Автор: Громаковский Иван Евгеньевич _____
Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика
Квалификация: Магистр
Руководитель: Лукин М.А., канд. техн. наук _____

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. _____

«___» _____ 20___ г.

Санкт-Петербург, 2017 г.

Студент Громаковский И.Е. **Группа** М4239 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Технологии проектирования и разработки программного обеспечения

Квалификационная работа выполнена с оценкой _____

Дата защиты «___» _____ 20___ г.

Секретарь ГЭК *Павлова О.Н.* Принято: «___» _____ 20___ г.

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

УТВЕРЖДАЮ
Зав. каф. компьютерных технологий
докт. техн. наук, проф.
_____ Васильев В.Н.
«__» _____ 20__ г.

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

Студент Громаковский И.Е. **Группа** М4239 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования **Руководитель** Лукин М.А., канд. техн. наук, тьютор кафедры КТ Университета ИТМО

1 Наименование темы: Разработка алгоритмов статического поиска выходов за пределы динамического массива в C/C++ программах

Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика

Направленность (профиль): Технологии проектирования и разработки программного обеспечения

Квалификация: Магистр

2 Срок сдачи студентом законченной работы: «31» мая 2017 г.

3 Техническое задание и исходные данные к работе.

Требуется разработать и реализовать алгоритм статического поиска выходов за пределы динамического массива в C/C++ программах. Необходимо обрабатывать промышленные программы больших размеров (состоящие из сотен тысяч строк) за разумное время (порядка нескольких десятков минут). Анализатор должен находить как можно больше ошибок с как можно меньшим числом ложных срабатываний. На сегодняшний день известно множество свободно распространяемых анализаторов, решающих аналогичную или более общую проблему. В данной работе необходимо также произвести сравнение с другими анализаторами.

4 Содержание магистерской диссертации (перечень подлежащих разработке вопросов)

а) Обоснование актуальности задачи, описание предметной области, обзор существующих решений.
б) Теоретическое исследование задачи поиска выходов за пределы динамического массива в C/C++. Разработка алгоритма.
в) Описание практической реализации алгоритма, результаты экспериментов, сравнение с другими анализаторами.

5 Перечень графического материала (с указанием обязательного материала)

Не предусмотрено

6 Исходные материалы и пособия

а) Li, Lian, Cristina Cifuentes, and Nathan Keynes. "Practical and effective symbolic analysis for buffer overflow detection."Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010.

б) Shahriar, Hossain, and Mohammad Zulkernine. "Classification of static analysis-based buffer overflow detectors."Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on. IEEE, 2010.

в) Le, Wei, and Mary Lou Soffa. "Marple: a demand-driven path-sensitive buffer overflow detector."Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 2008.

г) Ding, Sun, et al. "Detection of buffer overflow vulnerabilities in C/C++ with pattern based limited symbolic evaluation."Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual. IEEE, 2012.

7 Календарный план

№№ пп.	Наименование этапов магистерской диссертации	Срок выполнения этапов работы	Отметка о выполнении, подпись руков.
1	Ознакомление с основами статического анализа	до 15.11.2015	
2	Изучение существующих подходов к решению данной проблемы	до 28.02.2016	
3	Разработка и реализация прототипа анализатора	до 30.06.2016	
4	Исследование преимуществ и недостатков прототипа, разработка теоретических улучшений	до 20.11.2016	
5	Реализация улучшенной версии анализатора	до 25.01.2017	
6	Проведение экспериментов, сравнение с другими анализаторами	до 26.02.2017	
7	Написание пояснительной записки	до 31.05.2017	

8 Дата выдачи задания: «01» сентября 2015 г.

Руководитель _____

Задание принял к исполнению _____ «01» сентября 2015 г.

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

АННОТАЦИЯ
МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

Студент: Громаковский Иван Евгеньевич

Наименование темы работы: Разработка алгоритмов статического поиска выходов за пределы динамического массива в C/C++ программах

Наименование организации, где выполнена работа: Университет ИТМО

ХАРАКТЕРИСТИКА МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

1 Цель исследования: Разработка алгоритмов статического анализа C/C++ программ для поиска выходов за пределы динамического массива.

2 Задачи, решаемые в работе:

а) способность находить как можно больше ошибок, связанных с выходами за пределы динамического массива, с как можно меньшим числом ложных срабатываний;

б) обработка крупных промышленных приложений, состоящих из сотен тысяч строк, за разумное время;

в) проведение экспериментов, сравнение с другими анализаторами.

3 Число источников, использованных при составлении обзора: 27

4 Полное число источников, использованных в работе: 34

5 В том числе источников по годам

Отечественных			Иностранных		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
0	0	0	12	4	18

6 Использование информационных ресурсов Internet: 8

7 Использование современных пакетов компьютерных программ и технологий: Анализатор был реализован на языке C++ с использованием системы сборки CMake. Использовалась инфраструктура LLVM. Также использовались библиотеки Boost и STL. Для анализа больших программ использовалась утилита whole-program-llvm. Также были написаны Bash-скрипты для упрощения запуска приложения и автоматизации. Использовалась система контроля версий Git. Для написания пояснительной записки использовался LaTeX.

8 Краткая характеристика полученных результатов: В результате получился анализатор, способный находить ошибки в больших программах за разумное время и превосходящий существующие анализаторы, такие как PVS-Studio, в некоторых случаях (т. к. сравнение в общем случае невозможно).

9 Гранты, полученные при выполнении работы: Грантов или других форм государственной поддержи и субсидирования в процессе работы не предусматривалось.

10 Наличие публикаций и выступлений на конференциях по теме работы: По теме работы имеется выступление на конференции «Конгресс молодых ученых». Также имеются публикации:

1 *Громаковский И.*, *Лукин М.* Разработка алгоритмов статического поиска выходов за пределы динамического массива в C/C++ программах // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. — 2017.

2 *Громаковский И.*, *Анохина И.* Development of buffer overflow detection algorithms for C/C++ programs // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. — 2017.

Выпусник: Громаковский И.Е. _____

Руководитель: Лукин М.А. _____

«__» _____ 20__ г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 6
1. Поиск выходов за пределы динамического массива 8
1.1. Описание проблемы 8
1.2. Обзор существующих решений 10
1.2.1. Обзор теоретических решений 10
1.2.2. Обзор практических реализаций 13
1.3. Обзор статьи «Practical and effective symbolic analysis for buffer overflow detection» 14
1.3.1. Static Single Assignment 14
1.3.2. Символьные вычисления 15
1.3.3. Вычисление диапазонов 16
Выводы по главе 1 17
2. Разработанные улучшения 18
2.1. Обработка циклов 18
2.1.1. Монотонно изменяющиеся переменные 18
2.1.2. Обработка предиката «не равно» 19
2.2. Учёт предикатов 21
2.3. Межпроцедурный анализ 23
2.3.1. Триггеры 23
2.3.2. Построение триггеров 25
2.3.3. Проверка триггеров 26
Выводы по главе 2 26
3. Практическая реализация и результаты 28
3.1. Реализация 28
3.1.1. LLVM 28
3.1.2. Gated Single Assignment 30
3.1.3. Символьные вычисления 30
3.1.4. Обработка больших программ 32
3.2. Экспериментальные результаты 32
3.2.1. Результаты использования анализатора 32
3.2.2. Сравнение с другими анализаторами 34
Выводы по главе 3 38

ЗАКЛЮЧЕНИЕ. 39
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 40
ПРИЛОЖЕНИЕ А. Тесты для сравнения 43

ВВЕДЕНИЕ

Программное обеспечение всегда было и остаётся подвержено ошибкам. Одними из наиболее уязвимых являются программы, написанные на языках C и C++, поскольку они предоставляют прямой доступ к памяти и не имеют встроенных механизмов для предотвращения некорректного обращения к ней. Пожалуй, наибольшую опасность представляют ошибки, при которых происходит доступ к памяти (чтение или запись) за пределами выделенного буфера. Такие ошибки могут приводить как к остановке программы, так и к более серьёзным, с точки зрения безопасности, проблемам. Например, злонамеренный пользователь может получить доступ к приватной информации, исполнить произвольный код, а в худшем случае получить права суперпользователя [1].

Существует большое число подходов к решению данной проблемы, как статических [2–6], так и динамических [7, 8]. Динамические подходы обнаруживают ошибки во время работы программы и имеют несколько серьёзных недостатков по сравнению со статическими подходами, когда программа анализируется до запуска. Во-первых, дополнительные проверки, выполняемые во время работы программы, занимают какое-то время и тем самым ухудшают производительность. Во-вторых, даже если ошибка будет обнаружена, программа в лучшем случае просто остановится. В-третьих, зачастую оказывается довольно сложно быстро внести изменения, направленные на исправление ошибок, и донести их до конечного пользователя. Поэтому гораздо предпочтительнее обнаружить ошибки до поставки программы пользователю.

Основная цель данной работы состоит в разработке алгоритмов статического анализа C/C++ программ на предмет выхода за пределы динамического массива. При этом необходимо обрабатывать большие программы, состоящие из сотен тысяч строк, достаточно быстро, чтобы анализатором можно было пользоваться на практике. В общем случае задача поиска всех выходов за пределы массива без единого ложного срабатывания неразрешима, поэтому анализатор должен находить как можно больше реальных ошибок, при этом выдавая как можно меньше ложных срабатываний.

За основу данной работы был взят подход, описанный в статье [6]. Были выявлены преимущества и недостатки подхода, предложены и реализованы способы решения найденных недостатков. С помощью проделанных улучшений удалось существенно уменьшить число ложных срабатываний, тем самым

увеличив точность анализа. Также было проведено сравнение с другими свободно распространяемыми анализаторами, показано превосходство над ними. Реализованный алгоритм способен обрабатывать программы, состоящие из сотен тысяч строк, за время порядка десяти минут.

В первой главе приведён обзор предметной области и существующих решений описанной проблемы. Во второй главе представлены результаты исследования алгоритма из статьи [6], описан теоретический подход к устранению выявленных недостатков. В третьей главе рассмотрены вопросы практической реализации предложенного решения, приведены основные результаты работы и сравнение с другими анализаторами.

ГЛАВА 1. ПОИСК ВЫХОДОВ ЗА ПРЕДЕЛЫ ДИНАМИЧЕСКОГО МАССИВА

1.1. Описание проблемы

Ошибки в коде программного обеспечения могут представлять большую опасность и приводить с серьёзным убыткам. Например, в 2003-ем году большое число компьютеров по всему миру было заражено вирусом SQL Slammer [9], что приводило к отказу оборудования и существенному замедлению трафика в сети Интернет в целом. Другим примером служит червь Code-Red, убытки от которого оцениваются миллиардами долларов [10]. Оба вируса использовали уязвимости в программном коде, позволявшие осуществить выход за пределы массива. Такие ошибки являются одними из наиболее опасных, поскольку зачастую злоумышленик может, используя уязвимость, выполнить произвольный код и/или получить права суперпользователя [1].

Наиболее актуальна данная проблема для языков C/C++. Дизайн этих языков подразумевает высокую гибкость и производительность, жертвуя безопасностью. Языки позволяют осуществлять произвольный доступ к памяти и не имеют встроенных автоматических проверок корректности. Для многих приложений данный недостаток является менее существенным, чем производительность, поэтому языки C и C++ активно используются и по сей день. Также порой выбор языка происходит по историческим соображениям, когда приходится поддерживать старый код. Многие операционные системы написаны на C/C++ из соображений гибкости и производительности. Также существует большое число программ, работающих с сетью и получающих данные извне, написанных на C/C++ из соображений производительности. Уязвимости в таком программном обеспечении являются крайне опасными, поскольку операционные системы работают напрямую с аппаратными средствами, а ошибки в сетевых приложениях могут быть эксплуатированы удалённо фактически любым человеком.

В данной работе под выходом за пределы динамического массива подразумевается обращение (чтение и запись) к памяти за пределами выделенного участка памяти. Если существуют такие входные данные (то есть любые данные, неизвестные заранее), что при запуске программы с этими данными в ней присутствует инструкция, совершающая выход за пределы массива, то такая

инструкция считается ошибочной и потенциально уязвимой. Задача состоит в поиске таких инструкций.

В связи с актуальностью проблемы ей уделено большое внимание учёных и инженеров по всему миру. Известно большое число подходов к решению данной задачи. Глобально их можно разделить на динамические и статические.

Динамические подходы [7, 8, 11] состоят в добавлении в программу дополнительных проверок, предотвращающих обращение к памяти за пределами выделенного буфера. Основное преимущество этого подхода в том, что он, как правило, предотвращает большее число ошибок, поскольку значения всех переменных известны в момент обращения к памяти. Однако такие подходы существенно замедляют работу программы, что зачастую оказывается неприемлемо в случаях, когда производительность стоит на первом месте. Другим недостатком динамического подхода является тот факт, что при наличии ошибки она будет обнаружена уже во время исполнения программы, что, скорее всего, приведёт к остановке. Также динамические подходы способны находить лишь ошибки, которые воспроизводятся на реально выполняемых участках кода. На практике же нередко происходит так, что какой-то фрагмент кода может не выполняться на протяжении очень длительного времени, однако именно в нём может быть ошибка, которая очень долго будет оставаться незамеченной.

Статический анализ, в отличие от динамического, производится без запуска программы. Это позволяет исследовать все места в коде, даже редко выполняемые, а также позволяет найти ошибки до реального использования программы, пока они не проявили себя. Помимо этого статический анализ никак не меняет исполняемый код, а значит, не замедляет программу. В общем случае задача статического поиска всех выходов за пределы массива в C/C++ без единого ложного срабатывания неразрешима (это напрямую следует из проблемы останова [12]), поэтому одним из главных недостатков статического анализа является необходимость ручного рассмотрения результатов работы анализатора с целью выяснения, какие из найденных ошибок действительно являются таковыми.

Таким образом, динамические и статические подходы имеют свои преимущества и недостатки. В разных случаях имеет смысл применять разные подходы (в том числе комбинацию подходов). Оба варианта являются осмысленными и актуальными. В данной работе был сделан выбор в пользу ста-

тического анализа. Поскольку задача неразрешима в общем случае, требуется находить как можно больше ошибок в программах, минимизируя при этом число ложных срабатываний. Важным требованием является способность обрабатывать крупные программы, состоящие из сотен тысяч строк, за разумное время. Как правило, приемлемым является время порядка нескольких десятков минут. Это сопоставимо со временем сборки проекта и позволяет, например, включить анализатор в систему непрерывной интеграции.

Несмотря на огромное число исследований, посвящённых данной проблеме, уязвимости, связанные с выходом за пределы массива продолжают оставаться одними из наиболее часто встречаемых [13]. Таким образом, на сегодняшний день задача не перестаёт быть актуальной. Также стоит отметить, что большое число подходов, описанных в статьях, находятся в закрытом доступе или же поддерживаются на протяжении долгих лет. Большинство доступных анализаторов решают более общую проблему поиска различных ошибок и либо работают слишком долго на больших программах, либо находят слишком мало ошибок, связанных с выходами за пределы массива.

1.2. Обзор существующих решений

Существует большое число статей, посвящённых статическому поиску выходов за пределы динамического массива в C и C++. Отдельно стоит отметить статью [14], в которой представлена классификация различных подходов. В большинстве случаев основной компромисс делается между скоростью работы, числом находимых ошибок и точностью. К сожалению, на сегодняшний день ситуация такова, что большое число существующих решений либо только описаны в литературе, но не имеют открытой актуальной реализации, либо реализованы, но нигде не описаны. В связи с этим целесообразно разбить данный раздел на две части: обзор статей, посвящённых решаемой проблеме, и обзор реализованных анализаторов, работающих на практике.

1.2.1. Обзор теоретических решений

Наиболее простые способы решения проблемы [15, 16] не учитывают поток управления и зависимости по данным в программе, а выполняют лишь символьный анализ. Такие подходы просты в реализации и работают очень быстро, однако точность анализа крайне низкая. По сути, такие анализаторы лишь находят потенциально опасные места в программе и эвристически сор-

тируют их. В данной работе требуется большая точность, поэтому необходимо учитывать семантику программы.

Некоторые подходы [17–19] основаны на использовании пользовательских аннотаций для вывода ограничений на значения переменных. В таком случае разработчик пишет аннотации в различных местах в коде. Аннотации сообщают анализатору различные инварианты, гарантированно выполняющиеся в определённом месте. Например, что значение переменной ограничено сверху определённым значением. Анализатор работает в предположении, что все инварианты, задаваемые аннотациями, выполнены, что ползволяет повысить точность анализа. Некоторые анализаторы также проверяют корректность аннотаций. Недостатком таких подходов является необходимость добавления аннотаций в код. В некоторых случаях это не представляет большой проблемы, особенно если приложение пишется с нуля. Однако для больших программ добавление аннотаций к каждой функции может занять чрезвычайно много времени. Наибольшую трудность представляет аннотирование старого кода, написанного незнакомыми людьми. Поскольку основной целью работы является анализ крупных программ, использование аннотаций не имеет большого смысла, и в этой работе они не используются.

Также для решения рассматриваемой задачи можно использовать подходы, базирующиеся на ограниченной проверке моделей (bounded model checking) [20, 21]. Такой подход позволяет проверять различные свойства программы, включая наличие выходов за пределы массива. Для этого программа описывается системой переходов, проверяемое свойство формулируется в виде логического утверждения, после чего выполнимость условия проверяется SAT- или SMT-решателем. При этом рассматриваются пути в программе с ограниченной глубиной, чтобы гарантировать, что анализ закончится. Такие подходы обладают хорошей точностью и позволяют проверять различные свойства программы, однако это достигается за счёт низкой производительности. Проблема заключается в экспоненциальном росте сложности задачи при увеличении числа ветвлений в программе. В результате такие подходы не могут быть применены к большим программам.

Подходы, описанные в статьях [2, 4, 17], в целом весьма похожи. Сначала анализатор обходит весь код программы и строит систему ограничений на значения переменных. Затем система решается с использованием различ-

ных алгоритмов (зависит от анализатора). В результате выводится как можно более точная оценка возможных значений переменных. После этого обращения к массиву анализируются, исходя из выведенных ограничений. Подходы различаются в основном чувствительностью анализа (учётом контекста, наличием межпроцедурного анализа и т. д.), гранулярностью анализа и способом разрешения ограничений. За счёт различных эвристик такие подходы работают относительно быстро на программах среднего размера, однако на больших программах получается слишком много ограничений, и их решение занимает слишком много времени.

В статье [3] описан подход, способный обрабатывать программы, состоящие из сотен тысяч строк кода. Этот подход использует полностью символьный анализ, в рамках которого возможные значения переменных представлены символьно, что позволяет учитывать зависимости между переменными, не зная конкретных значений. Анализ является межпроцедурным, что существенно повышает точность на практике. При анализе конкретной функции происходит проход по графу потока управления всей функции. Каждая инструкция может модифицировать контекст анализатора. При обработке условных переходов выполняется проверка значения, влияющего на переход. Если значение не может быть посчитано статически, происходит размножение контекста, и анализатор продолжает работу с большим числом контекстов. При обработке обращения к массиву анализатор может непосредственно вывести, возможно ли переполнение. За счёт отсутствия необходимости разрешать системы ограничений такой подход работает быстрее предыдущих.

Более современная идея заключается в использовании анализа «по требованию», при котором анализируются только те инструкции программы, которые влияют на результат какого-то запроса. Например, статья [5] использует анализ по требованию для поиска выходов за пределы массива. Основная цель данной идеи заключается в уменьшении числа анализируемых путей программы и, следовательно, повышении скорости анализа. «Требование» анализа возникает в местах, где может выполниться какой-то запрос (в данном случае в местах обращения к массиву). Затем происходит распространение запроса в обратном направлении по графу потока управления. В процессе распространения происходит сбор информации о значениях переменных, влияющих на запрос. Обход заканчивается, когда ответ на запрос может быть однозначно

вычислен или при достижении входа в функцию. На практике такой подход работает существенно быстрее аналогичных подходов, анализирующих программу целиком. Схожие идеи применены также в статьях [6, 22, 23].

1.2.2. Обзор практических реализаций

Одним из наиболее популярных средств статического анализа, используемых в промышленном программировании, является PVS-Studio [24]. Данный анализатор активно развивается и используется для поиска ошибок в большом числе программ на C/C++, в том числе в крупных проектах. Число различных видов ошибок и подозрительных мест, обнаруживаемых анализатором, на сегодняшний день близко к тысяче. Одним из видов ошибок являются выходы за границу массива. PVS-Studio — это закрытый проект, и используемые алгоритмы не описаны в публичном доступе, однако для open-source проектов данный анализатор может быть использован бесплатно. Стоит отметить, что для проведения анализа необходимо предварительно собрать проект.

Ещё одним популярным анализатором является Cppcheck [25]. Он также имеет большое число встроенных проверок, однако меньшее, чем PVS-Studio. Также поддерживаются языки C и C++, при этом не требуется компилировать исходный код, так как анализ проводится только на уровне исходного кода. Как и PVS-Studio, анализатор обладает хорошей скоростью работы и способен обрабатывать большие программы достаточно быстро. В отличие от PVS-Studio, исходный код анализатора находится в публичном доступе.

Анализатор Splint [26] является реализацией подхода, описанного в статье [17]. Данный анализатор также является свободно распространяемым. Большим недостатком анализатора является отсутствие поддержки C++. Активная разработка анализатора была прекращена много лет назад, в связи с чем он зачастую неспособен обрабатывать современный код.

Также стоит упомянуть средство статического анализа кода LLBMC [21], выполняющее ограниченную проверку моделей. Исходный код анализатора находится в закрытом доступе, однако анализатор бесплатно доступен в бинарном виде для некоммерческого использования. Минусом используемого подхода является плохая масштабируемость при увеличении размера кода.

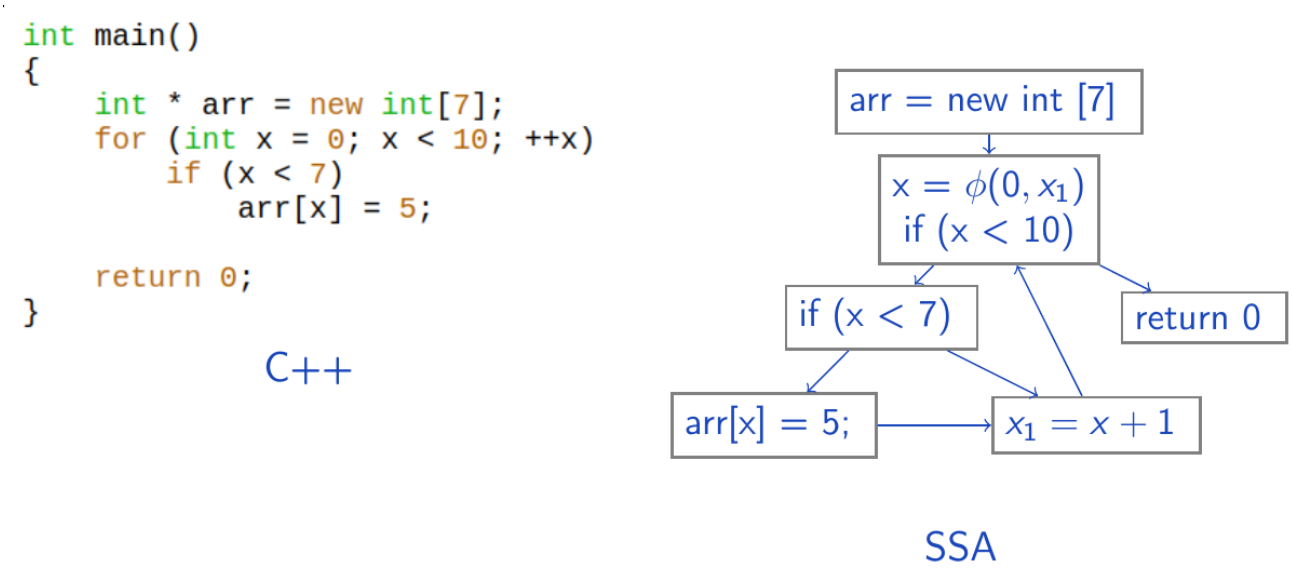

Рисунок 1 – SSA форма

1.3. Обзор статьи «Practical and effective symbolic analysis for buffer overflow detection»

На основании проведённого изучения литературы было принято решение использовать алгоритм, описанный в статье [6], в качестве базового. Во-первых, этот подход является легковесным, что позволяет достаточно быстро обрабатывать большие программы. Во-вторых, результаты, представленные в статье, говорят о том, что анализатор способен находить много ошибок в крупных открытых проектах. В-третьих, несмотря на довольно хорошие результаты, авторы также отмечают наличие ложных срабатываний и спорных ситуаций, в которых алгоритм не может принять однозначное решение. А значит, у этого подхода есть пространство для улучшений.

1.3.1. Static Single Assignment

Описанный в статье алгоритм рассматривает программу, представленную в Static Single Assignment форме [27] (SSA). Такое представление является весьма удобным для статического анализа. SSA форма имеет две особенности. Во-первых, значение любой переменной присваивается ровно один раз. Во-вторых, вводится специальная инструкция ϕ, возвращающая один из своих аргументов в зависимости от того, из какого предка поток управления пришёл в эту инструкцию. За счёт этого удаётся представить различные языковые конструкции, при которых происходит изменение значения переменной, например, циклы.

На рисунке 1 представлен пример SSA формы. В C++ программе значение переменной x меняется на каждой итерации цикла. Однако в SSA форме значение переменной может присваиваться только один раз. Для этого значение x определяется как ϕ от начального значения (0) и дополнительной переменной x_1, определённой как $x+1$. Таким образом, на каждой итерации цикла значение переменной x будет на 1 больше предыдущего, что соответствует C++ коду.

1.3.2. Символьные вычисления

Алгоритм базируется на символьных вычислениях для обнаружения выходов за пределы массива. Символьные вычисления позволяют учесть зависимости между переменными, не зная их численных значений. В основе алгоритма лежит понятие символьного выражения (symbolic expression), а также символьного диапазона или отрезка (symbolic range).

Символьным выражением может быть число, переменная, афинная функция другого символьного выражения, а также два специальных значения $\bot$ и $\top$, соответствующих наименьшему и наибольшему возможному значению. Для символьных выражений естественным образом определяются операции сложения, вычитания, умножения и деления. Также вводится частичный порядок ($\prec$) следующим образом: $E_1 \prec E_2$ тогда и только тогда, когда $E_1 - E_2$ не больше нуля (для всех непостоянных значений).

Символьный диапазон определяется как пара символьных выражений. Для символьных диапазонов также определяются операции сложения, вычитания, умножения и деления. К ним добавляются операции пересечения и объединения. Для переменной V вводится понятие define range S_V — символьного отрезка, соответствующего диапазону возможных значений V в месте её определения. Так как программа представлена в SSA форме, определение переменной и присвоение ей значения происходит ровно в одном месте. Также вводится понятие use range для переменной V и инструкции P: $S_{V,P}$. Этот символьный диапазон соответствует возможным значениям V в месте инструкции P. Такой диапазон может отличаться от define range, поскольку на пути к инструкции P могут быть условные переходы с предикатами, ограничивающими значение V.

1.3.3. Вычисление диапазонов

Для расчёта описанных выше диапазонов рассматриваются два вида зависимостей: зависимости по данным и зависимости потока управления.

Зависимости по данным используются для расчёта define range S_V, то есть диапазона возможных значений V в месте определения. Он вычисляется через use range аргументов в инструкции, определяющей V. Например, если $V = a + b$, то $S_V = S_{a,V} + S_{b,V}$. Для $V = \phi(a, b)$ используется объединение, поскольку ϕ может вернуть любой из аргументов: $S_V = S_{a,V} \cup S_{b,V}$. Аналогично вычисляется define range для других инструкций.

Зависимости потока управления используются для уточнения диапазона значений переменной V в инструкции P ($S_{V,P}$), который может отличаться от S_V. Для уточнения используются условные переходы, лежащие на путях P, в которых используются предикаты, затрагивающие V. При этом рассматриваются только условные переходы, удовлетворяющие двум условиям. Во-первых, узел, соответствующий P в графе потока управления, должен доминироваться узлом, соответствующим условному переходу. Другими словами, любой путь из входа в функцию в инструкцию P должен проходить через рассматриваемый условный переход. Во-вторых, P должна быть достижима только из одного потока условного перехода. В таком случае условие, соответствующее этому потомку, считается выполненым, и применяется для уточнения диапазона значений V. Например, для условия $V \geqslant y$ диапазон значений V будет пересечён с $[y, \top]$. Также учитываются предикаты, в которых V фигурирует не напрямую, а через афинное преобразование. Например, $2 * V + 3 < W$. В этом случае диапазон будет пересечён с $[\bot, \frac{W-3}{2}]$.

Выход за пределы массива размера n при обращении по индексу $index$ (инструкция P) фиксируется, если $S^{max}_{n,P} \prec S^{max}_{index,P}$, то есть максимальная оценка размера меньше либо равна максимальной оценки индекса, или $S^{min}_{index,P} \prec -1$, то есть индекс может быть отрицательным. Если $S^{max}_{index,P} \prec S^{min}_{n,P} - 1$ и $0 \prec S^{min}_{V,P}$, то выход за пределы массива точно не может случиться. В остальных случаях алгоритм не может однозначно определить, возможно ли переполнение. В таких случаях алгоритм по умолчанию не сообщает об ошибке, поскольку это приводило бы к слишком большому числу ложных срабатываний, и анализатором было бы очень сложно пользоваться для обработки больших программ.

Алгоритм использует идею анализа «по требованию». Вычисление диапазонов значений начинается в инструкциях, в которых происходит непосредственное обращение к массиву. Для проверки корректности вычисляются диапазоны значений размера массива и индекса, по которому производится обращение. Поскольку в графе зависимостей между переменными могут присутствовать циклы, используется специальный итерационный алгоритм, гарантирующий прогресс и итоговое завершение работы анализатора (итерации выполняются до достижения неподвижной точки).

Выводы по главе 1

В данной главе была подробно описана проблема выхода за пределы динамического массива в C/C++ программах, показана её актуальность. Был приведён обзор существующих решений этой проблемы, описаны ключевые характеристики различных статических подходов. Наиболее оптимальными, с точки зрения производительности, являются подходы, использующие анализ «по требованию». Также был обоснован выбор алгоритма из статьи [6] в качестве базового подхода для данной работы, приведён более подробный обзор этого алгоритма.

ГЛАВА 2. РАЗРАБОТАННЫЕ УЛУЧШЕНИЯ

В данной работе было проведено подробное исследование подхода, представленного в статье [6]. В связи с отсутствием реализации описанного алгоритма в открытом доступе он был реализован с нуля в полном соответствии с приведённым в статье описанием. В ходе исследования основное внимание было уделено недостаткам этого подхода. Были выявлены случаи, в которых описанный алгоритм работает некорректно. Ниже представлено описание таких случаев, а также предложены способы решения этих проблем.

2.1. Обработка циклов

В ходе проведённого исследования были выявлены две проблемы базового алгоритма, связанные с обработкой циклов. Эти проблемы приводят к ложным срабатываниям. Ниже представлено более подробное описание недостатков и предложены улучшения, направленные на их устранение.

2.1.1. Монотонно изменяющиеся переменные

Листинг 1 – Простой цикл: C++

```
int main()
{
    int * arr = new int[10];
    for (int x = 0; x < 10; ++x)
    {
        arr[x] = 5;
    }

    return 0;
}
```

В листинге 1 представлен фрагмент C++ кода с простым циклом, в котором значение переменной x меняется от нуля до девяти включительно. На рисунке 2 представлена соответствующая ему SSA форма.

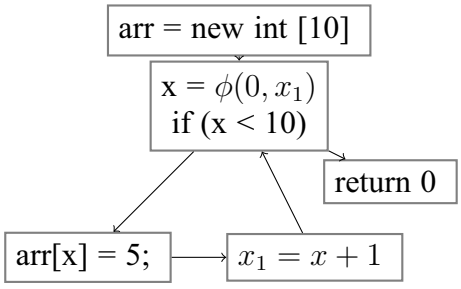

Рисунок 2 – Простой цикл: SSA форма

В данном случае алгоритм, описанный в статье [6], будет работать следующим образом. Для проверки корректности записи в массив будет посчитан диапазон возможных значений переменной x в месте записи в массив. Для этого сначала будет посчитан диапазон значений x в месте определения. Изначально в качестве диапазона будет взят $[\bot, \top]$. Поскольку x выражается как ϕ-инструкция, необходимо посчитать диапазон значений x_1 в месте определения x и объединить с $[0, 0]$ (диапазоном значений второго аргумента). Диапазон значений x_1 вычисляется через диапазон значений x в месте определения x_1. Изначально для x был взят отрезок $[\bot, \top]$, однако в месте определения x_1 также будет применён предикат $x < 10$. Таким образом, диапазон значений x_1 будет $[\bot, 10]$. Объединяя его с $[0, 0]$, получаем $[\bot, 10]$ и для x. Это значит, что в данном примере анализатор посчитает, что x может быть отрицательным, а значит, запись в массив некорректна. Однако нетрудно видеть, что в действительности это не так, x не может быть меньше нуля, а запись в массив всегда корректна. Таким образом, алгоритм выдаёт ложное срабатывание на данном примере.

Для решения проблемы предлагается использовать дополнительное правило для вычисления define range. Предположим, что значение x определено как $\phi(a, b)$, при этом $b = f(x)$. Пусть f удовлетворяет условию, что последовательность $a, f(a), f(f(a)), \ldots$ — монотонна (не умаляя общности, предположим, что последовательность возрастает). Нетрудно видеть, что в таком случае множество возможных значений x содержится в $\{f^i(a) : i \in [0..\inf)\}$ и что все элементы этого множества не меньше a. Тогда для вычисления define range x применяется условие $x \geqslant a$. Примером такой функции f является прибавление или вычитание значения с постоянным знаком.

За счёт использования описанного выше правила в приведённом примере define range переменной x будет $[0, 10]$, т. к. прибавление единицы удовлетворяет сформулированному выше условию. Use range в инструкции записи в массив будет $[0, 9]$. Таким образом, в данном случае удаётся избежать ложного срабатывания.

2.1.2. Обработка предиката «не равно»

В листинге 2 представлен фрагмент C++ кода с циклом, аналогичным представленному листингом 1, но в котором значение счётчика ограничено с

Листинг 2 – Цикл с условием «не равно»: C++

```
int main()
{
    int * arr = new int[10];
    for (size_t x = 0; x != 10; ++x)
    {
        arr[x] = 5;
    }

    return 0;
}
```

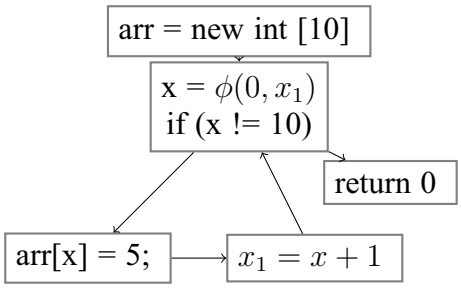

Рисунок 3 – Цикл с условием «не равно»: SSA форма

помощью условия «не равно». Такие циклы очень часто встречаются в программах. На рисунке 3 представлена соответствующая ему SSA форма.

Подход из статьи [6] учитывает предикат $x \neq y$ при вычислении use range переменной v только в том случае, если равенство $x = y$ выполняется для граничного значения v. В таком случае диапазон значений v сокращается на одно значение. Из-за этого алгоритм неспособен корректно обрабатывать циклы, в которых значение счётчика ограничено таким предикатом. В примере на рисунке 3 диапазон значений x без учёта предиката $x \neq 10$ будет $[0, \top]$. Равенство $x = 10$ не соответствует граничному значению, значит, предикат будет проигнорирован. В результате алгоритм посчитает, что запись в массив может выйти за границы, хотя легко видеть, что это не так.

Решение данной проблемы похоже на решение предыдущей проблемы и заключается в введении дополнительного правила для вычисления use range. Предположим, что значение x определено как $\phi(a, b)$, при этом $b = f(x)$. Пусть f снова удовлетворяет условию, что последовательность $a, f(a), f(f(a)), \ldots$ — монотонна (опять считаем, что последовательность возрастает). Предположим, что условие $x \neq y$ всегда выполняется в инструкции P, для которой вычисляется use range переменной x. Пусть существует такое c, что $y = f^c(a)$.

Нетрудно видеть, что последовательность значений x задаётся как $x_i = f^i(a)$. Из условия существования $c : y = f^c(a)$ и предиката $x \neq y$ следует, что всего может быть не более чем c значений x. Из свойства функции f следует, что $x < f^c(a) = y$. Таким образом, при вычислении use range применяется предикат $x < y$.

В примере на рисунке 3 $f(x) = x + 1$, $y = 10$, $a = 0$, $c = 10$. Таким образом, анализатор способен вывести, что $x < 10$, за счёт чего обращение к массиву будет обработано корректно.

2.2. Учёт предикатов

Листинг 3 – Цикл с дополнительным условием внутри: C++

```
int main()
{
    int * arr = new int[7];
    for (size_t x = 0; x < 7; ++x)
    {
        if (x < 7)
        {
            arr[x] = 5;
        }
    }

    return 0;
}
```

В цикле, представленном в листинге 3, значение переменной x, использующейся как индекс при записи в массив, дополнительно ограничено числом 7. Таким образом, значение счётчика цикла находится в диапазоне $[0, 9]$, однако значение индекса должно быть меньше семи в месте записи в массив. Программа в SSA форме представлена на рисунке 4. SSA форма похожа на

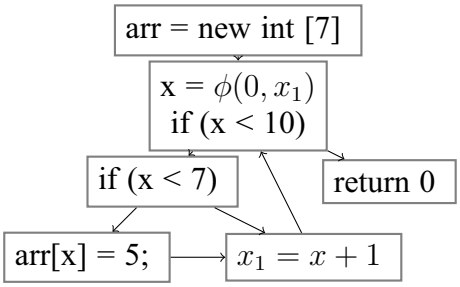

Рисунок 4 – Цикл с дополнительным условием внутри: SSA форма

предыдущие примеры и отличается добавлением инструкции условного перехода.

Проблема подхода [6] состоит в правиле, определяющем, какие предикаты должны учитываться при вычислении use range переменной v в инструкции P. Во-первых, условный переход, использующий предикат, должен доминировать инструкцию P, то есть все пути из входа в функцию в P должны проходить через предикат. Во-вторых, P должна быть достижима только из одного потомка условного перехода. Однако, как нетрудно видеть из рисунка 4, в данном случае инструкция записи в массив достижима из обоих потомков условного перехода. Данная инструкция является непосредственным потомком по ветке `true`, но также есть и путь из потомка по ветке `false`, проходящий ещё раз через условный переход. Это приводит к тому, что условие $x < 7$ не применяется для вычисления use range x в месте записи в массив, из-за чего анализатор выдаёт несуществующую ошибку.

Описанная проблема может быть решена модификацией второго правила, определяющего, должен ли учитываться предикат при вычислении use range. Модифицированное правило ослабляет требование, что инструкция P должна быть достижима только из одного потомка условного перехода, использующего предикат. Ослабленное требование заключается в том, что должен быть ровно один потомок S условного перехода C, такой что P достижима из C, игнорируя все рёбра из C, кроме $C \to S$. В таком случае условие, соответствующее переходу в S, считается выполненым.

Покажем, что предложенная модификация сохраняет корректность. Рассмотрим произвольный путь в инструкцию P. По первому правилу P доминируется C, а значит, путь обязан пройти через C. Рассмотрим последнее ребро на этом пути, исходящее из C. Пусть это ребро $C \to T$. Если $T = S$, значит, условие, соответствующее переходу в S, выполнено, что и требовалось доказать. Если же $T \neq S$, то путь в P проходит по другому ребру из C, а значит, предположение о том, что ребро $C \to T$ является последним на пути, неверно.

При использовании модифицированного правила предикат $x < 7$ будет учтён в месте записи в массив, т. к. из вершины $x_1 = x + 1$ нет пути, не проходящего по ребру, соответствующему условию $x < 7$. Таким образом, предложенная модификация позволяет избавиться от ложного срабатывания в приведённом примере, при этом сохраняя общую корректность анализа.

2.3. Межпроцедурный анализ

В статье [6] описывается лишь алгоритм без межпроцедурного анализа. Каждая функция анализируется независимо, информация о возможных значениях аргументов игнорируется. Каждый аргумент рассматривается точно так же, как и любое значение, приходящее извне. На практике программы обычно состоят из большого числа маленьких функций, и анализа одной изолированной функции недостаточно, чтобы точно определить, всегда ли обращение к массиву корректно.

В данной работе анализ был расширен до межпроцедурного. В целом известно два общих подхода к межпроцедурному анализу: в одном подходе при анализе вызывающей стороны определяются возможные значения аргументов и используются для анализа вызываемой функции, в другом подходе при анализе вызываемой функции определяются условия, которым должны соответствовать аргументы, и проверяются при непосредственном вызове. В данной работе используется второй вариант. Основная идея описана в статье [3]. Функции анализируются в порядке топологической сортировки: от вызываемой к вызывающей. Если в графе вызовов есть цикл (рекурсия), то топологическая сортировка неопределена, и в таком случае цикл разрывается в случайных местах. Также вводится понятие триггера, который задаёт условие, влияющее на факт выхода за пределы массива. При анализе вызываемой функции формируется множество триггеров. При анализе вызова функции проверяется выполнимость каждого триггера.

2.3.1. Триггеры

Триггер задаётся упорядоченной парой символьных выражений e_1 и e_2. Оба выражения должны состоять только из констант и аргументов функции (включая их афинные преобразования). Смысл триггера в том, что если $e_1 \prec e_2$, то это приведёт к выходу за пределы массива. Также триггер должен хранить внутри себя инструкцию, в которой может произойти выход за пределы массива, чтобы при его срабатывании можно было понять, в каком месте случится ошибка.

Для иллюстрации рассмотрим упрощённый фрагмент кода из ядра Линукс версии 2.5.53, представленный листингом 4. В данном примере функция `get_slot_by_minor` вызывает функцию `get_drv_by_nr`. При анализе `get_drv_by_nr` будет произведено построение триггеров для этой функции.

Листинг 4 – Межпроцедурный анализ

```
#define ISDN_MAX_DRIVERS 32
#define ISDN_MAX_CHANNELS 64

struct isdn_driver;
struct isdn_slot;

struct isdn_driver * drivers[ISDN_MAX_DRIVERS];

struct isdn_driver * get_drv_by_nr(int di)
{
    struct isdn_driver * drv;
    drv = driver[i];
    // ...

}

struct isdn_slot * get_slot_by_minor(int minor)
{
    int di, ch;
    struct isdn_driver * drv;
    for (di = 0; di < ISDN_MAX_CHANNELS; di++)
    {
        drv = get_drv_by_nr(di);
        // ...
    }
    // ...

}
```

Затем эти триггеры будут проверены при анализе `get_slot_by_minor` (она будет анализирована после `get_drv_by_nr`, поскольку функции анализируются в порядке топологической сортировки). Нетрудно видеть, что в данном случае выход за пределы массива в функции `get_drv_by_nr` может произойти тогда и только тогда, когда $ISDN\_MAX\_DRIVERS \leqslant di \vee di \leqslant -1$. Таким образом, для функции должно быть построено два триггера, соответствующих этим неравенствам. В функции `get_slot_by_minor` происходит вызов `get_drv_by_nr`, и в месте вызова можно расчитать диапазон возможных значений аргумента, передаваемого в `get_slot_by_minor`. За счёт этого можно вывести, что один из триггеров выполняется в месте вызова, а значит, в программе есть ошибка.

Использование триггеров позволяет учесть информацию о значениях аргументов функций, а также о завимостях между этими аргументами. Ниже представлено более подробное описание использования триггеров.

2.3.2. Построение триггеров

Построение триггеров происходит при определении корректности доступа к массиву в тех случаях, когда одно или несколько рассматриваемых символьных значений зависят от аргументов функции. Как уже было сказано в первой главе, выход за пределы массива размера n при обращении по индексу $index$ (инструкция P) фиксируется, если $S^{max}_{n,P} \prec S^{max}_{index,P}$ или $S^{min}_{index,P} \prec -1$. Таким образом, для проверки используются диапазоны n и $index$. Если $S^{max}_{n,P}$ или $S^{max}_{index,P}$ зависит от аргумента функции, то их сравнение в общем случае невозможно. В таком случае для функции добавляется триггер $S^{max}_{n,P} \prec S^{max}_{index,P}$, хранящий также инструкцию P. Выполнение этого триггера означает выход за пределы массива в результате исполнения инструкции P. Аналогично обрабатывается условие $S^{min}_{index,P} \prec -1$. В результате обработки функции все сгенерированные триггеры сохраняются в глобальном контексте анализатора для дальнейшей проверки в месте вызова. Если при проверке был сгенерирован хотя бы один триггер, то выход за пределы массива не фиксируется, а проверка откладывается для вызывающей стороны.

Вернёмся к примеру, представленному листингом 4. При анализе обращения к массиву $drivers$ внутри функции `get_drv_by_nr` (назовём эту инструкцию P) необходимо проверить два условия: $32 \prec S^{max}_{di,P}$ и $S^{min}_{di,P} \prec -1$. Поскольку di является аргументом функции, оба условия приведут к созданию триггера для рассматриваемой функции, ассоциированных с инструкцией P. Триггеры будут сохранены и проверены во время анализа функции `get_slot_by_minor`.

Описанное правило применяется в тех случаях, когда рассматриваемые символьные выражения зависят только от констант и аргументов функции. Точно так же обрабатываются случаи, когда символьное выражение задаётся функцией от нескольких аргументов. Например, если бы функция принимала также аргумент si, а обращение происходило по индексу $2 * si - di$, то такое символьное выражение тоже привело бы к созданию триггера. За счёт этого учитываются зависимости между аргументами, которые нередко имеют место в сложных функциях.

2.3.3. Проверка триггеров

При анализе инструкции вызова функции происходит проверка выполнимости триггеров, сохранённых для этой функции. Напомним, что триггер состоит из двух символьных выражений, в которых могут присутствовать аргументы вызываемой функции. Каждому аргументу вызываемой функции соответствует какое-то значение из вызывающей функции. Для каждого такого значения рассчитывается диапазон значений в месте вызова функции (use range). Таким образом, в общем случае символьным выражениям e_1 и e_2, сравниваемым для проверки триггера, соответствуют символьные отрезки r_1 и r_2. После вычисления r_1 и r_2 проверка триггера происходит аналогично проверке выходу за пределы массива в простом случае. Если $r_1^{max} \prec r_2^{min}$, то триггер всегда выполняется, и в таком случае фиксируется ошибка. Если $r_2^{max} \prec r_1^{min} + 1$, то ошибки точно нет. В противном случае ошибка потенциально может произойти, но не точно. По умолчанию в спорных ситуациях ошибка не фиксируется для избежания слишком большого числа ложных срабатываний.

В примере из листинга 4 при анализе вызова функции `get_drv_by_nr` из функции `get_slot_by_minor` проверяются два триггера, построенные при анализе `get_drv_by_nr`: $32 \prec S_{di,call}^{max}$ и $S_{di,call}^{min} \prec -1$, где di является аргументом `get_drv_by_nr`, а $call$ — инструкция вызова функции. В данном случае диапазон $S_{di,call}$ будет вычислен как $[0, 63]$. В результате будет выполнено условие первого триггера, что приведёт к сообщению об ошибке. За счёт хранения инструкции, в которой происходит обращение к массиву, вместе с триггером анализатор сможет сообщить как место вызова функции, так и место внутри функции, в котором случается ошибка, что крайне важно для пользователя.

Выводы по главе 2

В главе 2 представлены результаты исследования алгоритма, описанного в статье [6]. В ходе исследования были выявлены различные недостатки алгоритма, которым было уделено внимание в данной работе. Показаны примеры кода, когда алгоритм работает некорректно, предложены модификации исходного алгоритма, направленные на устранение выявленных проблем. Доказана корректность предложенных модификаций.

Одним из наиболее значимых улучшений является использование межпроцедурного анализа, что крайне важно для анализа промышленных про-

грамм. Используемый подход к межпроцедурному анализу базируется на идеях, использованных в статье [3], однако адаптирован для алгоритма из статьи [6].

ГЛАВА 3. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ И РЕЗУЛЬТАТЫ

3.1. Реализация

Предложенный подход был реализован на языке C++. Как и в статье [6], было принято решение анализировать не исходный код на C/C++, а промежуточное представление LLVM [28]. У такого подхода есть несколько весомых преимуществ [21]. Во-первых, LLVM-IR состоит из более простых языковых конструкций, нежели C или тем более C++, что упрощает анализ и позволяет охватить большее число возможностей языка с меньшими усилиями. Во-вторых, код LLVM-IR представляет из себя результат работы компилятора и является очень близким к тому, что будет реально выполняться. Это позволяет найти ошибки, появившиеся в результате трансформаций, выполняемых компилятором. Также инфраструктура LLVM содержит огромное число встроенных оптимизаций, средств для анализа и т. п., что можно использовать при реализации. Ещё одним преимуществом анализа LLVM-IR является то, что за счёт этого автоматически поддерживается любой язык, для которого есть компилятор в LLVM-IR, не только C и C++ (являющиеся примерами таких языков). Также стоит отметить, что программа в LLVM-IR всегда представлена в SSA форме, на которой базируется описанный ранее алгоритм. Альтернативное решение состоит в построении SSA формы для исходной программы на C/C++ и её анализе, однако было показано, что анализ LLVM-IR имеет существенные преимущества. Язык C++ был выбран для реализации анализатора преимущественно по двум причинам. Во-первых, библиотека LLVM написана на C++ и может быть использована напрямую. Для большого числа популярных языков написаны интерфейсы вызова сторонних функций библиотеки LLVM, однако они могут быть неполными или устаревшими. Во-вторых, за счёт использования C++ проще достичь высокой производительности.

Изначально подход из статьи [6] был реализован в полном соответствии с описанием. После этого было проведено исследование работы алгоритма на различных тестовых примерах, найдены недостатки и разработаны способы их устранения, описанные в предыдущей главе.

3.1.1. LLVM

В LLVM каждое значение (`llvm::Value`) идентифицируется обычным указателем (`llvm::Value *`), поэтому диапазоны переменных ассоциируются с указателями на значения. Для идентификации инструкций в рамках

use range используются указатели на базовые блоки (`llvm::BasicBlock`), поскольку use range переменной одинаков во всех инструкциях из одного базового блока. Для выявления инструкций, выделяющих блоки памяти, используется функция `llvm::isAllocationFn`. Размер массива вычисляется как число байт, выделяемых такой функций, поделённое на размер типа данных в массиве. Информация о размере типа в байтах также предоставляется LLVM. В качестве инструкций, обращающихся к памяти, рассматриваются `store` и `load`. Анализатор начинает вычисление диапазонов, только встретив одну из этих инструкций, то есть анализ выполняется «по требованию».

Также LLVM предоставляет полезные функции и классы для расчёта use range. Для проверки того, что инструкция доминируется предикатным узлом, необходимой для уточнения use range, используется `llvm::DominatorTree`, с помощью которого можно проверять предикат доминируемости. Другая функциональность, необходимая для уточнения диапазона значений, заключается в проверке достижимости одного базового блока из другого. В общем случае такая проверка невозможна, однако в LLVM есть функция `llvm::isPotentiallyReachable`, которая возвращает `false`, если базовый блок точно недостижим, и `true` в противном случае. Как было показано в предыдущей главе, простая проверка достижимости не является достаточно точной и может приводить к неправильным результатам. При более точном подходе необходимо игнорировать рёбра из C, кроме ребра $C \to S$. Поэтому использовалась модифицированная версия функции, игнорирующая такие рёбра. Ещё одной важной функциональностью из LLVM является оптимизация `mem2reg`, которая превращает использование переменной, хранимой на стеке, в использование регистра. Данная оптимизация крайне важна, поскольку анализ использует только переменные, представленные регистрами.

При обнаружении ошибки о ней необходимо сообщить пользователю в понятном формате. Для этого нужно сопоставлять инструкции из промежуточного представления LLVM выражениям в исходном коде на высокоуровневом языке. Эта проблема решается за счёт использования отладочной информации, генерируемой компилятором при использовании специального флага. В LLVM такая информация представлена классом `DebugLoc`. Если программа скомпилирована с отладочной информацией, то по инструкции LLVM можно полу-

чить её `DebugLoc` и сообщить пользователю, где в исходном коде находится ошибка.

3.1.2. Gated Single Assignment

Одной из наиболее важных деталей алгоритма является использование Gated Single Assignment формы (GSA) [29]. Отличие этой формы от SSA заключается в том, что аргументы ϕ-инструкций также содержат условия, которые гарантировано выполняются, если данное значение возвращается ϕ-инструкцией. Это позволяет существенно повысить точность анализа, отбрасывая невозможные значения переменных.

В реализации анализатора использовался алгоритм для конвертации SSA формы в GSA, описанный в статье [30]. Данный алгоритм является одним из наиболее простых в реализации. Он не требует, чтобы исходная программа была в SSA, однако существенно упрощается, если это выполнено. В основе алгоритма лежит понятие «выражение пути» (path expression), которое является регулярным выражением над алфавитом, состоящим из рёбер в графе потока управления. Пути, приходящие в ϕ-инструкцию, представляются выражениями пути. В конце работы алгоритма выражения пути превращаются в предикаты GSA-формы. Несмотря на свою простоту, алгоритм также является более эффективным по сравнению со своими аналогами. В результате работы алгоритма для каждого операнда каждой ϕ-инструкции известен определённый набор условий, который должны выполняться, чтобы данное значение было результатом ϕ-инструкции. После этого вычисленные условия применяются для уточнения диапазонов значений ϕ-инструкций в месте их определения.

3.1.3. Символьные вычисления

Базовой составляющей алгоритма являются символьные вычисления, используемые для расчёта диапазонов значений переменных в программе. В статье [6] используются три понятия для описания символьных вычислений: символы-атомы, символьные выражения и символьные диапазоны. Изначально множество символов-атомов пустое, однако оно пополняется в ходе анализа. Это происходит, когда результат какой-то операции не может быть представлен афинной функцией существующих символов-атомов. В этом случае добавляются новые символы-атомы, являющиеся афинными преобразованиями существующих атомов. Символьные выражения являются афинными преоб-

разованиями символов-атомов. Символьные диапазоны представляются парой символьных выражений.

Для представления символов-атомов используются четыре класса: `atomic_const`, `atomic_var`, `atomic_linear`, `atomic_bin_op`. `atomic_const` соответствует константному атому и хранит одно число. `atomic_var` соответствует некоторому значению в программе и хранит указатель на это значение (являющееся идентификатором). `atomic_linear` используется для представления атома, являющегося произведением другого атома и константы. Наконец, `atomic_bin_op` представляет применение простейшей бинарной операции к двум атомам. С помощью этих классов можно представить любой символ-атом, т. к. он задаётся как афинное преобразование других символов-атомов. Нетрудно видеть, что `atomic_linear` может быть представлен как `atomic_bin_op` с операцией умножения, однако за счёт выделения `atomic_linear` в отдельный класс упрощается распознание предикатов, используемых для учёта зависимостей потока управления.

Символьное выражение представляется классом `sym_expr`, внутри которого хранятся два числа `coeff` и `delta` и символ-атом `atom`. Такое представление соответствует символьному выражению, являющемуся афинным преобразованием атома `atom`: `coeff · atom + delta`. Помимо этого внутри `sym_expr` хранится специальный флаг (`boost::tribool`), для задания $\bot$ и $\top$. Флаг может принимать три значения, два из которых соответствуют $\bot$ и $\top$, а третье соответствует обычному символьному выражению. Всего в программе существует ровно один $\bot$ и ровно один $\top$. Для символьных выражений определены операции сложения, вычитания, умножения, деления, проверка равенства и неравенства и сравнение путём перегрузки операторов. Определения этих операций соответствуют описанию из статьи [6]. Наибольшую трудность представляют операции умножения и деления, так как для них необходимо рассматривать большое число частных случаев, например, когда какой-то коэффициент равен нулю. Помимо этого определены операции `meet` и `join`.

Последней сущностью, используемой для символьных вычислений, является символьный диапазон. Для его представления используется простая структура `sym_range`, состоящая из двух полей типа `sym_expr`. Для удобства определяются два специальных диапазона: `empty` и `full`, соответствующие пустому диапазону ($[\top, \bot]$) и диапазону, содержащему любой другой

диапазон ($[\bot, \top]$). Операции сложения, умножения, вычитания деления определены как для пары диапазонов, так и для пары из диапазона и выражения. Также определены операции пересечения и объединения двух диапазонов.

3.1.4. Обработка больших программ

Одной из главных проблем при обработке больших программ, состоящих из большого числа файлов и сложного механизма сборки, является необходимость скомпилировать программу в LLVM-IR. Для решения проблемы используется утилита `whole-program-llvm` [31], написанная на языке Python, которая используется также в проекте KLEE [32]. Утилита подменяет компилятор (`gcc` или `clang`) обёртками, которые помимо компиляции кода также сохраняют дополнительную служебную информацию о расположении кода в генерируемых файлах. При линковке объектных файлов служебная информация объединяется. Таким образом, получающийся в результате сборки исполняемый файл (или библиотека) содержит всю необходимую информацию для восстановления биткода. На последнем шаге из файла извлекается биткод, соответствующий всей программе целиком. При этом сохраняется отладочная информация, если она была сгенерирована компилятором. В дальнейшем биткод полностью передаётся анализатору. Поскольку утилита является простой обёрткой над компилятором, её очень легко использовать с различными системами сборки, такими как `GNU Make` [33]. Достаточно лишь верно указать используемый компилятор, чтобы обычно делается с помощью переменных окружения. Также были написаны небольшие скрипты для автоматизации стандартного процесса анализа: компиляция программы с помощью `wllvm`, оптимизация `mem2reg`, извлечение биткода, запуск анализатора.

3.2. Экспериментальные результаты

После реализации анализатора были проведены различные исследования для оценки получившихся результатов.

3.2.1. Результаты использования анализатора

Для проверки работоспособности и эффективности разработанного анализатора было произведено его тестирование на крупных open-source проектах. В рамках тестирования были проанализированы система сборки CMake версии 3.1.0 и игра Multi Theft Auto версии 1.3.1. Оба проекта состоят из сотен тысяч строчек кода как на C, так и на C++.

CMake содержит 180000 строчек кода на C++ и 170000 строчек кода на C. После компиляции в LLVM размер биткода составил 31 мегабайт. Обработка заняла чуть больше одиннадцати минут. Анализатор запускался в режиме, в котором спорные ситуации пропускаются. В таком режиме было найдено восемь ошибок. Найденные ошибки были проверены вручную, в результате оказалось, что пять из них являются реальными ошибками, а остальные три являются ложными срабатываниями. В новых версиях CMake ошибки были исправлены, что подтверждает факт ошибок.

Multi Theft Auto содержит 410000 строчек кода на C++ и 350000 строчек кода на C. Размер биткода составил 182 мегабайта. Обработка заняла немногим более сорока минут, что в целом приемлемо для столь большого проекта. Вновь использовался режим, в котором спорные ситуации пропускаются. Всего было найдено одиннадцать ошибок. В результате ручной проверки семь из них оказались действительными ошибками. Ещё четыре ошибки были обнаружены неверно.

Таблица 1 – Результаты работы анализатора

Программа	Строки кода	Биткод	Время	Общее число	TP	FP
CMake 3.1.0	350000	31M	11:03	8	5	3
MTA 1.3.1	760000	182M	42:49	11	7	4

В таблице 1 подведены итоги экспериментальных запусков. Вторая колонка содержит число строк кода, третья колонка показывает размер биткода, полученного в результате компиляции. Следующая колонка содержит время работы анализатора. Наконец, последние три колонки содержат общее число найденных ошибок, число верно найденных ошибок (в таблицах обозначается как TP, true positive) и число ложных срабатываний (в таблицах обозначается как FP, false positive). Таким образом, разработанное средство статического анализа действительно способно обрабатывать программы большого размера за разумное время и находить в них ошибки. К сожалению, число ложных срабатываний оказалось сопоставимо с числом реальных ошибок, поскольку было сделано большое число упрощений с целью повышения скорости работы. Однако в целом число ложных срабатываний невелико, и ручная проверка занимает довольно мало времени. Во многих ситуациях способность находить реальные ошибки является более важной, чем необходимость потратить немного времени на ручной анализ.

3.2.2. Сравнение с другими анализаторами

Также было проведено сравнение разработанного анализатора с другими доступными анализаторами C/C++ кода. Использовались следующие анализаторы: Cppcheck, PVS-Studio, Splint, LLBMC, а также реализация анализатора, описанного в статье [6], и реализация предложенного подхода. В первой главе присутствует краткое описание каждого анализатора. Cppcheck запускался с аргументами `-std=c++11 -std=c11 -enable=warning -inconclusive`, из найденных ошибок выбирались содержащие фразу `out of bounds`, соответствующие выходу за пределы массива. PVS-Studio запускался со стандартными настройками, рассматривались ошибки V557 (выход за пределы массива). Splint запускался с флагами `-weak -hints +bounds`. Утилита LLBMC, к сожалению, распространяется лишь в бинарном формате и не обновлялась уже много лет, в связи с чем поддерживает лишь очень старую версию LLVM, поэтому произвести анализ с её помощью не удалось.

Сравнение проводилось на Multi Theft Auto 1.3.1 и CMake 3.1.0, упомянутых ранее, а также на наборе синтетических тестов, представленных в листинге А.1.

Таблица 2 – Сравнение анализаторов: Multi Theft Auto 1.3.1

–	TP	FP	Время работы
Cppcheck	1	8	11:53
PVS-Studio	4	0	10:36
Splint	—	—	—
LLBMC	—	—	—
Li et at.	7	202	37:55
Предложенный метод	7	4	42:49

Результаты сравнения на Multi Theft Auto 1.3.1 представлены в таблице 2. Анализаторы Cppcheck и PVS-Studio вновь показали довольно хорошее время работы, на сей раз оно составило немногим более десяти минут. Cppcheck смог найти лишь одну ошибку, при этом было восемь ложных срабатываний из-за проблем с обработкой директив препроцессора. PVS-Studio удалось найти четыре реальных ошибки, причём снова без единого ложного срабатывания. С анализаторами Splint и LLBMC возникли те же проблемы, что и для CMake, поэтому результаты их работы собрать не получилось. Реализованный в рамках данной работы анализатор нашёл на три ошибки больше, чем PVS-Studio.

Вновь не обошлось без небольшого числа неверно найденных ошибок, что в целом не критично для практического использования. Время работы оказалось сильно больше, чем у PVS-Studio и Cppcheck, однако вполне приемлемо для проекта, содержащего почти миллион строчек кода. Также была протестирована реализация алгоритма из статьи [6]. Она нашла те же семь реальных ошибок, но суммарное число найденных ошибок оказалось больше двухсот. Учитывая проделанные изменения, можно сделать вывод, что, скорее всего, все остальные ошибки являются недействительными. Ручная проверка всех сообщений об ошибках, найденных этой реализацией, не проводилась ввиду большого числа таких сообщений.

Листинг 5 – Ошибка в Multi Theft Auto

```
struct CEntitySAInterface {
    int usNumberOfRefs;
};

CEntitySAInterface* *pSector =
    new CEntitySAInterface* [2*NUM_SectorRows];

// ...

for ( int i = 0; i != NUM_SectorRows; i++ )
{
    pSector[i] = // ...
}

// ...

for ( int i = NUM_SectorRows; i != 2*NUM_SectorRows; i++ )
{
    pSector[i] = // ...
}

// ...

for ( int i = 0; i <= 2*NUM_SectorRows; i++ )
{
    pSector[i]->usNumberOfRefs++;
}
```

В листинге 5 продемонстрирована упрощённая версия кода одной из ошибок, найденных в Multi Theft Auto 1.3.1, которую не удалось найти с помощью Cppcheck или PVS-Studio. Как видно, переменная `pSector` указывает на массив размера `2 * NUM_SectorRows`. При этом значение

NUM_SectorRows заранее не известно. Сначала заполняется первая половина массива, затем вторая. При заполнении массива все обращения к нему корректны. Однако после этого происходит проход по всему массиву, в котором есть ошибка: в качестве условия выхода из цикла используется предикат $\leqslant$, хотя должен использоваться предикат $<$ или $!=$. Причина, по которой ни PVS-Studio, ни Cppcheck не смогли найти такую ошибку, заключается, по-видимому, в том, что размер массива, а также ограничение на значение счётчика цикла, не выражаются константой или какой-то переменной. В данном случае размер массива представляется произведением константы и переменной. В то же время подход из статьи [6] учитывает зависимости от переменных, выражаемые афинными преобразованиями, за счёт чего такая ошибка находится анализатором.

Таблица 3 – Сравнение анализаторов: CMake 3.1.0

–	TP	FP	Время работы
Cppcheck	1	0	5:59
PVS-Studio	1	0	5:18
Splint	—	—	—
LLBMC	—	—	—
Li et at.	5	79	9:12
Предложенный метод	5	3	11:03

Результаты сравнения на CMake 3.1.0 представлены в таблице 3. Анализаторы Cppcheck и PVS-Studio нашли по одной ошибке с абсолютной точностью. Причём были найдены разные ошибки. Время работы обоих анализаторов составило менее шести минут. Splint умеет обрабатывать только код на языке C, однако даже при попытке запуска только на C файлах он не мог обработать многие из них из-за ошибок, связанных с препроцессором. Как сообщалось ранее, провести анализ утилитой LLBMC не удалось, поскольку её исходный код отсутствует в публичном доступе а исполняемые файлы сильно устарели. Разработанный в рамках данной работы анализатор нашёл пять ошибок с тремя ложными срабатываниями. В то же время реализация алгоритма из статьи [6] без модификаций нашла те же пять ошибок, но при этом суммарное число сообщений об ошибках было близко к сотне. Таким образом, разработанный анализатор способен находить больше ошибок, чем существующие анализаторы. Вместе с увеличением числа находимых реальных ошибок

увеличилось и число ложных срабатываний, а также время работы. Однако оба показателя являются вполне приемлемыми.

Таблица 4 – Сравнение анализаторов: синтетические тесты

–	TP	FP	FN
scan-build	0	0	0
CppCheck	3	0	9
PVS-Studio	4	0	8
Splint	9	3	3
LLBMC	-	-	-
Li et at.	12	8	0
Разработанный анализатор	12	0	0

В таблице 4 представлены результаты сравнения на синтетических тестах (листинг А.1). Этот набор тестов содержит различные конструкции, которые нередко можно встретить в коде: условные переходы, некоторые виды циклов, а также тесты, в которых необходим межпроцедурный анализ. Для сравнения также использовался статический анализатор, поставляемый с компилятором `clang`: `scan-build` [34]. Однако, как видно из таблицы, анализатор не смог найти ни одной ошибки. Хотя авторы заявляют наличие анализа выходов за пределы массива, работает он лишь в совсем тривиальных случаях. Анализаторы Cppcheck и PVS-Studio снова продемонстрировали схожие результаты и нашли 3 и 4 ошибки соответственно. Вновь не было ни единого ложного срабатывания. Однако суммарно в тестах было 12 ошибок, поэтому число ненайденных ошибок (False Negative, FN) весьма велико. Splint использует другую стратегию и сообщает об ошибке всегда, когда не может доказать её отсутствие (то есть корректность обращения к массиву). Ему удалось найти 9 реальных ошибок, при этом было 3 ложных срабатывания. 3 ошибки оказались не найдены. Результаты были бы лучше, если бы Splint выполнял межпроцедурный анализ. Алгоритм, описанный в [6] и реализованный без модификаций, нашёл все 12 реальных ошибок, однако также выдал 8 несуществующих ошибок. Наконец, улучшенной версии анализатора, реализованной в данной работе, удалось найти также все 12 ошибок, но без единого ложного срабатывания.

Выводы по главе 3

В данной главе были описаны основные детали реализации анализатора. Обосновано решение проводить анализ на уровне промежуточного представления LLVM. Приведены экспериментальные результаты запуска анализатора на крупных проектах. Показано, что анализатор способен находить в них ошибки, работая за приемлемое время. Также было проведено сравнение с доступными статическими анализаторами C/C++ кода на тех же проектах и на наборе синтетических тестов, представленных в листинге А.1. Продемонстрировано, что в целом анализатор не уступает существующим решениям и в некоторых случаях превосходит их.

ЗАКЛЮЧЕНИЕ

Как было показано в данной работе, задача поиска выходов за пределы динамического массива в C/C++ является крайне актуальной и представляет практический интерес. Были рассмотрены различные подходы к решению данной задачи, выделены основные характеристики различных подходов. За основу был взят подход, описанный в статье [6].

В ходе работы было проведено детальное исследование подхода [6], выявлены и продемонстрированы различные недостатки. Были предложены способы устранения выявленных недостатков, показана корректность этих способов. Разработанные улучшения направлены в основном на повышение точности анализа за счёт снижения числа ложных срабатываний. Была улучшена обработка циклов и учёт зависимостей потока управления. Также анализ был расширен до межпроцедурного за счёт адаптации подхода, описанного в [3].

Алгоритм был реализован на языке C++. В результате экспериментов было показано, что анализатор способен находить ошибки в крупных проектах и работает за разумное время. Несмотря на наличие ложных срабатываний, такой анализатор является полезным инструментом разработчика. Также было проведено сравнение с доступными анализаторами. Показано, что реализованный подход в целом работает не хуже промышленных анализаторов, а также базового подхода, предложенного в [6], и в некоторых случаях превосходит их.

Разработанное средство статического анализа было внедрено в проекте «IoT Devices», реализуемом ООО «Интеллектуальные системы».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *One A.* Smashing the stack for fun and profit. Phrack, 7 (49), November 1996. — 1996.

2 A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. / D. Wagner [и др.] // NDSS. — 2000. — С. 2000–02.

3 *Xie Y.*, *Chou A.*, *Engler D.* Archer: using symbolic, path-sensitive analysis to detect memory access errors // ACM SIGSOFT Software Engineering Notes. — 2003. — Т. 28, № 5. — С. 327–336.

4 Buffer overrun detection using linear programming and static analysis / V. Ganapathy [и др.] // Proceedings of the 10th ACM conference on Computer and communications security. — ACM. 2003. — С. 345–354.

5 *Le W.*, *Soffa M. L.* Marple: a demand-driven path-sensitive buffer overflow detector // Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. — ACM. 2008. — С. 272–282.

6 *Li L.*, *Cifuentes C.*, *Keynes N.* Practical and effective symbolic analysis for buffer overflow detection // Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. — ACM. 2010. — С. 317–326.

7 Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. / C. Cowan [и др.] // Usenix Security. Т. 98. — 1998. — С. 63–78.

8 *Ruwase O.*, *Lam M. S.* A Practical Dynamic Buffer Overflow Detector. // NDSS. Т. 2004. — 2004. — С. 159–169.

9 The spread of the sapphire/slammer worm / D. Moore [и др.]. — 2003.

10 Code-Red: a case study on the spread and victims of an Internet worm / D. Moore, C. Shannon [и др.] // Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment. — ACM. 2002. — С. 273–284.

11 *Hastings R.*, *Joyce B.* Purify: Fast detection of memory leaks and access errors // In proc. of the winter 1992 usenix conference. — Citeseer. 1991.

12 *Turing A. M.* On computable numbers, with an application to the Entscheidungsproblem // Proceedings of the London mathematical society. — 1937. — Т. 2, № 1. — С. 230–265.

13 URL: www.us-cert.gov.

14 *Shahriar H.*, *Zulkernine M.* Classification of static analysis-based buffer overflow detectors // Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on. — IEEE. 2010. — С. 94–101.

15 Token-based scanning of source code for security problems / J. Viega [и др.] // ACM Transactions on Information and System Security (TISSEC). — 2002. — Т. 5, № 3. — С. 238–261.

16 Flawfinder. — URL: www.dwheeler.com/flawfinder/.

17 Statically Detecting Likely Buffer Overflow Vulnerabilities. / D. Larochelle, D. Evans [и др.] // USENIX Security Symposium. Т. 32. — Washington DC. 2001.

18 Modular checking for buffer overflows in the large / B. Hackett [и др.] // Proceedings of the 28th international conference on Software engineering. — ACM. 2006. — С. 232–241.

19 *Dor N.*, *Rodeh M.*, *Sagiv M.* CSSV: Towards a realistic tool for statically detecting all buffer overflows in C // ACM Sigplan Notices. Т. 38. — ACM. 2003. — С. 155–167.

20 *Cordeiro L.*, *Fischer B.*, *Marques-Silva J.* SMT-based bounded model checking for embedded ANSI-C software // IEEE Transactions on Software Engineering. — 2012. — Т. 38, № 4. — С. 957–974.

21 *Merz F.*, *Falke S.*, *Sinz C.* LLBMC: Bounded model checking of C and C++ programs using a compiler IR // International Conference on Verified Software: Tools, Theories, Experiments. — Springer. 2012. — С. 146–161.

22 Detection of buffer overflow vulnerabilities in C/C++ with pattern based limited symbolic evaluation / S. Ding [и др.] // Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual. — IEEE. 2012. — С. 559–564.

23 *Ding S.*, *Tan H. B. K.*, *Zhang H.* ABOR: An Automatic Framework for Buffer Overflow Removal in C/C++ Programs // International Conference on Enterprise Information Systems. — Springer. 2014. — С. 204–221.

24 Статический анализатор PVS-Studio. — URL: https://www.viva64.com/ru/pvs-studio/.

25 Статический анализатор Cppcheck. — URL: http://cppcheck.sourceforge.net/.

26 Статический анализатор Splint. — URL: http://splint.org/.

27 Efficiently computing static single assignment form and the control dependence graph / R. Cytron [и др.] // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1991. — Т. 13, № 4. — С. 451–490.

28 *Lattner C.*, *Adve V.* LLVM: A compilation framework for lifelong program analysis & transformation // Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. — IEEE Computer Society. 2004. — С. 75.

29 *Ottenstein K. J.*, *Ballance R. A.*, *MacCabe A. B.* The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages // ACM SIGPLAN Notices. — 1990. — Т. 25, № 6. — С. 257–271.

30 *Tu P.*, *Padua D.* Efficient building and placing of gating functions // ACM SIGPLAN Notices. — 1995. — Т. 30, № 6. — С. 47–55.

31 Whole Program LLVM. — URL: https://github.com/travitch/whole-program-llvm.

32 KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. / C. Cadar, D. Dunbar, D. R. Engler [и др.] // OSDI. Т. 8. — 2008. — С. 209–224.

33 GNU Make. — URL: https://www.gnu.org/software/make/.

34 Статический анализатор scan-build. — URL: http://splint.org/.

ПРИЛОЖЕНИЕ А. ТЕСТЫ ДЛЯ СРАВНЕНИЯ

Листинг А.1 содержит код со множеством обращений к массиву. На этом примере были запущены различные анализаторы, проведено сравнение.

Листинг А.1 – Тесты, использовавшиеся для сравнения анализаторов

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int external_function()
{
    int i = 50;
    scanf("%d", &i);
    return i;
}

void if_eq()
{
    int * p = (int*) malloc(8 * sizeof(int));
    int x = external_function();
    int y = 2 * x;
    if (x == 3)
        p[x] = 8;   // good

    if (2 + x == 8)
        p[x] = 8;   // good

    if (x - 2 == 8)
        p[x] = 8;   // bad

    if (x == 2)
        p[y] = 42;   // good

    p[x] = 9;   // bad

    free(p);
}

void if_cmp()
{
```

```
    int * p = (int*) malloc(8 * sizeof(int));
    int x = external_function();
    int y = -x;
    if (x <= 3) {
        if (x > 0)
            p[x] = 8;  // good

        if (x - 2 > 0)
            p[x] = 42;  // good

    }

    if (x < 0)
        if (x > -2)
            p[y] = 7; // good

    if (x > -2)
        p[y] = 10; // bad

    p[x] = 9;  // bad
    free(p);
}

void symbolic_and_numeric()
{
    int n = external_function();
    int * arr_n = (int*) malloc(n * sizeof(int));
    int * arr_10 = (int*) malloc(10 * sizeof(int));
    int x;

    for (x = 0; x < n; ++x)
    {
        arr_n[x] = 6;  // ok
        arr_10[x] = 6;  // bad

        if (x < 10)
        {
            arr_n[x] = 6;  // ok
            arr_10[x] = 6; // ok
        }
    }
```

```
    free(arr_n);
    free(arr_10);
}

void simple_for()
{
    int * arr = (int*) malloc(10 * sizeof(int));
    int x;
    for (x = 0; x < 10; ++x)
        arr[x] = 6;  // ok

    for (x = 0; x < 15; ++x)
        arr[x] = 6;  // bad

    free(arr);
}

void harder_for()
{
    int n = external_function();
    int * arr = (int*) malloc(n * sizeof(int));
    int x;
    for (x = 0; x < n; ++x)
        arr[x] = 6;  // ok

    for (x = 0; x < n + 5; ++x)
        arr[x] = 6;  // bad

    free(arr);
}

void for_with_ne()
{
    int * arr = (int*) malloc(10 * sizeof(int));
    int x;
    for (x = 0; x != 10; ++x)
        arr[x] = 6;  // ok

    for (x = 0; x != 15; ++x)
        arr[x] = 6;  // bad
```

```
    free(arr);
}

void for_with_guard()
{
    int * arr = (int*) malloc(7 * sizeof(int));
    int x;
    for (x = 0; x < 10; ++x)
    {
        if (x < 7)
            arr[x] = 5;  // ok

        arr[x] = 6;  // bad
    }

    free(arr);
}

char * tosunds_str(char * str)
{
    int i, j, n;
    char * buf;
    n = external_function();
    buf = (char *) malloc(n * sizeof(char));
    j = 0;
    for (i = 0; i < strlen(str); i++)
    {
        if (str[i] == 10)  // potential overflow, because size of
           str is unknown
            buf[j++] = '%'; // good

        buf[j++] = str[i]; // buf[j++] is bad, str[i] is potential
           overflow
        if (j >= n) break;
    }
    if (j + 1 >= n)
        j = n - 1;

    buf[j] = '\0';  // good
    return buf;
```

```
}

// interprocedural
void inter(int i)
{
    int * arr = (int*) malloc(7 * sizeof(int));
    arr[i] = i;
    free(arr);
}

// interprocedural
void inter2(int i)
{
    int * arr = (int*) malloc(7 * sizeof(int));
    arr[i] = i;
    free(arr);
}

int main()
{
    for_with_guard();
    inter(5);
    inter(10);
    inter2(5);
    return 0;
}
```